Visual Studio 2003 Visual Basic and Visual C#

Copyright© 2016 Microsoft Corporation

The content in this document is retired and is no longer updated or supported. Some links might not work. Retired content represents the latest updated version of this content.

Visual Studio .NET

Microsoft Visual Studio.net	Microsoft® Visual Studio® .NET is the comprehensive tool set for rapidly building and integrating XML Web services, Microsoft Windows®–based applications, and Web solutions. Today, the world takes a little step into the future.

Check out the Visual Studio .NET Developer Center

You'll find in-depth content, community resources, downloads, product information, and more at this one-stop resource for Visual Studio .NET developers.

In This Library Section	Essentials
• **Documentation** • **Technical Articles** • **Introducing Visual Studio .NET** • **Developing with Visual Studio .NET** • **Visual Studio Walkthroughs** • **Code Samples**	• **Frequently Asked Questions About Visual Studio .NET** • **Inside the .NET Framework** • **Upgrading to .NET** • **Visual Studio .NET Support Center** • **Newsgroups** • **Communities**

Visual Basic and Visual C#

This section of the documentation discusses application development in Visual Basic or Visual C#. Either programming language can be used equally well in the subjects covered in this section. Applications made by either language are built on the services of the common language runtime and take advantage of the .NET Framework class library.

In This Section

Decision Chart
Provides an interactive guide to help you choose which approach or technology to use, as well as links to more information.
Walkthroughs in Visual Basic and Visual C#
Provides links to topics that walk you through complete scenarios covering different types of applications.
Common Tasks in Visual Basic and Visual C#
Provides links to topics on common programming tasks.
Code for Visual Basic Programmers
Highlights the code examples available within this section of the Visual Basic documentation.
Getting Started
Helps you begin working by listing what is new and by guiding you through version upgrading issues and application design decisions.
Upgrading Applications
Provides information about how to use programs created using previous versions of Visual Basic and Visual InterDev in Visual Basic .NET.
Managing Projects
Describes the project system for managing your applications as you write them.
Creating Applications
Discusses creating and configuring Windows applications, Web applications, and XML Web services, and helps you customize these applications for a global audience.
Accessing Data
Discusses aspects of incorporating data access functionality into your applications.
Programming with Components
Describes components of all kinds, including those you use on servers.
Programming with Office
Provides information on using Microsoft Office and Visual Studio .NET as part of your business application.
Reference
Contains links to several sections of reference information, including programming languages and user interface information.
Samples
Provides sample source code demonstrating how to program using Visual Basic and Visual C#.

Related Sections

Devenv Command Line Switches
Contains language reference topics that explain how to launch Visual Studio and build from the command prompt.
Error Messages
Contains error message topics for Visual Studio.
Automation and Extensibility Reference
Contains automation model language-reference topics for the integrated development environment (IDE), including debugging.
Language Equivalents
Compares keywords, data types, operators, and programmable objects (controls) for Visual Basic, C++, Visual C#, Java, JScript, and Visual FoxPro.
Class Library
Contains syntax and examples for the entire .NET Framework.
User Interface Reference
Contains topics for dialog boxes, windows, and other UI elements found in Visual Studio. Also covers shortcut keys.
Visual Studio Commands
Contains language-reference topics that explain how to use commands to interact with the IDE from the Command window and Find/Command box.
Visual Studio Walkthroughs
Contains topics discussing the steps involved in the development of specific applications types or major application features.
Getting Help from Microsoft Product Support Services
Describes various ways to get technical support.
Accessibility for People with Disabilities

Describes features that make Microsoft Visual Studio more accessible for people with disabilities.

Locating Readme Files

Gives you access to late-breaking notes on the product.

Visual Studio .NET

Links you to major starting points in the Visual Studio documentation.

Decision Chart

When designing your application, you may need help choosing between different technologies or approaches. This topic summarizes the choices and provides links to more information about rapid application development (RAD) with Visual Basic or C#. For more information about which language in Visual Studio to use, see Programming Languages.

Click the text for the option you are interested in; the node will expand to show the choices in that category.

What Do You Want to Do with Visual Basic or Visual C#?

Create a New Application

- **Application** - for help deciding between Windows Forms and Web Forms, see Windows Forms and Web Forms Recommendations
- **Control** - for more information, see Control Authoring for Windows Forms or Web User Controls and Web Custom Controls
- **Command-line application** - for details, see Console Application Template
- **Windows service** (formerly known as an "NT service") - a long-running Windows executable - for details, see Windows Service Applications
- **XML Web service** - XML Web services are applications that can exchange interoperable messages in a loosely coupled environment, using standard protocols such as HTTP, XML, XSD, SOAP and WSDL. XML Web services can be accessed from a Web application, a Windows desktop application, or another XML Web service - for details, see Programming the Web with XML Web Services
- **.NET component** - for details, see Component Authoring

For a step-by-step example, see Walkthrough: Creating a Distributed Application

For details, see Designing Distributed Applications

If you are targeting the browser on a cell phone, PDA (personal digital assistant), or pager, then use Mobile Web Forms - for details, see Creating ASP.NET Mobile Web Applications

Otherwise, use Web Forms - for details see Web Forms Pages

If you ...

- Are navigating between multiple, discrete tables of results ...
- Are manipulating data from multiple sources (for example, from different databases, from XML files, spreadsheets, and so on, all in the same dataset) ...
- Are exchanging data with other applications ...
- Are reusing the same set of rows and intend to cache them (for example, sorting, searching, and filtering cached results) ...
- Are doing a lot of processing per row ...
- Are manipulating the data using XML operations such as XML transforms or XPath ...
- Want ease of programming ...

Then use a dataset - for details, see ADO.NET Datasets

Otherwise, use direct access (data commands and data readers) - for details, see Performing Database Operations Directly.

For example, you would use data commands and data readers for:

- Performing DDL commands
- Performing updates or deletes based on a selection criteria
- Programmatically inserting records that are not based on user interaction
- Calling stored procedures to execute logic inside the server
- Retrieving scalar values from the database
- Processing a set of results in a forward only manner without updating and without caching the results
- Processing a set of results too large to fit in memory

For details, see Web Data Access Strategy Recommendations

For help deciding between the different types of server controls, see ASP.NET Server Controls Recommendations

- **Web server controls** - for details, see ASP.NET Server Controls by Function
- **HTML server controls** - for details, see ASP.NET Server Controls by Function (scroll down in the topic)
- **Create your own server control** - for details, see Web User Controls and Web Custom Controls

XML Web services are applications that can exchange interoperable messages in a loosely coupled environment, using standard protocols such as HTTP, XML, XSD, SOAP and WSDL. An XML Web service can be accessed from desktop applications (Windows Forms), Web applications, or other XML Web services. For details, see Programming the Web with XML Web Services

To send or receive a set of results to or from an XML Web service, use a dataset - for details, see ADO.NET Datasets

To access data within the logic of an XML Web service where that data is not being returned to the caller. . .

If you ...

- Are navigating between multiple, discrete tables of results...
- Are manipulating data from multiple sources (for example, from different databases, from XML files, spreadsheets, and so on, all in the same dataset) ...
- Are exchanging data with other applications ...
- Are reusing the same set of rows and intend to cache them (for example, sorting, searching, and filtering cached results) ...
- Are doing a lot of processing per row ...
- Are manipulating the data using XML operations such as XML transforms or XPath ...
- Want ease of programming ...

Then use a dataset - for details, see ADO.NET Datasets

Otherwise use direct access (data commands and data readers) - for details, see Performing Database Operations Directly.

For example, you would use data commands and data readers for:

- Performing DDL commands
- Performing updates or deletes based on a selection criteria
- Programmatically inserting records that are not based on user interaction
- Calling stored procedures to execute logic inside the server
- Retrieving scalar values from the database
- Processing a set of results in a forward only manner without updating and without caching the results
- Processing a set of results too large to fit in memory

Use Windows Forms - for details, see Creating Windows Applications

If you ...

- Are performing DDL commands ...
- Are calling stored procedures to execute logic inside the server ...
- Are retrieving scalar values from the database ...
- Are processing a set of results in a forward-only manner without displaying, updating, or caching the results ...
- Are processing a set of results too large to fit in memory ...

Then use direct access (data commands and data readers) - for details, see Performing Database Operations Directly

Otherwise, use a dataset - for details, see ADO.NET Datasets.

For example, you would use a dataset if you:

- Are binding results to a form to let the user view, insert, update, or delete records
- Are navigating between multiple tables of results
- Are manipulating data from multiple sources (for example, from different databases, from XML files, spreadsheets, and so on, all in the same dataset)
- Are exchanging data with other applications
- Are reusing the same set of rows (for example, sorting, searching, and filtering retrieved results)
- Are manipulating the data using XML operations such as XML transforms or XPath
- Want ease of programming

- Windows Forms controls - for help choosing between the different controls, see Windows Forms Controls by Function
- Inherit from an existing Windows Forms control (to expand its functionality), inherit directly from the control - for an example, see Walkthrough: Inheriting from a Windows Forms Control with Visual Basic .NET
- Custom controls - for details, see Control Authoring for Windows Forms

For details about working with Windows Forms controls (including securing applications, setting properties, handling events, anchoring controls, and so forth), see Windows Forms Controls

For help deciding which way to create a control, see Control Type Recommendations

For details, see Control Authoring for Windows Forms

- If you want to add to the functionality of an existing Windows Forms control, inherit directly from the Windows Forms control - for a step-by-step example, see Walkthrough: Inheriting from a Windows Forms Control with Visual Basic .NET or Walkthrough: Inheriting from a Windows Forms Control with Visual C#
- If you want to combine existing controls into a new control, inherit from the **UserControl** class - for details, see Inheriting from the UserControl Class
- If you want to create a control from scratch, inherit from the **Control** class - for details, see Inheriting from the Control Class

For details, see Web User Controls and Web Custom Controls

- **XML Web service**, which is technically not a component, but can be called from Windows applications, Web applications, or other XML Web services - for details, see Programming the Web with XML Web Services
- **.NET component** - for details, see Component Authoring

Create a console application - for details, see Console Application Template

Create a Windows service - for details, see Windows Service Applications

Access an Existing XML Web Service

For details, see Accessing XML Web Services in Managed Code

For a step-by-step example of accessing an XML Web service from a Windows application, see Walkthrough: Calling XML Web Services from Windows Forms

For a step-by-step example of accessing an XML Web service from a Windows application, see Walkthrough: Accessing an XML Web Service Using Visual Basic or Visual C#

Program with Objects and Components

For details, see .NET Framework Class Library in Visual Studio

Visual Basic - for details, see Object-Oriented Programming in Visual Basic

C# - for details, see new

Windows Forms Controls - for details, see Windows Forms Controls by Function

Web Forms server controls - for details, see ASP.NET Server Controls by Function

For help deciding between Web and HTML server controls, see ASP.NET Server Controls Recommendations

Web server controls - for details, see Controls You Can Use on Web Forms Pages

HTML server controls - for details, see HTML Server Controls

For help deciding between Web user controls and Web custom controls, see Recommendations for Web User Controls vs. Web Custom Controls

- **Web user control** - for details, see Introduction to Web User Controls
- **Web custom control** - for details, see Introduction to Web Custom Controls

- **Windows services** - for details, see Monitoring Windows Services
- **Processes** - use the Monitoring component - for details, see Monitoring and Managing Windows Processes
- **PerformanceCounter component** - for details, see Monitoring Performance Thresholds
- **EventLog component** - for details, see Logging Application, Server, and Security Events

- **File system** - for details, see Reacting to File System Events
- **Active Directory** - for details, see Creating Active Directory Components

Messaging component - for details, see Creating Messaging Components

- **On the server** - for details, see Creating Timer-Based Server Tasks
- **On the client** - for details, see Timer Component (Windows Forms)

For help deciding which way to create a control, see Control Type Recommendations

For details, see Control Authoring for Windows Forms

- If you want to add to the functionality of an existing Windows Forms control, inherit directly from the Windows Forms control - for an example, see Walkthrough: Inheriting from a Windows Forms Control with Visual Basic .NET
- If you want to combine existing controls into a new control, inherit from the **UserControl** class - for details, see Inheriting from the UserControl Class
- If you want to create a control from scratch, inherit from the **Control** class - for details, see Inheriting from the Control Class

For details, see Developing ASP.NET Server Controls

- **XML Web service**, which technically is not a component, but can be called from Windows applications, Web applications, or other XML Web services - for details, see Programming the Web with XML Web Services
- **.NET component** - for details, see Component Authoring

- **Expose your .NET component to COM** - for details, see Exposing .NET Framework Components to COM
- **Expose your COM component to the .NET Framework** - for details, see Exposing COM Components to the .NET Framework

Search the Web for controls and components created by third-party companies - try the Resource Center at the Got Dot Net Web site (http://www.gotdotnet.com)

Automate Tasks or Extend the Environment

For help deciding whether you need a macro, add-in, or wizard, see The Spectrum of Visual Studio .NET Automation

- **Macro** (Visual Basic only) - for details, see Automating Repetitive Actions by Using Macros
- **Add-in** - for details, see Creating an Add-in
- **Wizard** - for details, see Creating a Wizard

For details, see Introduction to Project Extensibility in Visual Basic and Visual C#

For advanced customization of the development environment, such as creating a new project type or a customized editor - see the " Visual Studio Integrator Program (VSIP)" section of The Spectrum of Visual Studio .NET Automation

Debug, Test, or Troubleshoot

For details, see Debugging Managed Code

For details, see Debugging XML Web Services in Managed Code

For details, see Introduction to Web Application Debugging

For details, see Debugging Preparation: Windows Applications

For details, see Debugging Preparation: Class Libraries

For details, see Debugging Windows Service Applications

For details, see Testing

Testing scalability - for details, see Testing for Scalability

For details, see Troubleshooting Problems Portal

Deploy

For help choosing which deployment project type to use, see Deployment Projects

For details, see Deployment of a Web Setup Project

For an example, see Walkthrough: Deploying a Web Solution

For details, see Deploying XML Web Services in Managed Code

For details, see Setup Projects

For an example, see Walkthrough: Deploying a Windows Application

If you are deploying a .NET component, create a merge module. For details, see Merge Module Projects. For a step-by-step example, see Walkthrough: Creating and Consuming a Merge Module.

If you are deploying an ActiveX control, create a CAB file. For details, see CAB File Projects. For a step-by-step example, see Walkthrough: Creating a CAB File.

Use dynamic properties - for details, see Introduction to Dynamic Properties

Upgrade from a Previous Version

For details, see Upgrading Applications Created in Previous Versions of Visual Basic

For details, see Upgrading Visual InterDev 6.0 Applications to Visual Studio .NET

Learn about New Features

What's New in Visual Basic and Visual C#

What's New in Visual Studio .NET

What's New in Deployment

Introduction to Visual Basic .NET for Visual Basic Veterans

Inside the .NET Framework

Walkthroughs in Visual Basic and Visual C#

Walkthrough: Creating a Distributed Application

Data Walkthroughs

XML Walkthroughs

XML Web Service Walkthroughs (XML Web services and Web applications)

Windows Forms Walkthroughs

Component Programming Walkthroughs

Deployment Walkthroughs

Visual Basic Samples

Visual C# Samples

See Also

Common Tasks in Visual Basic and Visual C# | Choosing Tools and Technologies |.NET Framework Class Library in Visual Studio

Walkthroughs in Visual Basic and Visual C#

Walkthroughs give step-by-step instructions for common scenarios, which makes them a good place to start learning about the product or a particular feature area.

Walkthroughs are presented in the following areas:

• Distributed Applications Walkthrough • Data and XML Walkthroughs • XML Web Services Walkthroughs • Windows Application Walkthroughs • Web Walkthroughs • Visual Basic Language Walkthroughs • Deployment Walkthroughs	• Component and Control Authoring Walkthroughs • Dynamic Properties Walkthrough • Internationalization Walkthroughs • Accessibility Walkthroughs • Framework Services Component Walkthroughs • Upgrading Walkthrough • Office Development Walkthroughs

Note Some features listed in the walkthrough topics are not available in the Standard Edition of Visual Basic and Visual C# .NET. For more information, see Visual Basic Standard Edition Features or Visual C# Standard Edition Features.

Distributed Applications Walkthrough

Creating a Distributed Application
Explains how to create a multitiered, distributed application that has both a Windows client and a browser interface.

Data and XML Walkthroughs

Simple Data Access in a Windows Form
Describes the basic steps required for accessing SQL Server data via a dataset in a read-write Windows Form.
Creating a Dataset with Tables, Keys, and One-to-Many Relationships
Describes the steps to create a dataset and use it to validate data in a one-to-many or keyed relationship.
Displaying Data in a Windows Form Using a Parameterized Query
Describes how to create a dataset containing only selected records, based on criteria that users provide at run time in a Windows Form.
Creating a Master-Detail Windows Form
Describes how to create a parent-child relationship in a dataset and display related records in a Windows Form.
Reading XML Data into a Dataset
Shows how to load XML data into a dataset, display it in a Windows Forms **DataGrid** control, and then display an XML schema for the file in a text box.
Mapping Data Source Tables to Dataset Tables
Describes how to load data from a database into a dataset based on a different schema using data adapter table and column mappings.
Handling a Concurrency Exception
Describes how to use the **DBConcurrencyException** object to identify concurrency exceptions and the actual record that caused the error.
Displaying Data in a Web Forms Page
Illustrates a basic data-access scenario, providing step-by-step instructions for how to create a Web Forms page that displays data in a grid control.
Creating Read-Only Data Access in a Web Forms Page
Describes how use an ADO.NET data command and data reader to provide optimized access to read-only data in a Web Forms page.
Updating Data Using a Database Update Query in Web Forms
Describes the basic techniques of creating a Web Forms page that uses a data command to update the database.
Using a DataGrid Web Control to Read and Write Data
Gives step-by-step details on how to use a grid control to allow users to view and edit data.
Creating Paged Data Access Using a Web Forms Page
Describes how to create a Web Forms page that allows users to view data records a few records at a time, paging back and forth to view earlier and later records.
Displaying an XML Document in a Web Forms Page using Transformations
Gives step-by-step details on how to read an XML file and display it in different ways on a Web Forms page.
Creating an XML File with an Associated XML Schema

Describes the steps for creating an XML file and a schema, associating the two, and working on the XML file in the designer's XML view.

Creating an XML Schema with the XML Designer

Describes the steps to create an XML purchase order schema as part of a Windows application.

XML Web Services Walkthroughs

Creating an XML Web Service Using Visual Basic or Visual C#

Explains how to create an XML Web service that converts temperatures measured in Fahrenheit to Celsius.

Accessing an XML Web Service Using Visual Basic or Visual C#

Describes how to make use of an existing XML Web service.

Calling XML Web Services from Windows Forms

Describes how to make calls to Web service methods from a Windows application.

Windows Application Walkthroughs

Creating a Simple Windows Form

Describes how to build and run a simple Windows Form.

Creating a Master-Detail Windows Form

Describes how to create a parent-child relationship in a dataset and display related records in a Windows Form.

Creating Dynamic Context Menus on Windows Forms

Steps through a detailed example of creating a single context menu to serve two or more different controls.

Switching Windows Forms Menu Structures Based on Application State

Gives directions for switching between **MainMenu** objects programmatically.

Demonstrating Visual Inheritance

Describes how to create a base Windows Form and compile it into a class library. You will import this class library into another project, and create a new form that inherits from the base form.

Creating a Multipane User Interface with Windows Forms

Describes how to create a multipane user interface that is similar to the one used in Microsoft Outlook.

Simple Data Access in a Windows Form

Describes the basic steps required for accessing SQL Server data via a dataset in a read-write Windows Form.

Displaying Data in a Windows Form Using a Parameterized Query

Describes how to create a dataset containing only selected records, based on criteria that users provide at run time in a Windows Form.

Calling XML Web Services from Windows Forms

Describes how to make calls to Web service methods from a Windows application.

Creating a Windows Service Application in the Component Designer

Describes how to create a long-running executable file that does not have a user interface.

Persisting an Object

Demonstrates how serialization can be used to persist an object's data between instances, allowing you to store values and retrieve them the next time the object is instantiated.

Retrieving Dialog Box Information Collectively Using Objects

Gives directions for using an object to expose a group of related data from a dialog box.

Updating Status-Bar Information at Run Time

Gives directions for programmatically controlling the data within status bar panels.

Web Walkthroughs

Creating a Basic Web Forms Page

Illustrates the basic techniques of how to create a Web Forms page in Visual Basic or Visual C#, add controls to it, and run it.

Creating a Web Application Using Visual C# or Visual Basic

Describes how to write a Web Forms application and then integrate a business-object component that gives you a 10% discount for purchases.

Creating a Web Application Using a Third-Party Business Object

Describes how to write a Web Forms application using Managed Extensions for C++ and then integrate a third-party business-object component that gives you a 10% discount for purchases.

Validating User Input in a Web Forms Page

Describes how to use Web Forms validation controls to check user entries without code, including required entries, data types, ranges, patterns, and specific values.

Displaying Data in a Web Forms Page

Illustrates a basic data-access scenario, providing step-by-step instructions for how to create a Web Forms page that displays data in a grid control.

Creating Read-Only Data Access in a Web Forms Page
Describes how use an ADO.NET data command and data reader to provide optimized access to read-only data in a Web Forms page.
Updating Data Using a Database Update Query in Web Forms
Describes the basic techniques of creating a Web Forms page that uses a data command to update the database.
Using a DataGrid Web Control to Read and Write Data
Gives step-by-step details on how to use a grid control to allow users to view and edit data.
Creating Paged Data Access using a Web Forms Page
Describes how to create a Web Forms page that allows users to view data records a few records at a time, paging back and forth to view earlier and later records.
Creating a Web User Control
Describes how to create a Web Forms user control — a page configured to act as a control — and use it on another page.
Converting a Web Forms Page to a User Control
Describes how to convert a basic Web Forms page into a user control that can be hosted on another page.
Creating a Web Custom Control
Describes how to create a custom control, add it to the toolbox, and use it on a Web Forms page.
Displaying an XML Document in a Web Forms Page using Transformations
Gives step-by-step details on how to read an XML file and display it in different ways on a Web Forms page.

Visual Basic Language Walkthroughs

Declaring and Raising Events
Illustrates how events are declared and raised in Visual Basic .NET.
Handling Events
Shows how to handle events using either the standard **WithEvents** keyword or the new **AddHandler/RemoveHandler** keywords.
Structured Exception Handling
Demonstrates how to create and maintain programs with comprehensive error handling, and how to use **Try...Catch...Finally**.
Defining Classes
Describes how to declare a class and its fields, properties, methods, and events.
Creating and Implementing Interfaces
Shows how interfaces are declared and implemented in Visual Basic .NET.
Multithreading
Shows how to create a multithreaded application that that searches a text file for occurrences of a word.
Calling Windows APIs
Explains how to use **Declare** statements and call Windows APIs. Includes information about using attributes to control marshaling for the API call and how to expose an API call as a method of a class.
Creating COM Objects with Visual Basic .NET
Demonstrates how to create COM objects with Visual Basic .NET, with or without the COM class template.
Implementing Inheritance with COM Objects
Demonstrates how to use Visual Basic 6.0 to create a COM object containing a class, and then use it as a base class in Visual Basic .NET.
Interacting with Files and Folders in Visual Basic .NET
Provides an introduction to the fundamentals of file I/O.
Manipulating Files Using .NET Framework Methods
Demonstrates how to read and write to a text file using the .NET Framework classes, Streamreader and Streamwriter.

Deployment Walkthroughs

Deploying a Web Solution
Guides you through deploying a Web application to a Web server.
Deploying a Windows Application
Guides you through deploying a Windows application to another computer.
Creating a CAB File
Guides you through distributing an ActiveX control.
Creating a Custom Action
Guides you through directing a user to a Web site following installation.
Passing Data to a Custom Action
Demonstrates passing data to a dynamic property during installation using a custom action and the **CustomActionData** property.
Using a Custom Action to Create a Database During Installation
Guides you through creating a custom action to create a database during installation.

Using a Custom Action to Pre-compile an Assembly During Installation
Guides you through creating a custom action to pre-compile an assembly after installation.
Creating and Consuming a Merge Module
Guides you through packaging a component in a merge module and then including the merge module in an installer.
Redirecting an Application to Target a Different XML Web Service During Installation
Demonstrates how to create a Web application that can be redirected to target a different XML Web service by using the URL Behavior property, an Installer class, and a Web Setup project.

Component and Control Authoring Walkthroughs

Authoring a Component with Visual Basic .NET
Shows the development of a simple component in Visual Basic, illustrating the interaction between client and component, object lifetime and circular references, debugging clients and components, and the use of shared methods and instance methods.
Authoring a Component with Visual C#
Shows the development of a simple component in Visual C#, illustrating the interaction between client and component, object lifetime and circular references, debugging clients and components, and the use of shared methods and instance methods.
Creating Your Own Collection Class
Shows how you can create your own collection classes by inheriting from one of the many .NET Framework collection classes and adding code to implement your own custom functionality.
Authoring a User Control with Visual Basic .NET
Shows the development of a simple control in Visual Basic that inherits from the **UserControl** class, and how to inherit from an established user control.
Authoring a User Control with Visual C#
Shows the development of a simple control in Visual C# that inherits from the **UserControl** class, and how to inherit from an established user control.
Inheriting from a Windows Forms Control with Visual Basic .NET
Shows the development of a simple control in Visual Basic that inherits from a pre-existing Windows Forms control.
Inheriting from a Windows Forms Control with Visual C#
Shows the development of a simple control in Visual C# that inherits from a pre-existing Windows Forms control.
Authoring a Simple Multithreaded Component with Visual Basic
Demonstrates the development of a simple multithreaded component in Visual Basic, explaining how threads work and how to coordinate multiple threads in your component.
Authoring a Simple Multithreaded Component with Visual C#
Demonstrates the development of a simple multithreaded component in Visual C#, explaining how threads work and how to coordinate multiple threads in your component.

Dynamic Properties Walkthrough

Storing and Retrieving Dynamic Properties
Describes the major steps in storing and retrieving property values.

Internationalization Walkthroughs

Localizing Web Forms Pages
Describes the major steps in creating a global-ready Web Forms application.
Localizing Windows Forms
Describes the major steps in creating a global-ready Windows Forms application.

Accessibility Walkthroughs

Creating an Accessible Web Application
Describes how to create an ASP.NET Web application that is optimized for different accessibility issues.
Creating an Accessible Windows Application
Describes how to create a Windows application that is optimized for different accessibility issues.

Framework Services Component Walkthroughs

Creating a Queue and Working with Messages
Describes how to use the **MessageQueue** component to interact with Windows message queues.
Creating a Message Queue Transaction with COM+ Services
Describes how to create a message queue transaction hosted by Enterprise Services (COM+ services).

Exploring Event Logs, Event Sources, and Entries

Describes how to use the **EventLog** component to interact with existing logs, create custom logs, and read and write values to logs.

Retrieving Categories and Counters

Describes how to use the **PerformanceCounter** component to retrieve lists of Windows performance counter categories on a server.

Changing and Retrieving Performance Counter Values

Describes how to use the **PerformanceCounter** component to retrieve and set raw and calculated values from a Windows performance counter.

Managing a Windows Process

Describes how to use the **Process** component to monitor and interact with Windows processes on a server.

Reacting to File System Events

Describes how to use the **FileSystemWatcher** component to monitor files and directories and react when changes occur.

Adding ActiveDirectory Objects

Describes how to use the **DirectorySearcher** component to interact with Active Directory file system structures.

Installing an Event Log Component

Describes how to use an installation component to configure server resources needed for your **EventLog** component instance.

Upgrading Walkthrough

Upgrading a Visual Basic 6.0 Application to Visual Basic .NET

Describes how to upgrade a standard application from Visual Basic 6.0 to Visual Basic .NET.

Office Development Walkthroughs

Programming with Office

Describes how to use Microsoft Office and Visual Studio .NET as part of your business application.

See Also

Samples and Walkthroughs | Common Tasks in Visual Basic and Visual C# | Visual Basic Code Example Topics

Common Tasks in Visual Basic and Visual C#

This topic is designed to help you quickly find the answers to your most frequent questions about programming in Visual Basic .NET. The following areas are represented in the list below:

• Projects • Windows Forms and Windows Applications • Web Forms and Web Applications • XML Web Services • Component Creation • Windows Services	• Data • XML • Server Components • Debugging • Deployment • Language (Visual Basic)

Note Some features listed in this topic are not available in the Standard Editions of Visual Basic and Visual C# .NET. For more information, see Visual Basic Standard Edition Features or Visual C# Standard Edition Features.

Projects

How do I ...

- Upgrade my Visual Basic 6.0 project to Visual Basic .NET?
- Upgrade from J++ to C#?
- Create a new project?
- Reference a .NET or COM component?
- Reference an XML Web service?
- Add new files to my project?
- Adding existing files to my project?
- Add resources to my application?
- Make a copy of a project?

Windows Forms and Windows Applications

How do I ...

- Create a Windows application?
- Add a Windows Form to my application?
- Create a form that inherits from another Windows Form?
- Add controls to a Windows Form?
- Add controls at run time?
- Create a dynamic layout for my Windows Form?
- Choose the form to display when my application starts?
- Handle events on a Windows Form?
- Handle multiple events with the same procedure?
- Display message box prompts?
- Display modal dialog boxes?
- Add menus and menu items to my application?
- Create an MDI application?
- Draw lines and shapes?
- Create a Windows application in another language?
- Provide Help for my application?

Web Forms Pages and Web Applications

How do I ...

- Create a Web application?

- Add a Web Forms page to my application?
- Add controls to a Web Forms page?
- Add HTML controls to a Web Forms page?
- Handle events on a Web Forms page?
- Use styles to customize the appearance of my server controls?
- Reference controls on a Web Forms Page?
- Make sure that users fill in the required fields on my Web Forms page?
- Save Web Forms page values between round trips to the server?
- Create user controls and custom controls?
- View my Web Forms pages in a browser?
- Create a Web Forms page that will work with different browsers?
- Localize a form for different languages?
- Select an encoding for my Web pages?
- Secure my Web application?
- Protect against script exploits in my Web application?
- Display safe error messages?
- Use encoding to open and save files?
- Access SQL server as a local user?
- Access SQL server using Windows integrated security?

XML Web Services

How do I ...

- Create an XML Web service?
- Access an XML Web service so I can use it in my application?
- Add methods to my XML Web service?
- Publish my XML Web service so that others can find it?
- Secure my XML Web service?
- Handle and raise exceptions in XML Web services?

Component Creation

How do I ...

- Create a reusable library of classes?
- Create a Windows Forms user control in my project?
- Create a Windows Forms custom control in my project?
- Create a Web Forms user control in my project?
- Create a Web Forms custom control in my project?
- Use COM+ services for my class?

Windows Services

How do I ...

- Create a Windows service?
- Monitor the activities of an existing Windows service?
- Install a Windows service?
- Start a Windows service?

Data

How do I ...

- Decide which data access strategy to use?

- Create a connection to a database?
- Retrieve data from a database?
- Add new data to the data I have retrieved?
- Write changed data back to my database?
- Filter rows in my retrieved data?
- Update a database with the results of my work?
- Execute a command against a database?
- Check for and handle concurrency errors in my data?
- Bind to data in a Windows Form?
- Bind to data in Web Forms page?

XML

How do I ...

- Create an XML file?
- Validate an XML file?
- Create an XML schema?
- Transform XML on a Web Forms page?

Server Components

How do I ...

- Monitor changes to my file system?
- Monitor and control a Windows process?
- Use a timer?
- Read from and write to event logs?
- Create my own performance counters?
- Read from and write to performance counters?
- Create my own message queues?
- Read entries from a message queue?
- Send messages to a queue?

Debugging and Tracing

How do I ...

- Debug a Windows application?
- Debug a Web application?
- Debug an XML Web service?
- Debug a Windows service?
- Debug a class library or user control?
- Set breakpoints?
- Monitor the values of variables?
- Add tracing functionality to my application?
- Read from and write to event logs?

Deployment

How do I ...

- Work with multiple versions of the .NET Framework?
- Specify the runtime version for my application?
- Create a setup program for my application?
- Add a custom installer?

- Create a CAB file?
- Deploy my application over the Web?
- Make a property value configurable after deployment?

Language (Visual Basic)

How do I ...

- Declare a variable?
- Write an If condition?
- Write a For loop?
- Write a Case condition?
- Write a While loop?
- Enumerate a collection or array?
- Convert the type of a variable?
- Define and use an array?
- Define a class?
- Define a property?
- Define a method?
- Define an event?
- Pass an argument?
- Inherit from a class?
- Define a collection?
- Define and implement an interface?
- Define a user-defined type?
- Define a code module?
- Handle events?
- Handle exceptions?
- Perform string operations?
- Perform numeric calculations?
- Use custom attributes?
- Call a Win32 API?
- Read and write to files?
- Create a thread?
- Interact with COM components?
- Put comments in my code?
- Retrieve an embedded resource at run time?

See Also

Keywords and Members by Task | Decision Chart | Walkthroughs in Visual Basic and Visual C# | What's New in Visual Basic and Visual C# | Visual Basic and Visual C# | Visual Basic Code Example Topics

Code for Visual Basic Programmers

There are many sources of code within the Visual Basic documentation set. This topic helps you get started in locating that code. Note that some of this code is contained within topics that provide further explanation or instructions on how to perform a task, whereas some is contained in special topics devoted only to displaying a code example.

Code in the Visual Basic Documentation

The following topics highlight useful code samples, by feature, within the Visual Basic documentation.

Code for Windows Forms Applications
: Lists code examples in the "Creating Windows Applications" section of the documentation.

Code for Windows Forms Controls
: Lists code examples in the "Windows Forms Controls" section of the documentation.

Code for Data Access
: Lists code examples in the "Accessing Data" section of the documentation.

Code for Web Applications
: Lists code examples in the "Creating Web Applications and Services" section of the documentation.

Code for Web Forms Controls
: Lists code examples in the "ASP.NET Server Controls" section of the documentation

Code for Component Programming
: Lists code examples in the "Programming with Components" section of the documentation.

Other Sources for Sample Code

You can also find sample code in the following locations:

Visual Basic Code Example Topics
: Lists quickstart-like code examples that help you perform common tasks in Visual Basic .NET.

Visual Basic Language Tour
: Provides links to topics that contain code showing how to use basic language features, such as threading, inheritance, and file IO.

Visual Basic Sample Applications
: Lists compiled sample applications available to you in the documentation set.

Walkthroughs in Visual Basic and Visual C#
: Lists tutorial topics that provide step-by-step instructions on the process of creating applications and learning to use specific features.

Visual Basic QuickStarts and Other Samples
: Lists samples available to you when you install the .NET Framework SDK.

.NET Framework Reference Information
: Provides links to reference topics, some of which code examples of how to use classes and their members in.

Getting Started

This section of the documentation helps you get started with Visual Basic and Visual C# application development.

In This Section

Additional Resources for Visual Basic Programmers
Lists sites and newsgroups that can help you find answers to common Visual Basic problems.
Additional Resources for Visual C# Programmers
Lists sites and newsgroups that can help you find answers to common Visual C# problems.
What's New in Visual Basic and Visual C#
Covers new features for different areas, including Web development, data access, component creation, and the Visual Basic and Visual C# languages.
Visual Basic Standard Edition Features
Describes the features of the standard edition of Visual Basic.
Visual C# .NET Standard Edition
Provides links to descriptions of the features of the standard edition of Visual C#.

Related Sections

Decision Chart
Provides an interactive guide to help you choose which approach or technology to use, as well as links to more information.
Walkthroughs in Visual Basic and Visual C#
Provides links to topics that walk you through complete scenarios covering different types of applications.
Common Tasks in Visual Basic and Visual C#
Provides links to topics on common programming tasks.
Code for Visual Basic Programmers
Highlights the code examples available within this section of the Visual Basic documentation.
.NET Framework Class Library in Visual Studio
Explains essential concepts and lists the most important namespaces for each feature area, and provides links to more information.
Samples
Provides links to sample code in both Visual Basic and Visual C#.
Using Help in Visual Studio .NET
Explains how to use filters and provides links to topics about searching Help, using the index, and finding information in the MSDN Library.
Customizing the Development Environment
Covers how to set options for the integrated development environment (IDE), such as shortcut keys and debugging preferences.
Product Support
Covers how to receive support and give feedback.
Visual Studio .NET
Provides links into the Visual Studio documentation.
Visual Studio .NET Editions
Lists the features of the three editions of Visual Studio.
Visual Basic and Visual C#
Provides links into the documentation on rapid application development.
Visual C++
Provides links into the Visual C++ documentation.
Introducing Visual J#
Provides an overview of Visual J#, as well as a description of the Visual J# architecture.
Programming with Office
Provides information on using Microsoft Office and Visual Studio .NET as part of your business application.

Additional Resources for Visual Basic Programmers

The following Web sites and newsgroups can help you find answers to common problems.

Microsoft Resources

On the Web

http://support.microsoft.com
Provides access to Knowledge Base (KB) articles, downloads and updates, support Webcasts, and other services.
http://www.gotdotnet.com/team/vb
Contains articles, samples, and other information of interest to Visual Basic .NET developers.
http://msdn.microsoft.com/vbasic
Provides code samples, upgrade information, and technical content.
http://msdn.microsoft.com/newsgroups
Provides a way to connect as a community with experts from around the world.
http://www.asp.net
Provides articles, demonstrations, tool previews, and other information for Web development in Visual Basic.

In Newsgroups

microsoft.public.dotnet.languages.vb
Provides a forum for questions and general discussion of Visual Basic .NET.
microsoft.public.dotnet.languages.vb.upgrade
Provides a forum for questions and issues on upgrading to Visual Basic .NET.
microsoft.public.vsnet.general
Provides a forum for questions and issues on Visual Studio .NET.
microsoft.public.vsnet.ide
Provides a forum for questions about working in the Visual Studio environment.
microsoft.public.vsnet.documentation
Provides a forum for questions and issues on the Visual Basic .NET documentation.

Third-Party Resources

MSDN's Web site provides information on current third-party sites and newsgroups of interest. For the most current list of resources available, see the MSDN Community Web site (http://msdn.microsoft.com/vbasic/community).

On the Web

Universal Thread (http://www.universalthread.com/net)
Provides the latest news, interviews, and hints on Visual Basic .NET.
DevX Visual Basic Zone (http://www.devx.com/vb)
Provides in-depth technical articles for today's Visual Basic developer moving to Visual Basic .NET.
VbCity (http://www.devcity.net/net)
Provides access to more than 5,000 Microsoft Visual Basic and Active Server Pages (ASP) articles, source code, tutorials, and user-submitted content when you join vbCity.com.

Newsgroups

vb.dotnet.technical (http://news.devx.com)
Provides a forum for discussion of new features in Visual Basic .NET in the DevX forums.

Additional Resources for Visual C# Programmers

The following sites and newsgroups can help you find answers to common problems.

Microsoft Resources

On the Web

http://support.microsoft.com/
Provides access to KB articles, downloads and updates, support Webcasts, and other services.
http://www.gotdotnet.com/team/csharp/
Contains articles, samples, and other information of interest to Visual C# .NET developers who are targeting the .NET Framework and common language runtime.
http://msdn.microsoft.com/vcsharp/
Provides code samples, upgrade information, and technical content.
http://msdn.microsoft.com/newsgroups/
Provides a way to connect as a community with experts from around the world.

In Newsgroups

microsoft.public.dotnet.languages.csharp
Provides a forum for questions and general discussion of Visual C# .NET.
microsoft.public.vsnet.general
Provides a forum for questions and issues on Visual Studio .NET.
microsoft.public.vsnet.ide
Provides a forum for questions about working in the Visual Studio environment.
microsoft.public.vsnet.documentation
Provides a forum for questions and issues on the Visual C# .NET documentation.

Third-Party Resources

MSDN's Web site provides information on current third-party sites and newsgroups of interest. For the most current list of resources available, see the MSDN Community Web site (http://msdn.microsoft.com/vcsharp/community)).

On the Web

http://www.c-sharpcorner.com
Provides an online community for developers using C# to target the .NET Framework.
http://www.codeproject.com
Provides a community of Windows developers specializing in C# and .NET where source code, articles, and tutorials are shared.
http://www.codeguru.com/
Provides information about programming with C# and targeting the .NET Framework.

What's New in Visual Basic and Visual C#

Visual Basic and C# both have RAD (rapid-application development) support in Visual Studio .NET, with project templates, designers, and other features of the development environment. Both languages use the .NET Framework base classes.

In This Section

What's New in Web Development
Visual Basic and C# include support for Web Forms and XML Web services. Web Forms allow you to easily and quickly create the browser-based interface for ASP.NET Web applications. XML Web services allow you to package a Visual Basic or C# method and make it accessible on the Web.

What's New in Data
ADO.NET provides scalable, high-performance data access for all Visual Studio applications based on the .NET Framework. You can access data directly in the data source or create an in-memory cache — a dataset — for working with disconnected data. You can also work directly with XML as relational data or by creating and editing XML files directly.

What's New in Windows Forms and Controls
You can use Windows Forms and controls to create the presentation layer of a distributed application. Windows Forms provides a clear, object-oriented, extensible set of classes that enable you to develop rich Windows applications.

What's New in Projects
You can use the Project templates in this version to easily create various types of Windows and ASP.NET Web applications and controls. Project templates set necessary references for you, import namespaces, and add default items to your projects.

What's New in Components and Component Authoring
You can use nonvisual components and associated features to incorporate resources such as message queues, event logs, and performance counters into your applications. This version also offers RAD support for component creation via the Component Designer and framework classes that help you create controls and components.

What's New in Debugging
You can now debug projects that are part of the same solution but written in different languages, perform remote debugging, and implement trace functionality throughout your code to retrieve finely tuned output.

What's New in Extensibility and Automation
Visual Studio .NET includes a programmable object model that provides access to the underlying components and events of the integrated development environment (IDE). This model allows you to extend the functionality of the IDE, automate repetitive tasks, and integrate the IDE with other applications.

What's New in International Applications
Internationalization features in Visual Basic .NET make it easy to create applications targeting multiple locales, building on the globalization and localization features of the .NET Framework.

What's New in Upgrading
Visual Basic .NET 2003 now includes support for upgrading Visual Basic 6.0 WebClass and UserControl projects, plus a new tool for upgrading blocks of Visual Basic 6.0 code. Visual Studio .NET 2003 also includes the Java Language Conversion Assistant, which you can use to convert your Visual J++ 6.0 and Java-language projects to Visual C# .NET.

Related Sections

What's New in the Visual Basic Language
Visual Basic .NET, the next generation of the Visual Basic language, is designed to be a fast and easy way to create .NET applications, including XML Web services and ASP.NET Web applications.

Visual Basic .NET has many new and improved features such as inheritance, interfaces, and overloading that make it a powerful object-oriented programming language. Other new language features include free threading and structured exception handling. Visual Basic .NET fully integrates the .NET Framework and the Common Language Runtime, which together provide language interoperability, garbage collection, enhanced security, and improved versioning support.

Visual C# Language
Visual C# (pronounced C sharp) is a new programming language introduced in Visual Studio .NET. An evolution of C and C++, C# is simple, modern, type-safe, and object-oriented. It was designed for building a wide range of enterprise applications that run on the .NET Platform. Code that you write with Visual C# is compiled as managed code, which means it benefits from the services of the Common Language Runtime. These services include language interoperability, garbage collection, enhanced security, and improved versioning support.

Visual C# is fully supported within Visual Studio .NET by project templates, designers, property pages, code assistants, an object model, and other features of the development environment. The library for Visual C# programming is the .NET Framework.

What's New in Visual C# .NET 2003

Discusses the programmatic improvements in Visual C#, including enhancements to IntelliSense and the development environment.

What's New in Visual Studio .NET

Summarizes new features in the shared Visual Studio IDE used by Visual Basic, Visual C#, Visual C++, XML Web services, XML support, sample applications, and Help.

What's New in Deployment

Describes the new deployment features of Visual Studio .NET. Using the new deployment features you can distribute applications using Microsoft Windows Installer technology, deploy to production and staging servers, deploy tiers of your application to different test computers, and deploy ASP.NET Web applications to Web servers.

Inside the .NET Framework

Describes features of the .NET Framework, including the common language runtime, managed versus unmanaged execution, the common type system, and the .NET Framework class library.

What's New in Visual Studio Analyzer?

Describes the features of Visual Studio Analyzer, which analyzes the performance of distributed applications and performs debugging.

What's New in Visual Database Tools

Describes the Visual Database Tools, including the Database Designer, the SQL Editor, and Server Explorer. You can use these tools to help create and manage data-driven applications.

Introducing Visual J#

Provides an overview of the capabilities, features, and architecture of the Visual J# development system.

What's New in Web Development

An important new feature of Visual Studio is the ability to create Web applications — that is, Visual Basic and Visual C# applications that run on a Web server.

What's New in Visual Studio .NET 2003

In Visual Studio .NET 2003, there are small changes to Web Forms and the Web Forms Designer, all of which are designed to make Web Forms more robust and easier to develop.

Request Validation
By default, the page framework automatically scans posted user information for HTML elements, including tags and reserved characters. If it detects HTML, it raises an error. This provides protection against potentially malicious user input ("script injection"). For details, see Scripting Exploits.

Consolidated View and Build Commands
The **View in Browser** command (available when you right-click a Web Forms page in Solution Explorer) has been consolidated with the **Build and Browse** command. The **View in Browser** command now builds before displaying the page in a docked window in the designer. (Formerly it did not compile before displaying the page.) This change makes the **Build and Browse** command unnecessary, so that command has been removed from the shortcut menu.

Visual J# .NET Support
You can now develop Web Forms pages using Visual J# .NET. For more information, see Visual J# Overview.

Web References
The Start Browsing for XML Web Services pane in the Add Web Reference Dialog Box now provides Web links to both local and Internet sources of available XML Web services, including your own machine, UDDI (Universal Description, Discovery and Integration) servers on the local network, the Microsoft UDDI Business Registry, and the Test Microsoft UDDI Directory. For more information, see Locating XML Web Services and Adding and Removing Web References.

What's New in Visual Studio .NET 2002

The two major developments in Web applications are Web Forms and XML Web services:

- Web Forms technology allows you to quickly and easily create the user interface for ASP.NET Web applications. Web Forms pages represent a revolutionary improvement over existing Web-development tools, combining the speed and ease of a rapid application development (RAD) environment with the power of compiled programming languages. Web Forms output is compatible with any browser or mobile device and automatically renders the correct browser-compliant HTML for features such as styles, layout, and so on. For more information see Web Forms Pages.
- An XML Web service is a piece of business functionality — a component — that can be called by other components or applications via Internet protocols. XML Web services enable the exchange of data using standards like HTTP and XML messaging to move data across firewalls. XML Web services are not tied to a particular component technology or object-calling convention. As a result, programs written in any language, using any component model, and running on any operating system can access XML Web services. For more information see XML Web Services in Managed Code.

> **Visual Basic Note** If you are familiar with the Visual Basic language, see Language Changes in Visual Basic for a list of differences from previous versions.

RAD Support for Creating Web Forms Pages
Visual Studio includes a forms designer for creating HTML and ASP.NET-based forms. The designer allows you to drag HTML elements, controls, data classes, and other objects onto your form, use the Properties window to work with them, and use RAD techniques for adding code to them. For more details, see Creating and Managing Web Forms Pages.

RAD Support for Creating XML Web Services
Visual Studio includes a project template for creating XML Web services, in which you can expose methods for public access on the Web. You can create and distribute XML Web services using Visual Basic, Visual C#, or ATL server. For information, see Introduction to Programming XML Web Services in Managed Code.

RAD Support for Accessing XML Web Services
Visual Studio contains features that make it easy for you to locate existing XML Web services and make use of them in your applications. The Add Web Reference dialog box can be used to navigate services and add the necessary proxies to your project to access them. For details, see Accessing XML Web Services in Managed Code.

Programming Language Support
You can program Web Forms pages in any language supported in the .NET Framework, including Visual Basic .NET and Visual C# .NET. As in all .NET programming, you can create components in different languages; for example, you can create a Web Forms page in Visual Basic and call components written in Visual C#. For details, see Programming Languages.

Server Controls

When creating Web Forms pages, you use a special class of controls designed for use in browsers. The controls render as HTML, but execute code on the server, giving you maximum flexibility and power over the form. For details, see ASP.NET Server Controls.

Validation

You can create user input validation easily using a set of controls that automatically read user input, perform a validation check, and produce error information. For details, see Web Forms Validation.

Data Access in Web Applications

You can bind controls to a variety of data sources, ranging from simple arrays to datasets created in the ASP.NET Framework. The Web Forms Designer allows you to drag data objects onto the form and configure them. You can then bind any property of a Web Forms server control to any available data element. For more information, see Data Access in Web Forms Pages.

Compiling, Deploying, and Running Web Forms Pages

Web Forms pages are compiled into classes that are deployed as DLLs, giving you both performance and code security in your distributed application.

Debugging Web Applications

The Visual Studio .NET debugger is fully supported for XML Web services and for Web Forms pages. You can use the Visual Studio to debug your Web applications, including code that is running on a remote computer. For more information, see Introduction to Web Application Debugging.

Custom Web Control Creation

You can create your own Web server controls in two ways:

- **User Controls** You can create visual controls as Web Forms pages encapsulating the functionality you want, then embed those pages in other Web Forms pages. User Controls provide you with simple, easily developed controls that are ideal for components such as menus and toolbars. For more information, see Introduction to Web User Controls.
- **Custom server controls** Using an SDK provided with the .NET Framework, you can create your own controls that are fully integrated into the Visual Studio .NET environment. Developing custom controls allows you to create the design-time and run-time look of the control, specify property editors, set right-click behavior, and so on. For details, see Introduction to Web Custom Controls.

See Also

What's New in Visual Basic and Visual C# | What's New in Visual Studio .NET | What's New in the Visual Basic Language | C# Language Tour | What's New in Data

What's New in Data

ADO.NET is an evolutionary improvement to Microsoft ActiveX Data Objects (ADO). It represents a standards-based programming model that facilitates the development of distributed applications, especially Web applications. ADO.NET uses standards-based XML as a medium for data transfer, making it particularly appropriate for exchanging data between components and applications, even on unlike platforms.

Visual Basic Note If you are familiar with data access in Visual Basic 6.0, see Data Changes in Visual Basic .NET.

What's New in Visual Basic .NET 2003

Visual Basic .NET 2003 includes two new .NET Framework data providers and enhanced support for large XML Schemas.

.NET Framework Data Provider for ODBC

The .NET Framework Data Provider for ODBC provides optimized performance when working with ODBC data sources. For more information, see .NET Framework Data Providers.

.NET Framework Data Provider for Oracle

The .NET Framework Data Provider for Oracle provides optimized performance when working with Oracle databases. For more information, see .NET Framework Data Providers.

Improved Design-Time Experience for Working with Large XML Schemas

The XML Designer provides improved viewing options when working with schemas that contain many elements. A new zooming feature has been added that enables fine control over how schemas are viewed in the designer. The ability to expand and contract elements in a nested hierarchy has also been added to the designer. For more information, see XML Designer.

What's New in Visual Basic .NET 2002

Visual Basic .NET 2003 also includes the following features, which were introduced in Visual Basic .NET 2002.

Access Data Sources with High-Performance Data Components

ADO.NET provides high-performance .NET Framework data providers for direct access to databases and other data sources. Data providers give you precise control over SQL commands and stored procedures that can return data or scalar values or perform other database operations. Data reader objects provide an extremely fast way to return read-only data from a data source. For more information, see .NET Framework Data Providers.

Work with Disconnected Data

To make the most efficient use of database and server resources, you can create datasets that act as a relational cache for data drawn from a data source. (A data source is typically a database, but can also be XML or any source for which data providers are available.) You can define data adapters that fill the dataset and propagate dataset changes back to the data source using stored procedures or SQL statements. For more information about this new data model, see Introduction to Data Access with ADO.NET.

Create Datasets with Strongly Typed Access

With ADO.NET you can create a dataset based on an XML schema, producing a typed class in which tables and columns become first-class objects in the object model. Typed access makes code easier to read, is less error prone, and is fully supported by IntelliSense in the Visual Studio Code Editor. For more information, see Creating ADO.NET Typed DataSets from Schemas.

Exchange Data As XML With Other Applications or Components

You can serialize data into and out of XML streams that can be exchanged with other XML-enabled applications or components by using either the built-in features of datasets or XML-specific methods. ADO.NET can easily create XML Web services with methods that return datasets as XML. For more information, see Building XML Web Services Using ASP.NET.

Use Wizards for Configuring Data Access

Wizards can be used to create and configure data adapters and connections.

The Data Form Wizard helps you create a form that is bound to a dataset, creating type-appropriate data-bound controls. The Data Form Wizard can create simple forms bound to a single table or master-detail forms bound to related tables, and can display data one record at a time or in a data-bound grid. For more information, see Data Form Wizard and Creating Data Adapters Using a Wizard.

Add Data Components to a Form or Component by Dragging

Server Explorer greatly simplifies data configuration by providing the capability to drag and drop tables from existing data stores to create data adapters and connection objects. Adapters, connections, commands, data views, and dataset objects are all available for drag and drop operations from the **Data** tab of the Toolbox. For more information, see Data Tab, Toolbox.

Create XML Schemas

You can also use the Visual Studio .NET XML Designer to visually design XML Schemas, which are files that validate XML data as well as describe the structure of datasets. For more information, see Introduction to XML Schemas.

Edit XML Documents

XML can be edited in a code editor with IntelliSense, or by using a graphical user interface. XML documents can also be validated against any XML schema. For more information, see XML Files.

Bind Any Control Property to Data

Whether you are working with Windows Forms or Web Forms pages, you can bind any property — or multiple properties — of any control to data. For step-by-step examples, see Walkthrough: Displaying Data in a Web Forms Page and Walkthrough: Displaying Data in a Windows Form Using a Parameterized Query.

Work with Data Programmatically

The data classes provided in the .NET Framework are used to create and manipulate data and XML in code. For more information, see Data Namespaces in Visual Studio and XML Namespaces.

Work with ADO

ActiveX Data Objects (ADO) is still supported using the COM interoperability support in the .NET Framework. For more information, see COM Component Compatibility.

See Also

Accessing Data | What's New in Visual Basic and Visual C# | What's New in Visual Studio .NET

What's New in Windows Forms and Controls

Windows Forms is the new object-oriented framework that implements the .NET Framework. Windows Forms, in combination with the Windows Forms controls, provide a robust architecture for Windows application development in Visual Basic, C#, and Managed Extensions for C++.

> **Visual Basic Note** If you are familiar with forms in Visual Basic 6.0, see Forms Task Changes in Visual Basic .NET for an introduction to the new concepts.

What's New in Visual Studio .NET 2003

A number of additions have been made to Windows Forms in this release.

FolderBrowserDialog Component
The Windows Forms **FolderBrowserDialog** component is a modal dialog box that is used for creating, browsing, and selecting folders. For details, see FolderBrowserDialog Component (Windows Forms).

Security in Windows Forms
A fundamental aspect of programming is making sure your applications are secure. In Windows Forms, a great deal of work has been done to make this as easy as possible for developers. Nonetheless, there are important issues to be aware of. For details, see Windows Forms Security.

Nonrectangular Windows Forms
It is easy to customize the shape of Windows Forms. All that is required for nonrectangular forms is an image file and a few property settings. For details, see Creating Nonrectangular Windows Forms.

Windows Forms Designer for Managed Extensions for C++
Visual C++ introduces a rapid application development solution for creating your Windows Forms applications in Managed Extensions for C++. For details, see Managed Extensions for C++ Windows Applications.

Smart Device Programmability
Visual Basic .NET now allows you to develop rich applications for resource-constrained devices. You can use the tools provided by Visual Studio .NET and the .NET Compact Framework, a subset of the .NET Framework, to create, build, and debug applications that run on smart devices. For details, see Smart Device Projects.

While almost the entire Visual Basic language is supported for device development, certain Visual Basic .NET features were changed or dropped to accommodate for the inherent differences in device applications. For more information, see Developing Device vs. Desktop Applications.

What's New in Visual Studio .NET 2002

The following are all a part of client application development with Windows Forms.

Windows Forms
Windows Forms is the new platform for Microsoft Windows application development, based on the .NET Framework. This framework provides a basis for creating powerful applications. For details, see System.Windows.Forms Namespace. This framework provides a clear, object-oriented, extensible set of classes that enable you to develop rich Windows applications. For details, see Windows Forms.

Windows Forms as the Presentation Layer of a Three-Tier Application
Windows Forms can act as a user-interface layer within a three-tier application. Using HTTP calls to a server that is connected to a database, Windows Forms are capable of acting as the front end to a powerful, data-enabled architecture that leverages the rich user interface of a client application with the advanced processing of an application server. For details, see Walkthrough: Creating a Distributed Application.

Creating a Client Application
With Windows Forms, you are able to create a rich client application using C# or Visual Basic. This application can access a wide variety of data sources and provide data-display and data-editing facilities using Windows Forms controls. For details, see Walkthrough: Creating a Simple Windows Form and Walkthrough: Simple Data Access in a Windows Form.

Windows Forms and Web Forms
When designing applications that involve a user interface, you have two choices: Windows Forms and Web Forms. Both have full design-time support within the development environment and can provide a rich user interface and advanced application functionality to solve business problems. For a guide to deciding which technology is appropriate for a given application, see Windows Forms and Web Forms Recommendations.

Visual Basic Code Behind a Windows Form
When you select a Windows Application from the New Project dialog box, a solution is created with a Visual Basic project in it. This project contains references and one class, which is called Form1.vb. For details on the code in this class, see Anatomy of the Visual Basic Code Behind Windows Forms.

Windows Forms Security Model

Windows Forms security is based on the security policies established within the .NET Framework. The System.Security namespace provides the underlying structure of the .NET Framework security system, including base classes for permissions. Windows Forms Security addresses some of the concerns developers need to keep in mind as they create Windows applications.

Tag and Name Properties

The **Tag** and **Name** properties have been added to the Control Class. The **Tag** property is a repository within memory to store data about the control. The **Name** property contains the name of the control. The **Name** property can be used at run time to evaluate the object by name rather than type and programmatic name. For more information about the **Tag** property, see Control.Tag Property. For more information about the **Name** property, see Control.Name Property.

See Also

What's New in Components and Component Authoring | What's New in Visual Basic and Visual C# | What's New in Visual Studio .NET

What's New in Projects

Projects are the application building blocks for Visual Basic and Visual C# code. A finished application can be created with multiple projects of different languages.

> **Note** Some project types are not available in the Standard Edition of Visual Basic and Visual C# .NET. For more information, see Visual Basic Standard Edition Features or Visual C# Standard Edition Features.

What's New in Visual Basic .NET 2003

Visual Basic .NET 2003 has added functionality for upgrading UserControl and WebClass projects from Visual Basic 6.0, plus support for multiple versions of the .NET Framework.

Upgrade Visual Basic 6.0 UserControl Projects
UserControl projects (also known as ActiveX Control projects) can be upgraded to a Windows Control Library. For more information, see UserControl Changes in Visual Basic .NET.

Upgrade Visual Basic 6.0 WebClass Projects
WebClass projects (also known as IIS Application projects) can be upgraded to an ASP.NET Web Application project. For more information, see WebClass Changes in Visual Basic .NET.

Target Multiple Versions of the .NET Framework
Applications can target a specific version of the .NET Framework. For more information, see Working With Multiple Versions of the .NET Framework.

What's New in Visual Basic .NET 2002

The following information highlights what has changed since version 6.0.

XML Web Services
XML Web services enable you to create server components that you can call from your applications using Internet-based protocols such as HTTP and XML. XML Web services can perform complex application logic and return data to a client. For more information, see Introduction to Programming XML Web Services in Managed Code.

Web Application Project
The Web Application template creates a new Web application by creating all necessary files on a specified Web server. For more information, see ASP.NET Web Application Template.

Windows Service Project
A Windows service application runs in the background of your system and requires no user interaction. It can perform functions such as performance monitoring. For more information, see Windows Service Template.

Console Application Project
The Console Application template creates an application that contains the necessary items needed to create applications that run from the command line. For more information, see Console Application Template.

Web Control Library Project
The Web Control Library template creates an application to author and publish Web controls. For more information, see Web Control Library Template.

Database Projects
The Database Project template is an application wizard that assists you in managing database elements such as tables and stored procedures. For more information, see Database Projects.

> **Note** This project type is not available in the Standard Edition of Visual Basic and Visual C# .NET. For more information, see Visual Basic Standard Edition Features or Visual C# Standard Edition Features.

Project Extensibility
Visual Studio .NET includes a programmable, project-neutral object model that provides access to the underlying components and events of the integrated development environment (IDE). For more information, see What's New in Extensibility and Automation.

Windows Forms and Controls
The new object-oriented Windows application structure is Windows Forms. Combined with Windows Forms Controls, Visual Studio .NET provides a design environment to quickly create standard Windows applications. For more information see What's New in Windows Forms and Controls.

Projects and Deployment
Visual Studio .NET includes deployment projects to assist in rapid deployment of your applications. The traditional deployment model has changed since Visual Studio 6.0. For more information see What's New in Deployment.

Upgrading from Visual Basic.6.0
The differences between Visual Basic 6.0 and Visual Basic .NET make it necessary to make changes when upgrading

applications. For more information, see Preparing a Visual Basic 6.0 Application for Upgrading.

Option Strict (Visual Basic Only)

The **Option Strict** statement disallows any data-type conversions in which data loss would occur and disallows any conversion between numeric types and strings. For more information, see Type Checking.

Creating New Web and Windows Projects in Existing Directories

Visual Studio .NET allows you to create new Web and Windows projects in existing folders with the New Project in Existing Folder template. For more information, see New Project in Existing Folder Template.

See Also

What's New in Visual Basic and Visual C# | What's New in Visual Studio .NET | What's New in the Visual Basic Language | Visual C# Language | Managing Projects | Programming with Office

What's New in Components and Component Authoring

You can easily create your own components in code or assemble them from the sophisticated components in the .NET Framework classes, using either Visual Basic or C#. You can use non-visual components and associated features to easily incorporate resources such as message queues, event logs, and performance counters into your applications.

The .NET Framework classes greatly expand the component programming capabilities of Visual Studio. Instead of a limited set of base classes, you have a large library of sophisticated components that you can use to assemble applications, or to derive high-quality components of your own.

> **Visual Basic Note** For an overview of how the Visual Basic language differs from previous versions, see Language Changes in Visual Basic.

What's New in Visual Basic .NET 2002

Visual Basic .NET 2003 includes the following features, which were introduced in Visual Basic .NET 2002.

Enhanced RAD Support for Creating Your Own Components
The new RAD (rapid application development) support makes authoring controls and non-visual components as easy as creating forms. Use inheritance to start from an existing component, and add other components to build in the functionality you need. For details, see Component Authoring.

Enhanced RAD Support for Creating Your Own Windows Controls
When you select the Windows Control Library project template, Visual Studio creates a **UserControl** class that inherits from **System.Windows.Forms.UserControl**. If you add a Custom Control item to your project, you will get a **CustomControl** class that inherits from **System.Windows.Forms.Control**. For details, see Control Authoring for Windows Forms.

Create Your Own Web Controls
Visual Studio provides a Web Control Library project template, which automatically creates a **WebControl** class that inherits from **System.Web.UI.WebControls.WebControl**. For details, see Developing ASP.NET Server Controls in the *.NET Framework Developer's Guide*.

.NET Framework Classes for Component Authoring
Inherit from the **Component** class or implement the **IComponent** interface to create your own component. For details, see Component Class, IComponent Interface and Component Model Namespaces in Visual Studio.

Messaging Components
You can create component instances that expose Microsoft Message Queuing functionality, allowing you to interact with message queues on local and remote computers. For details, see Creating Messaging Components.

System Monitoring Components
You can create instances of components that allow you to monitor and interact with a variety of system resources such as performance counters, event logs, services, and processes. For details, see Creating System Monitoring Components.

Timed Components
You can create server-based components that raise events on a recurring schedule or interval. For details, see Creating File System and Timer Components.

File System and Directory Components
You can create component instances that interact with file and directory structures, allow you to watch for and react to changes to a directory or file, or retrieve information about a system's directory structure. For details, see Creating File System and Timer Components.

Server Explorer
You can use the new Server Explorer tool to examine the resources that are available on servers to which you have network access, to verify the creation of resources you create, and to easily create components in your applications that reference existing resources. For details, see Introduction to Server Explorer

Common Language Runtime
The common language runtime is designed to simplify the creation and use of components. An important aspect of this simplification is enhanced interoperability between components written using different programming languages. For details, see Common Language Runtime.

Deploy Shared Components
You can create packages for sharing components that are used by multiple applications, eliminating versioning problems. For details, see Merge Module Projects.

See Also

What's New in Visual Basic and Visual C# | What's New in Visual Studio .NET | What's New in the Visual Basic Language | What's New in Windows Forms and Controls | What's New in Web Development | Component Authoring Changes in Visual Basic .NET

What's New in Debugging

Visual Studio .NET provides a single integrated debugger for all Visual Studio languages, including Visual Basic and C#. A new, unified interface combines features of the Visual C++ and Visual Basic 6.0 debuggers, as well as many new features.

> **Visual Basic Note** If you are familiar with the debugger in Visual Basic 6.0, see Debugging Changes in Visual Basic .NET.

What's New in Visual Basic .NET 2002

Visual Basic .NET 2003 includes the following features, which were introduced in Visual Basic .NET 2002.

Cross-Language Debugging
Using the integrated debugger, you can debug projects that are part of the same solution but are written in different languages. For example, you can debug a solution that contains a project consisting of a Visual Basic or Visual C# user interface application and a Visual C++ server application, and you can step back and forth between these projects, for example from Managed Extensions for C++ to Visual Basic and back to Managed Extensions for C++. For more information, see Mixed-Mode Debugging.

Attaching to a Running Program
You can attach the debugger to a program that is already running and debug the program. Attaching to a running program works the same way whether the program is running on a host machine or a remote machine. The program does not need to be launched in the Visual Studio .NET integrated development environment (IDE). For more information, see Attaching to a Running Program or Multiple Programs.

Remote Debugging
You can attach to and debug a process that is running on a different computer from the one on which you are running Visual Studio. For example, if you are debugging a Windows Forms client application and an XML Web service it uses, you can run Visual Studio on the client and then attach to the server that is providing the XML Web service. You typically perform this kind of debugging for Web Forms projects or ASP.NET Web Service projects. For more information, see Attaching to a Running Program or Multiple Programs.

Debugging Multithreaded Applications
New for Visual Basic users is the ability to write and debug multithreaded applications. The Threads window can be used to view the threads that are running and to switch context. For more information, see Using the Threads Window.

Debugging Multiple Programs
You can debug multiple programs by attaching to running programs or by launching multiple programs from the Visual Studio IDE. For more information, see Debugging Multiple Programs.

Debugging ASP.NET Web Applications
Configuring debugging for ASP.NET Web applications has been significantly improved. In most cases you can simply choose Start from the Debug menu and the debugger will automatically attach to the ASP.NET worker process for debugging Web Forms, even if the ASP.NET worker process is running on a different server. Security configuration has been made easier by the addition of a debugger users group. For more information, see Debugging Script and Web Applications.

.NET Framework Classes for Debugging and Code Tracing
You can include instrumentation code in your Microsoft .NET application. Several .NET Framework classes are available for use both in debugging and in instrumenting your code. For more information, see Tracing and Instrumenting Applications in Visual Basic and Visual C#, Debug class, and Trace class, and scan the table of contents in the System.Diagnostics Namespace in the .NET Framework documentation.

Replacements for Visual Basic's Debug.Print
In Visual Basic .NET the **Debug.Print** method has been replaced by four new methods: Debug.Write method, Debug.WriteIf method, Debug.WriteLine method, and Debug.WriteLineIf method. The method most resembling **Debug.Print** is **Debug.Write**. To use these or any other methods of the **Debug** class, Visual Basic and Visual C# users should remember to include at the appropriate place in their code the statement `Imports System.Diagnostics` or `Using System.Diagnostics`, respectively. For more information, see Programming Element Support Changes Summary.

See Also

What's New in Visual Studio .NET | Debugging Managed Code | What's New in Visual Basic and Visual C#

What's New in Extensibility and Automation

Visual Studio .NET includes a programmable, project-neutral object model that provides access to the underlying components and events of the integrated development environment (IDE). This model has components that represent solutions, projects, tool windows, code editors, debuggers, code objects, documents, and events. You can extend the functionality of the IDE, automate repetitive tasks, and integrate other applications into the IDE. The object model can be used through four programming methods: macros, add-ins, wizards, and the Visual Studio .NET Integration Program (VSIP).

What's New in Visual Basic .NET 2002

Visual Basic .NET 2003 includes the following features, which were introduced in Visual Basic .NET 2002.

The Object Model for Extensibility of Visual Basic and Visual C# Projects

While Visual Studio .NET provides a general extensibility model, there is also an object model that is specific to Visual Basic and Visual C# projects. This language-specific model, contained in the **VSLangProj** namespace, allows you to programmatically manipulate project attributes found only in Visual Basic and Visual C# projects. These items include:

- VSProject Object Provides access to project properties.
- VSProjectItem Object Provides access to project item properties.
- Reference Object Provides access to a reference in a project.
- Imports Object Provides access to the project-level **Imports** statements for a Visual Basic project.
- BuildManager Object Provides support for implementing custom tools.
- Properties Property (Visual Basic and Visual C# Project Object) Contains the Visual Basic and Visual C# project properties.
- Properties Property (Visual Basic and Visual C# Configuration Object) Contains the Visual Basic and Visual C# configuration properties.
- Properties Property (Visual Basic and C# ProjectItem Object) Contains the properties of the individual files and folders contained in the project.

For more information on the general extensibility model, see Creating Add-Ins and Wizards. For more information on the Visual Basic and Visual C# extensibility model, see Introduction to Project Extensibility in Visual Basic and Visual C#.

Record Macros

You can automate repetitive tasks using the macro-recording features. The macros are stored as Visual Basic .NET procedures that may be edited using the Visual Studio .NET Macros IDE. This allows developers to edit the macro and even use macros to learn about the extensibility object model. For more information, see Automating Repetitive Actions by Using Macros.

Write Macros

For more complex situations, you can write macros using Visual Basic .NET. You can use all the features of Visual Basic .NET in macros, including inheritance, structured exception handling, and the .NET Framework. For more information, see Automating Repetitive Actions by Using Macros. You have programmatic access to macro projects, too. For more information, see The Macro Project Object Model.

Organize Macros

The Macro Explorer in the Visual Studio .NET Macros IDE provides a hierarchical view of the available macros, which are organized into modules contained in macro projects. From this window, you can organize, edit, and run macros. For more information, see Macro Explorer Window.

Manipulate Code without Parsing

The project-neutral extensibility model, called the general extensibility model, provides access to the individual code files of your project. You can navigate to and examine a specific class method or class variable without having to parse the code. Go straight to the code element you want to modify using the CodeModel Object, get its location in the Document Object, and examine and modify your code. For more information about using these models with Visual Basic and Visual C#, see Working with the CodeModel Object of a Visual Basic Application, and Working with the CodeModel Object of a Visual C# Application.

Add-ins

Add-ins are extensions to the IDE. They are compiled components that manipulate the environment and automate tasks. Add-ins can be invoked in a variety of ways, including the Add-in Manager, toolbar commands or buttons, the devenv command line, and events such as IDE startup. Add-in projects implement the IDTExtensibility2 interface. The power of add-ins rests in the fact that:

- You have access to all the features of .NET to modify the IDE.
- You can write add-ins using any of the programming languages included with Visual Studio.
- Add-ins are compiled, and therefore your code is not exposed as it is in macros.

The Add-In Wizard is provided to create skeleton add-in projects. For more information, see Creating an Add-In.

Wizards

Wizards are components that implement the IDTWizard interface. These applications lead users step-by-step to complete a task. For more information, see Creating a Wizard.

Visual Studio .NET Integration Program (VSIP)

Even more complex extensions to the IDE are possible through the VSIP. This program gives you access to additional objects and interfaces that let you create your own project types and designers. For more information, see The Spectrum of Visual Studio .NET Automation.

See Also

What's New in Visual Basic and Visual C# | What's New in the Visual Basic Language | What's New in Visual Studio .NET | Programming with Office

What's New in International Applications

Visual Studio .NET was designed from the start to make developing for an international audience easy by taking advantage of services built into the .NET Framework. In Visual Studio, there are two parts to creating an international-ready application. *Globalization* is the process of designing applications that can adapt to different cultures. *Localization* is the process of translating resources for a specific culture.

What's New in Visual Basic .NET 2002

Visual Basic .NET 2003 includes the following features, which were introduced in Visual Basic .NET 2002.

Cultures

A *culture* is composed of settings related to the user's language, environment, and cultural conventions. In applications that use the common language runtime and the .NET framework, culture takes the place of the National Language Support-based locale, which uses a system of LCID (Locale ID) codes. An LCID property of the CultureInfo class provides interoperability and eases integration with NLS-based software. There are two culture settings in Visual Studio: the UI culture setting determines which resources will be loaded, and the culture setting determines formatting of values such as dates, numbers, currency, and so on. For details, see Setting the Culture and UI Culture for Windows Forms Globalization or Setting the Culture and UI Culture for Web Forms Globalization.

XML Resource Files

Visual Studio resource files contain localized resources such as strings and images that are appropriate for particular cultures. For localization, you would typically create fallback resources for the culture where the application was originally developed and an additional resource file for each culture you wish to localize the application for. For details, see Introduction to International Applications in Visual Basic and Visual C#.

Main and Satellite Assemblies

When you build a project, the resource files are compiled and then embedded in satellite assemblies, or assemblies which contain only the localized resources. The fallback resources are built into the main assembly, which also contains the application code. For details, see Introduction to International Applications in Visual Basic and Visual C#.

Fallback System of Resources

Because the main assembly always contains the fallback resources, if a resource is not found in the localized satellite assembly, Visual Studio will attempt to load it in a hierarchical manner, eventually falling back to the resources in the main assembly. For details, see Hierarchical Organization of Resources for Localization.

Localization Project Support

Visual Studio provides project system support for editing and building localizable applications. The project system can create resource files for you, or you can manually create and edit them. For details, see Walkthrough: Localizing Windows Forms and Walkthrough: Localizing Web Forms Pages.

Full Unicode Support

Visual Studio resources are fully Unicode-based. The .NET Framework handles platform differences for you. It is no longer necessary to write separate code for different platforms such as Windows NT and Windows 98. For details, see Encoding and Windows Forms Globalization.

Support for International Web Applications

Visual Studio supports the development of globalized and localizable Web applications by providing resource management in Web applications, encoding Web pages in UTF-8, and allowing the layout of Web pages with relative sizing. For details, see Walkthrough: Localizing Web Forms Pages.

Deployment Localization

You can create localized setup projects in many languages. For details, see Deployment and Localization.

See Also

Planning World-Ready Applications | International Changes in Visual Basic .NET

What's New in Upgrading

Visual Basic .NET 2003 includes several new features for upgrading Visual Basic 6.0 applications and code to Visual Basic .NET.

What's New in Visual Basic .NET 2003

Support for WebClass Upgrade
You can now upgrade Visual Basic 6.0 WebClass projects to ASP.NET Web Applications. For more information, see Upgrading WebClass Projects to Visual Basic .NET.

Support for UserControl Upgrade
You can now upgrade Visual Basic 6.0 UserControl projects to Visual Basic .NET Windows Control Libraries. For more information, see UserControl Changes in Visual Basic .NET.

Reuse Your Visual Basic 6.0 Code
There is a new tool that allows you to paste in blocks of Visual Basic 6.0 code and convert it to Visual Basic .NET. For more information, see Upgrading Visual Basic 6.0 Code with the Upgrade Visual Basic 6 Code Dialog Box.

Convert Visual J++ 6.0 and Java-Language Applications to Visual C# .NET
The Java Language Conversion Assistant allows you to convert Java-language and Visual J++ 6.0 applications to Visual C# .NET. For more information, see Java Language Conversion Assistant.

See Also

What's New in Visual Basic and Visual C#

Visual Basic Standard Edition Features

This topic outlines and describes the features that are included in the Visual Basic .NET Standard Edition.

Platform Prerequisites

Before you can install Visual Basic .NET Standard Edition, the setup sequence automatically detects your system configuration and installs baseline components required by the product.

Baseline Components

Depending on your system configuration, one or more of the following baseline components is automatically installed on your computer:

- Microsoft Data Access Components 2.7
- Windows 2000 Service Pack 1

Common Language Runtime and Framework

The .NET Framework is a multi-language environment for building, deploying, and running applications and XML Web services. It consists of three parts:

- Common language runtime
- Unified programming classes
- ASP.NET

Languages and Libraries

Feature	Description
Visual Basic Compiler Options	Compiles Visual Basic source files within the Visual Studio integrated development environment (IDE) or from the command line using Vbc.exe. For more information on the Visual Basic .NET language and run-time library, see Visual Basic Language and Run-Time Reference.
C# Compiler Options	Compiles C# source files from the command line using Csc.exe. For more information, see Building from the Command Line.
C++ Compiler, Linker, and C++ Language	Compiles C++ source files from the command line using Cl.exe. For more information, see Building C++ Projects in Visual Studio.

Project and Editing Environments

Feature	Description
Visual Studio macros and programmability	The Visual Studio Macros development environment (separate from the Visual Studio environment) is used solely for creating, manipulating, editing, and running macros. For more information, see Macros Development Environment.
Visual Studio IDE add-ins	Extend the Visual Studio IDE by adding functionality that is not in the core product itself through COM add-ins. For more information, see Creating Add-Ins and Wizards.

Debugging

Feature	Description

Debugging Managed Code	Debugs Visual Basic and Visual C# and Visual Basic: • Windows applications • Class libraries • Windows control library • Web applications • Web Services • Console applications • Windows services Debugs Managed C/C++: • Applications • Class libraries • Web Services
Runtime Debugger (Cordbg.exe)	Helps tools vendors and application developers find and fix bugs in programs that target the .NET Framework common language runtime.

For more information, see Debug and Release Configurations.

Application Deployment

Feature	Description
Creating or Adding a Setup Project	Creates Windows Installer (.msi) files, which are used to distribute your application for installation on another computer or Web server.

Project Templates

The best way to create new projects is by using Visual Basic and Visual C# project templates, or application wizards. Project templates work in conjunction with application frameworks and libraries to create starter programs for you.

In the Standard Edition of Visual Basic, you do not have access to the empty project types and the project types that compile into DLLs. The following table lists the project types that are included.

Feature	Description
Introduction to Windows Applications	A project with the proper references and UI items to create a standard Windows-based application.
Introduction to ASP.NET Web Applications in Visual Studio	A project with the proper references and UI items to create an ASP.NET Web application.
Introduction to Programming XML Web Services in Managed Code	A project with the proper references and items to create a programmable XML Web service.
Console Application Template	A project with the proper references and items needed to create a project without a user interface that can be run from the command line.

For more information, see Visual Basic and Visual C# Projects. For a list of available project items, see File Types and File Extensions in Visual Basic and Visual C#.

Other Tools

The following designers are included in the Standard Edition.

Feature	Description
Web Forms Designer	Provides a set of visual tools for working with ASP.NET Web Forms pages and controls.
Windows Forms Designer	Provides a set of visual tools for working with Windows Forms and controls.
Component Designer	Provides a set of visual tools for working with nonvisual components.
XML Designer	Provides a set of visual tools for working with XML schemas, ADO.NET datasets, and XML documents.

Online Help and Documentation

Feature	Description

Visual Studio Combined Help Collection (VSCC)	Contains the standard Visual Studio documentation, as well as documentation for any installed third-party packages or add-ins.

Server Technologies

Feature	Description
Microsoft Data Engine (MSDE)	A fully SQL Server–compatible data engine used for building desktop and shared solutions that provide the easiest migration path to SQL Server 7.0. For more information, see http://www.microsoft.com/sql/msde/productinfo/features.asp.
Server Explorer	Provides access to data connections to MSDE or Access databases. For more information, see Introduction to Server Explorer.

Software Development Kits

Feature	Description
.NET Framework SDK	A wealth of resources — including DLLs, tools, and samples — that enable developers to build efficient, powerful, and scalable applications and services that take advantage of the new .NET Framework technology.

See Also

Visual Database Tools Editions | Visual Studio .NET Editions | Visual C# Standard Edition Features | Visual C++ Standard Edition Features

Visual C# .NET Standard Edition

Visual C# .NET Standard Edition is a member of the Visual Studio .NET family of development products. A Standard Edition is available for each programming language that targets the .NET Framework and is part of the Visual Studio suite.

In This Section

Visual C# Standard Edition Features
Take a look at Visual C# .NET Standard Edition Features.

Related Sections

Visual Studio .NET Editions
Provides more information on the various Visual Studio .NET products.
Visual Basic Standard Edition Features
Lists features included in Visual Basic .NET Standard Edition.
Visual C++ Standard Edition Features
Lists features included in Visual C++ .NET Standard Edition.
Visual Studio .NET Hardware Requirements
Make sure the Visual C# .NET Standard Edition is compatible with your current system configuration.

Visual C# Standard Edition Features

This topic outlines and describes the features that are included in the Visual C# .NET Standard Edition.

Platform Prerequisites

When you install Visual C# .NET Standard Edition, the setup sequence automatically detects your system configuration and then prompts you to install baseline components required by the product. This is Step 1 (Visual Studio .NET Prerequisites) of the four Visual Studio installation processes.

Baseline Components

Depending on your system configuration, one or more of the following baseline components are automatically installed on your machine:

- Windows 2000 Service Pack 3
- Microsoft Windows Installer 2.0
- Microsoft FrontPage 2000 Web Extensions Client
- Setup Runtime Files
- Microsoft Internet Explorer 6
- Microsoft Data Access Components 2.7
- Microsoft .NET Framework 1.1

Common Language Runtime and .NET Framework

The .NET Framework is a multilanguage environment for building, deploying, and running XML Web services and Web applications. It consists of three main parts:

- Common language runtime
- Unified programming classes
- ASP.NET

Languages and Libraries

The following language and library resources are available.

- C# compiler and runtime

  Compiles C# source files in the Visual Studio integrated development environment (IDE) or at the command line using CSC.EXE.

- C# environment language services, as follows:
  - Supports COM/platform code integration.
  - Includes garbage collection and type safety.
  - Provides security through intrinsic code trust mechanisms.
  - Supports XML Web-based components and extensible metadata concepts.
  - Provides interoperability with other languages, across platforms, and with legacy data.
  - Supports versioning to provide ease of administration and deployment.
  - Includes CodeSense.
- C++ compiler

  Compiles C++ source files from the command line using CL.EXE.

- Visual Basic compiler and runtime

  Compiles Visual Basic source from the command line using VBC.EXE.

Project and Editing Environments

The following project and editing environments are available.

- Visual C# Managed Project System

When compiling to managed code, the compiler translates your source code into Microsoft intermediate language (MSIL), a CPU-independent set of instructions that can be efficiently converted to native code.

- Visual Studio IDE add-ins

  Extend the Visual Studio IDE by adding functionality that is not in the core product itself through COM add-ins.

Debugging

The following topics provide debugging information for specific programming environments.

- Debugging Managed Code

  Provides information on managed debugging engines, which debug Managed Extensions for C++ applications, Windows applications, class libraries, and XML Web services.

- Debugging Visual C#

  Discusses debugging Visual C# Windows applications, class libraries, Windows control library, Web applications, XML Web services, console applications, and Windows services.

- Runtime Debugger

  Documents the .NET Framework SDK Debugger, which helps tools vendors and application developers find and fix bugs in programs that target the .NET Framework common language runtime.

For additional information, see Debug and Release Configurations.

Application Deployment

- Creating or Adding a Setup Project

  Discusses how to create Windows Installer (.msi) files, which are used to distribute your application for installation on another computer.

Application Wizards

The best way to create new projects is by using Visual C# application wizards. Visual C# application wizards work in conjunction with application frameworks and libraries to create starter programs for you that are modeled from project templates.

Application Wizard	Description
Windows Application	Creates a Visual C# application with a Windows user interface.
Class Library	Creates a project for creating classes that can be used in other a pplications.
Console Application	Creates a Visual C# application with a command-line interface.
ASP.NET Web Application	Creates a Visual C# application with a Web user interface.
ASP.NET Web Service	Creates an XML Web service with Visual C# that other applicatio ns can access.
ASP.NET Mobile Web Application	Creates an application viewable on PDAs, cellphones, and other mobile devices.
Empty Project	Creates an empty project for creating a local application.
Empty Web Project	Creates an empty project for creating a Web application.

Code Wizards and Items

After you create the framework of your project, you can use the Visual C# code wizards to help you add items such as classes, members, or functions into your programs. You can also add various files and items to your projects.

Solution Explorer Code Wizards

In Solution Explorer, use the **Add New Item** command. This opens the Add New Item dialog box, which is used to add an item to your project. When you select an item from the Categories list, the appropriate files and references are added to your project.

Note The **Add New Item Crystal Report** option is not included in the Visual C# Standard Edition.

Add New Item	Description

Windows Form	Adds a form for Windows applications to your project.
Class	An empty class declaration.
Component Class	A class for creating components using the visual designer.
Add User Control	Adds a class for creating a Windows Form control using the visual designer.
Data Form Wizard	Adds a .cs file for creating an XML schema with DataSet classes.
Data Set	Adds an .xsd file for creating an XML schema with DataSet classes.
XML File	Adds a blank .xml file.
XML Schema	Adds a file for creating a schema for XML documents.
Code File	Adds a blank .cs code file.
Custom Control	Adds a class for creating a user-drawn Windows Form control.
HTML Page	Adds an HTML page that can include client-side code.
Inherited Form	Adds a new form based on an existing Windows Form.
Web Custom Control	Adds a class for creating an ASP.NET server control.
Inherited User Control	Adds a new control based on an existing Windows Form control.
Windows Service	Adds a class for creating Windows services.
Text File	Adds a blank text file.
Frame Set	An HTML file that hosts multiple HTML pages.
XSLT File	Adds a file used to transform XML documents.
Style Sheet	Adds a cascading style sheet used for rich HTML style definitions.
Installer Class	Adds a class to be invoked at setup time.
Bitmap File	Adds a Win32 bitmap file.
Cursor File	Adds a Win32 cursor file.
Icon File	Adds a Win32 icon file.
Assembly Resource File	Adds a .NET resource file.
JScript File	Adds a script file containing JScript code.
VBScript File	Adds a script file containing VBScript code.
Windows Script Host	Adds a .WSF file containing script that is run as a Windows program.

Class View Code Wizards

In Class View, you can invoke the code wizards that have the new HTML interface. These wizards help you specify tokens when declaring programming elements such as classes, methods, and properties.

Code wizard	Description
C# Class Wizard	Adds an empty Visual C# class declaration to your project.
C# Method Wizard	Adds a method to your Visual C# class.
C# Property Wizard	Adds a property to your Visual C# class.
C# Field Wizard	Adds a field to your Visual C# class.
C# Indexer Wizard	Adds an indexer to your Visual C# class.
C# Interface Method Wizard	Adds a method to a C# interface.
C# Interface Property Wizard	Adds a property to a C# interface.
C# Interface Indexer Wizard	Adds an indexer to a C# interface.

For more information, see C# Code Wizards.

Other Tools

The following designers are available.

- Web Forms Designer

  Web Forms is an ASP.NET technology that you use to create programmable Web pages.

- Windows Forms Designer

Windows Forms is the new platform for Microsoft Windows application development, based on the .NET Framework. The Windows Forms Designer provides a rapid development solution for creating Windows applications.

- XML Designer

  The XML Designer provides a set of visual tools for working with XML schemas, ADO.NET datasets, and XML documents.

Online Help and Documentation

The Visual Studio Combined Help Collection (VSCC) contains the standard Visual Studio documentation, as well as documentation for any installed third-party packages or add-ins.

Server Technologies

Microsoft Data Engine (MSDE) is a fully SQL Server-compatible data engine used for building desktop and shared solutions that provide the easiest migration path to SQL Server 2000. For more information, see SQL Server 2000 Desktop Engine (MSDE 2000).

Software Development Kits

The .NET Framework SDK offers a wealth of resources — including DLLs, tools, and samples — that enable developers to build efficient, powerful, and scalable Web-based applications and services that take advantage of the new .NET Framework technology. For details, see .NET Framework SDK.

See Also

Visual C# .NET Standard Edition

Upgrading Applications

The upgrading section discusses how you can upgrade programs created with older versions of Visual Basic and Visual InterDev to Visual Basic .NET.

In This Section

Upgrading from Visual Basic 6.0
Covers how to convert applications to Visual Basic .NET.
Upgrading Visual InterDev 6.0 Applications to Visual Studio .NET
Describes how to upgrade a Web application built in Visual InterDev version 6.

Related Sections

Migrating to Web Forms
Describes how to upgrade ASP pages and HTML pages to Web Forms pages.
Introduction to Solutions, Projects, and Items
Describes the core concepts underlying application development in the integrated development environment (IDE). This topic provides background information about managing upgraded projects and performing project-related tasks.
Getting Assistance
Provides information about installation, editions, product support, and customization of the integrated development environment (IDE).

Upgrading from Visual Basic 6.0

Visual Basic .NET represents a departure from previous versions of Visual Basic in several ways. With a new development environment, an updated programming language, and a new forms package, moving from Visual Basic 6.0 to Visual Basic .NET may at first appear to be a daunting task.

In reality, much of what made Visual Basic distinctive has been retained and improved to make development easier than ever before. Although there are many changes, enhancements to the development environment and Help system ease the task of upgrading your code to conform with Visual Basic .NET and the .NET Platform. Extensive documentation is provided to help you make the transition, mapping your knowledge of Visual Basic 6.0 to Visual Basic .NET equivalents.

As you move forward to Visual Basic .NET, you will likely want to move some of your Visual Basic 6.0 projects forward, as well. Although there are differences between the two versions, tools are provided to ensure that your projects will be upgraded successfully.

The following documentation will help you to upgrade your applications and quickly get up to speed on the changes to Visual Basic.

In This Section

Introduction to Visual Basic .NET for Visual Basic Veterans
  Describes differences between Visual Basic 6.0 and Visual Basic .NET.
Upgrading Applications Created in Previous Versions of Visual Basic
  Describes upgrading your applications to Visual Basic .NET.
Working with Both Visual Basic 6.0 and Visual Basic .NET
  Describes compatibility between the two versions of Visual Basic.
Walkthrough: Upgrading a Visual Basic 6.0 Application to Visual Basic .NET
  Compares a Visual Basic 6.0 sample application and a Visual Basic .NET equivalent.

Related Sections

What's New in Visual Basic and Visual C#
  Describes new features in Visual Basic .NET.
Language Changes in Visual Basic
  Describes changes to the Visual Basic language.

Introduction to Visual Basic .NET for Visual Basic Veterans

As a veteran Visual Basic developer, you have undoubtedly invested a lot of time and effort in learning the intricacies of Visual Basic. At first glance, it may appear to you that Visual Basic .NET is so radically different from what you know that you will have to learn it all over again. In reality, while many things have changed, the overall experience of developing in Visual Basic remains essentially the same. Your knowledge of Visual Basic will help you immediately become productive in Visual Basic .NET.

There are, however, a number of changes to Visual Basic that will require you to adjust your mindset and set aside some old habits. Some of the rules for the language, tasks such as debugging and deployment, and even some terminology have changed — often in subtle ways. Ultimately, the changes make Visual Basic easier and more powerful than ever before.

The following topics will help you identify the changes in Visual Basic .NET from the perspective of a Visual Basic 6.0 veteran.

In This Section

Integrated Development Environment Changes in Visual Basic .NET
Describes changes to the development environment.
Project Changes in Visual Basic .NET
Describes changes to the project system.
Web Changes in Visual Basic .NET
Describes changes to Web-related development.
WebClass Changes in Visual Basic .NET
Describes changes to WebClass projects (also known as IIS Application projects).
Data Changes in Visual Basic .NET
Describes changes to data access and data tools.
Component Authoring Changes in Visual Basic .NET
Describes changes to component authoring.
UserControl Changes in Visual Basic .NET
Describes changes to UserControl projects (also known as ActiveX Control projects) and provides a link to information about the **PropertyBag** object.
Forms Changes in Visual Basic .NET
Describes changes to forms.
Debugging Changes in Visual Basic .NET
Describes changes to debugging.
Setup and Deployment Changes in Visual Basic .NET
Describes changes to Setup and deployment.
International Changes in Visual Basic .NET
Describes changes to localization.
Windows API Changes in Visual Basic .NET
Describes differences in accessing the Windows API.
Registry Access Changes in Visual Basic .NET
Describes changes in setting and retrieving registry settings.
Constant Changes in Visual Basic .NET
Provides a list of constants that have changed in Visual Basic .NET.
Namespace Changes in Visual Basic .NET
Describes differences and similarities between Visual Basic 6.0 object libraries and Visual Basic .NET namespaces.
Run-Time Changes in Visual Basic .NET
Discusses differences between Visual Basic 6.0 run-time files and the .NET Framework common language runtime.

Related Sections

Language Changes in Visual Basic
Describes changes to the Visual Basic language.
Control Changes in Visual Basic .NET
Describes changes to controls.
Upgrading from Visual Basic 6.0
Covers how to convert applications to Visual Basic .NET.
What's New in Visual Basic and Visual C#
Covers new features for different areas, including Web development, data access, component creation, and the Visual Basic and Visual C# languages.

Integrated Development Environment Changes in Visual Basic .NET

Visual Basic .NET is fully integrated into the Visual Studio integrated development environment (IDE), which differs in several ways from the Visual Basic 6.0 IDE. The following sections describe the changes:

Window and Layout Changes

Menu Changes

Keyboard Mapping Changes

Window and Layout Changes

The standard arrangement of windows is different in Visual Basic .NET than it was in Visual Basic 6.0. If you prefer the Visual Basic 6.0 arrangement, you can set it by choosing **Visual Basic 6** from the **Window Layout** drop-down list on the **My Profile** tab of the Visual Studio **Start Page**.

In Visual Basic 6.0, the IDE was set to an MDI layout by default; you could optionally set it to SDI layout in the **Tools**, **Options** dialog box. In Visual Basic .NET, the default layout for the IDE is the new Tabbed documents layout; you can optionally set it to MDI layout in the **Tools**, **Options** dialog box.

In Visual Basic 6.0, you could control the docking behavior for some tool windows in the **Tools**, **Options** dialog box. In Visual Basic .NET, all windows are dockable by default; you can control their behavior via the **Dockable** command on the **Windows** menu.

There are also some differences in the available windows between Visual Basic 6.0 and Visual Basic .NET.

> **Note** In Visual Basic .NET, some of the debugging windows are only available in run mode; when you switch to design mode, the windows are hidden.

The following table shows some Visual Basic 6.0 windows and their Visual Basic .NET equivalents.

Visual Basic 6.0	**Visual Basic .NET**
Color Palette	No equivalent
Data View	No direct equivalent; replaced by **Server Explorer**
Form Layout	No equivalent
Immediate	**Command Window**
Project Explorer	**Solution Explorer**
Watches	**Watch 1**, **Watch 2**, **Watch 3**, **Watch 4**

Menu Changes

Menu commands in Visual Studio .NET are standardized for all languages; in some cases, Visual Basic 6.0 menu commands are replaced by different commands in Visual Studio .NET.

In Visual Basic 6.0, all menu commands were visible at all times but were grayed-out when out of context; in Visual Basic .NET, the set of available commands differs according to context.

The following table lists some common Visual Basic 6.0 commands and their Visual Basic .NET equivalents.

Visual Basic 6.0	**Visual Basic .NET**
Add-Ins, **Add-In Manager**	**Tools**, **Add-in Manager**
Debug, **Add Watch**	**Add Watch** (context menu only)
Debug, **Edit Watch**	No equivalent. Watches are edited in the Watch window.
Debug, **Run To Cursor**	**Run To Cursor** (context menu only)
Debug, **Set Next Statement**	**Set Next Statement** (context menu only)
Debug, **Show Next Statement**	**Show Next Statement** (context menu only)
Debug, **Toggle Breakpoint**	**Disable Breakpoint** or **Breakpoint** (context menu only)
Edit, **Complete Word**	**Edit**, **Intellisense**, **Complete Word**

Edit, **Find**	**Edit**, **Find and Replace**
Edit, **Find Next**	**Edit**, **Find and Replace**
Edit, **Indent**	**Edit**, **Advanced**, **Increase Line Indent**
Edit, **Insert File**	**Edit**, **Insert File As Text**
Edit, **List Constants**	**Edit**, **Intellisense**, **List Members**
Edit, **List Properties/Methods**	**Edit**, **Intellisense**, **List Members**
Edit, **Outdent**	**Edit**, **Advanced**, **Decrease Line Indent**
Edit, **Parameter Info**	**Edit**, **Intellisense**, **Parameter Info**
Edit, **Quick Info**	**Edit**, **Intellisense**, **Quick Info**
Edit, **Replace**	**Edit**, **Find and Replace**
File, **Make *ProjectName***	**Build**, **Build *ProjectName***
File, **Make Project Group**	**Build**, **Build Solution**
File, **Print Setup**	**File**, **Page Setup**
File, **Recent files list**	**File**, **Recent Projects**
File, **Remove Project**	**Edit**, **Remove**
File, **Save Project Group**	**File**, **Save *SolutionName*** or **File**, **Save All**
File, **Save Project Group As**	**File**, **Save *SolutionName* As**
File, **Save Selection**	**File**, **Save Selected Items**
Format, **Size to Grid**	No equivalent. Controlled by **Snap to Grid** environment option.
Help, **Microsoft on the Web**	**Help**, **Show Start Page**
Project, **Add Class Module**	**Project**, **Add Class**
Project, **Add Data Environment**	No equivalent. Data Environment is not supported.
Project, **Add Data Report**	No equivalent. Data Reports are not supported.
Project, **Add DHTML Page**	No equivalent. DHTML Pages are not supported.
Project, **Add File**	**Project**, **Add New Item** or **Project**, **Add Existing Item** **Note** In Visual Basic 6.0 the file itself was added to the project; in Visual Basic .NET a copy of the file is added.
Project, **Add Form**	**Project**, **Add Windows Form** or **Project**, **Add Inherited Form**
Project, **Add MDI Form**	**Project**, **Add Windows Form** or **Project**, **Add Inherited Form**
Project, **Add Property Page**	No equivalent. Property Pages are not supported.
Project, **Add User Document**	No equivalent. ActiveX Documents are not supported.
Project, **Add Web Class**	No equivalent. Web Classes are not supported.
Project, **Components**	No equivalent. Components are managed in the Toolbox.
Project, **References**	No equivalent. References are managed in Solution Explorer.
Project, **Remove *item***	**Edit**, **Delete** **Note** **Remove** removed the item from the project but not from disk; **Delete** deletes the item.
Run, **Break**	**Debug**, **Break All**
Run, **End**	**Debug**, **Stop Debugging**
Run, **Restart**	**Debug**, **Restart**
Run, **Start**	**Debug**, **Start**
Run, **Start With Full Compile**	**Debug**, **Start Without Debugging**
Tools, **Add Procedure**	No equivalent. Procedures are added in the Code Editor.
Tools, **Menu Editor**	No equivalent. The Menu Editor is replaced by the **MainMenu** and **ContextMenu** components.

Tools, **Procedure Attributes**	No equivalent.
Tools, **SourceSafe**	**File**, **Source Control**
View, **Call Stack**	**Debug**, **Windows**, **Call Stack**
View, **Color Palette**	No equivalent. There is no Color Palette window.
View, **Data View Window**	No equivalent. There is no Data View window.
View, **Definition**	**Go To Definition** (context menu only)
View, **Form Layout Window**	No equivalent. There is no Form Layout window.
View, **Immediate Window**	**Debug**, **Windows**, **Immediate**
View, **Last Position**	No equivalent. Use Bookmarks.
View, **Locals Window**	**Debug**, **Windows**, **Locals**
View, **Object**	**View**, **Designer**
View, **Object Browser**	**View**, **Other Windows**, **Object Browser**
View, **Project Explorer**	**View**, **Solution Explorer**
View, **Watch Window**	**Debug**, **Windows**, **Watch**
Window, **Arrange Icons**	No equivalent.
Window, **Cascade**	Only available in MDI layout mode.
Window, **Tile Horizontally**	Only available in MDI layout mode.
Window, **Tile Vertically**	Only available in MDI layout mode.
Window, Window list	**Windows...**

Keyboard Mapping Changes

Keyboard mappings in Visual Studio .NET are standardized so that you can use the same keyboard shortcuts for all languages and tools. The result is that some of the familiar Visual Basic 6.0 keyboard shortcuts now invoke different functions. For example, in Visual Basic 6.0, F8 was the keyboard shortcut for Step Into when debugging; in Visual Basic .NET, F8 is the shortcut for Go to Next Location in the Code Editor.

If you prefer the Visual Basic 6.0 keyboard mappings, you can set them by choosing **Visual Basic 6** from the **Keyboard Scheme** drop-down list on the **My Profile** tab of the Visual Studio **Start Page**.

See Also

Customizing the Development Environment | Using the Debugger | Introduction to Visual Basic .NET for Visual Basic Veterans

Project Changes in Visual Basic .NET

Projects in Visual Basic .NET are similar to projects in Visual Basic 6.0; however, there are some differences that you should be aware of.

In Visual Basic 6.0, projects used a reference-based model — the project file contained references to project items specifying their path. For example, when you added a text file to a project, the project file pointed to the location of the file; when the project was built, the text file would be loaded from that location.

Visual Basic .NET uses a folder-based model — all project items are placed in the project folder hierarchy. When you add a text file, a copy of the file is placed in the project folder; when the project is built, the text file is loaded from that copy of the file.

In Visual Basic 6.0, the **Remove** command removed an item from a project, but the file still remained on disk. In Visual Basic .NET, the **Remove** command is replaced by a **Delete** command, which both removes and deletes the file. The **Exclude From Project** command can be used to exclude a file from a project without deleting it.

In Visual Basic 6.0, multiple projects could be added in the Project Explorer; multiple projects were called Project Groups. In Visual Basic .NET, Solution Explorer replaces the Project Explorer; Project Groups are replaced by solutions. Where Project Groups could only contain Visual Basic projects, solutions can contain projects created in any combination of Visual Studio .NET languages.

In Visual Basic 6.0, project files (.vbp) and project group files (.vbg) were text files that could be edited directly in a text editor. In Visual Basic .NET, project and solution files are in an XML format; they should not be edited directly.

In This Section

Project Property Changes in Visual Basic .NET
Describes project property differences between Visual Basic 6.0 and Visual Basic .NET.
Project Type Changes in Visual Basic .NET
Describes project type differences between Visual Basic 6.0 and Visual Basic .NET.
Reference Changes in Visual Basic .NET
Describes differences in reference handling between Visual Basic 6.0 and Visual Basic .NET.

See Also

Managing Projects
Describes the project system for managing your applications as you write them.
Introduction to Visual Basic .NET for Visual Basic Veterans
Describes differences between Visual Basic 6.0 and Visual Basic .NET.

Project Property Changes in Visual Basic .NET

Project properties in both Visual Basic 6.0 and Visual Basic .NET specify various global settings for a project. In Visual Basic 6.0, project properties were set in the Project Properties dialog box. In Visual Basic .NET, project properties are set in the Property Pages dialog box; some properties can be set on a per-configuration basis.

Many of the project properties differ between Visual Basic 6.0 and Visual Basic .NET. The following list provides a comparison of Visual Basic 6.0 and Visual Basic .NET project properties. Note that not all properties are available for all project types.

Visual Basic 6.0	**Visual Basic .NET**
Command Line Arguments	**Command line arguments** (configuration dependent)
Compile options	No equivalent. For more information, see Visual Basic Compiler Options.
Conditional Compilation Arguments	**Conditional Compilation Constants — Custom Constants**
DLL Base Address	**DLL base address**
Help File Name	No equivalent. For more information, see Help Support Changes in Visual Basic .NET.
Icon	**Application icon**
Project Description	No equivalent. Version information for assemblies is stored in the AssemblyInfo.vb file.
Project Help Context ID	No equivalent. For more information, see Help Support Changes in Visual Basic .NET.
Project Name	No equivalent. Replaced by **Assembly name** and **Root namespace**.
Project Type	**Output type**
Remote Server	No equivalent.
Remove information about Unused ActiveX Controls	No equivalent.
Require License Key	No equivalent.
Retained In Memory	No equivalent. This property was related to MTS components, which are not supported in Visual Basic .NET.
Start Mode	No equivalent.
Startup Object	**Startup object**
Threading Model	No equivalent. Visual Basic .NET supports multithreading.
Title	No equivalent. Version information for assemblies is stored in the AssemblyInfo.vb file.
Unattended Execution	No equivalent.
Upgrade ActiveX Controls	No equivalent.
Use existing browser	No equivalent.
Version Compatibility	No equivalent. Visual Basic .NET assemblies eliminate the need for binary compatibility settings.
Version Information	No equivalent. Version information for assemblies is stored in the AssemblyInfo.vb file.
Version Number	No equivalent. Version information for assemblies is stored in the AssemblyInfo.vb file.
When this project starts	**Start Action**

See Also

Project Changes in Visual Basic .NET | Component Authoring Changes in Visual Basic .NET

Project Type Changes in Visual Basic .NET

The types of projects that can be created with Visual Basic .NET differ from Visual Basic 6.0. In some cases, the project types are essentially the same but the names have changed; in other cases some Visual Basic 6.0 project types have no direct equivalent.

The following list provides a comparison of Visual Basic 6.0 and Visual Basic .NET project types.

Visual Basic 6.0	**Visual Basic .NET**
Standard EXE	Windows Application
ActiveX DLL	Class Library
ActiveX EXE	Class Library
ActiveX Control	Windows Control Library
ActiveX Document	No equivalent. Visual Basic .NET can interoperate with ActiveX Documents.
DHTML Application	No equivalent. Use ASP.NET Web Application.
IIS Application (Web Class)	No equivalent. Use ASP.NET Web Application.

See Also

Visual Basic and Visual C# Projects | Web Changes in Visual Basic .NET | Project Changes in Visual Basic .NET

Reference Changes in Visual Basic .NET

In Visual Basic 6.0, project references to type libraries were managed in the References dialog box. Once a reference was added, you could access the object in code.

In Visual Basic .NET, references are made to assemblies and are added and removed in the project's References node in Solution Explorer. References can also be made to COM components. Project-to-project references and Web references to XML Web services are also supported.

In Visual Basic 6.0, controls were added to a project in the Components dialog box. Once added, they were visible in the Toolbox.

In Visual Basic .NET, controls are added to the Toolbox in the Customize Toolbox dialog box.

See Also

Referencing Namespaces and Components | Project Changes in Visual Basic .NET

Web Changes in Visual Basic .NET

Visual Basic 6.0 included several features that supported programming for the Web: Internet Information Services (IIS) applications (Web classes), DHTML applications, ActiveX documents, and ActiveX controls that could be downloaded to Web pages.

Visual Basic .NET was built from the ground up to support Web programming with features such as ASP.NET Web applications, XML Web services, and much more. Because Visual Basic .NET is built on a new architecture, Visual Basic 6.0 Web features either are no longer supported or have changed substantially; however, your knowledge of Web programming should help you to quickly transition to the Web technologies in Visual Basic .NET.

In Visual Basic 6.0, IIS applications used the Active Server Pages (ASP) model to create applications that ran on Internet Information Services. In Visual Basic .NET, ASP.NET technology allows you to create the application using Web Forms pages, and to create components using XML Web services. These technologies make programming for the Web very similar to programming for Windows in Visual Basic 6.0. For more information, see Introduction to ASP.NET Web Applications in Visual Studio.

Visual Basic 6.0 DHTML applications used the Dynamic HTML object model and Visual Basic code to create applications that could respond to actions performed by the user in a browser. Visual Basic .NET Web Forms expand upon the DHTML model, providing richer dynamic user-interface capabilities as well as client-side validation.

Visual Basic 6.0 ActiveX documents are not supported in Visual Basic .NET. You can still interoperate with ActiveX documents from your Visual Basic .NET Web applications, but development should be maintained in Visual Basic 6.0.

Visual Basic .NET also allows you to create ActiveX controls that can be downloaded to Web pages, or to use existing ActiveX controls in your Visual Basic .NET applications.

See Also

Creating Web Applications and Services | Working with Both Visual Basic 6.0 and Visual Basic .NET | Introduction to Visual Basic .NET for Visual Basic Veterans | Web Namespaces in Visual Studio

WebClass Changes in Visual Basic .NET

In Visual Basic 6.0, WebClass projects (also known as IIS Application projects) were used to create Web applications based on Active Server Page (ASP) technology.

In Visual Basic .NET, ASP.NET Web Application projects are used to create Web applications based on the newer ASP.NET technology. When a Visual Basic 6.0 WebClass project is upgraded to Visual Basic .NET, it is converted to an ASP.NET Web Application project.

In Visual Basic 6.0, WebClass projects had a **StateManagement** property that could be used to keep an instance of the webclass alive between requests. This was done by setting the **StateManagement** property to **2 – wcRetainInstance** at design time; there was also a **ReleaseInstance** method that could be used to terminate an instance.

In Visual Basic .NET, ASP.NET Web applications do not have a **StateManagement** property; the model for managing application state differs considerably. Any code related to state management will need to be replaced. For more information, see Web Forms State Management.

Function and **Sub** procedures in your Visual Basic 6.0 code (for example, **ProcessTags** or **Respond**) will have their scope changed from **Private** to **Public** to allow the WebClass Compatibility runtime to execute them.

Certain Visual Basic 6.0 WebClass events are not supported in ASP.NET (for example, **Initialize**, **BeginRequest**, **EndRequest**, and **Terminate**). These event procedures will be upgraded but will not be called at runtime. After upgrading, you will need to move any code in these events to equivalent ASP.NET events.

Declarations will be added to your project: one for the WebClass and one for each of the WebItems and Templates in the WebClass project. A **Page_Load** event procedure will be added to the project, creating first a WebClass object and then WebItem objects for each of the WebItems and Templates associated with the Visual Basic 6.0 WebClass project. Finally, in the **Page_Load** event procedure, you will see a call to the WebClass Compatibility runtime: **WebClass.ProcessEvents**. This allows the runtime to render the WebItem specified in the Request URL. This code is the only new code added to your upgraded project and only serves to emulate the underlying behavior of the Visual Basic 6.0 WebClass runtime.

> **Note** There are a number of behavioral differences for properties, methods, and events between ASP and ASP.NET objects. Before upgrading a WebClass to ASP.NET, you should understand these differences so you know how to modify your code. For more information, see Migrating to Web Forms.

See Also

Upgrading WebClass Projects to Visual Basic .NET | Introduction to ASP.NET Web Applications in Visual Studio | Web Forms Page Processing | Creating Web Applications and Services

Data Changes in Visual Basic .NET

In Visual Basic 6.0, data access was accomplished using ActiveX Data Objects (ADO). In Visual Basic .NET, data access is accomplished using ADO.NET, which is a part of the .NET Framework. There are a number of differences both conceptually and in terms of tasks between the two technologies. For more information on conceptual differences between ADO and ADO.NET, see Comparison of ADO.NET and ADO.

> **Note** Visual Basic 6.0 also provided backward compatibility support for Remote Data Objects (RDO) and Data Access Objects (DAO). These technologies are supported in Visual Basic .NET only as COM objects; applications that use RDO or DAO data binding cannot be upgraded to Visual Basic .NET. For more information, see Upgrading Applications Created in Previous Versions of Visual Basic.

In Visual Basic 6.0, there were two common methods of implementing data access in an application: at design time by binding to an ADODC control or by using a Data Environment, or at run time by creating and interacting with recordset objects programmatically. In Visual Basic .NET, there are two analogous methods of implementing data access: at design time by using data adapters and datasets, or at run time by adding code to create data adapters and datasets programmatically. For more information, see Creating Data Adapters and ADO.NET Datasets.

In Visual Basic 6.0, data binding was accomplished by setting the binding-related properties of a control: **DataChanged**, **DataField**, **DataFormat**, **DataMember**, and **DataSource**. In most cases, the display property of a control (for example, the Text property of a **TextBox** control) was bound to a field in a data source.

In Visual Basic .NET, data binding has a much broader definition — you can bind any property of any control to any structure that contains data. Binding in Windows Forms is accomplished via a **DataBindings** property that contains a collection of **Binding** objects for the control. For more information, see Data Binding and Windows Forms.

> **Note** ADO data binding is supported for backward compatibility via the Microsoft Visual Basic .NET Compatibility Data Runtime.

In Visual Basic 6.0, data-bound forms were automatically created by dragging database elements from a Data Environment onto a form.

In Visual Basic .NET, you automatically create data-bound forms by running the Data Form Wizard. For more information, see Data Form Wizard.

In addition to support for ADO.NET, Visual Basic .NET also provides support for direct programmatic access to databases using data commands. For more information, see Performing Database Operations Directly.

See Also

Introduction to Data Access with ADO.NET | Data Access Technologies | Introduction to Visual Basic .NET for Visual Basic Veterans | Data Namespaces in Visual Studio

Component Authoring Changes in Visual Basic .NET

Although both Visual Basic 6.0 and Visual Basic .NET provide capabilities for creating components, there are some significant differences in component authoring. In Visual Basic 6.0, component authoring was all about creating COM components: ActiveX Controls, ActiveX DLLs, and ActiveX EXEs that could be used in COM applications.

In Visual Basic .NET, components are based on the .NET Framework; you create components that can be used in applications built using the .NET Framework. Components built with Visual Basic .NET are based on inheritance — every component derives from either the **Component** or **Control** base class. For more information, see Component Classes.

Visual Basic 6.0 provided two models for threading — components could be either single threaded or apartment threaded. Visual Basic .NET supports true multithreaded components. For more information, see Multithreading in Components.

In Visual Basic 6.0, the **Instancing** property of a class controlled the access level and the way a component could be created. In Visual Basic .NET, setting the access modifier and access level in the constructor for a component class controls instancing. For more information, see Component Instancing Changes in Visual Basic.

Binary compatibility for components was important in Visual Basic 6.0 in order to prevent version conflicts; it was controlled by the **Version Compatibility** property for the component project. In Visual Basic .NET, versioning is built into the assemblies for components; you no longer need to worry about setting binary compatibility. For more information, see Components and Assemblies.

In Visual Basic 6.0, you could create ActiveX controls (also referred to as user controls) that could be used in Windows applications. In Visual Basic .NET, you can create user controls for Windows Forms, or you can inherit from existing controls to add functionality. For more information, see Control Authoring for Windows Forms.

In Visual Basic 6.0, you could create components that used Microsoft Message Queue (MSMQ) to pass information between applications. In Visual Basic .NET, messaging is built into the .NET Framework. For more information, see Creating Messaging Components.

In Visual Basic 6.0, you could create MTS components that worked with Microsoft Transaction Server (MTS) to participate in transactions. With Visual Basic .NET, transactions are handled by the .NET Framework by adding a Transactional Component to your project or by adding a Transaction attribute to an existing class.

See Also

Component Authoring | Inheritance | Introduction to Visual Basic .NET for Visual Basic Veterans | Component Model Namespaces in Visual Studio

UserControl Changes in Visual Basic .NET

In Visual Basic 6.0, UserControl projects (also known as ActiveX Control projects) were used to create ActiveX controls. After they were compiled, UserControls could be hosted in any container that supported ActiveX, including Visual Basic 6.0 forms and Internet Explorer.

In Visual Basic .NET, Windows Control Library projects are used to create controls that can be hosted in Windows Forms applications. When a Visual Basic 6.0 UserControl project is upgraded to Visual Basic .NET, it is upgraded to a Windows Control Library; the resulting control can no longer be hosted in Internet Explorer.

In Visual Basic 6.0, before adding a UserControl to a form at design time, you had to close the UserControl Designer; this step automatically compiled (built) the control. When subsequent changes were made to the UserControl, they would not be reflected in the instance of the UserControl on the form until the application was run; if open, the UserControl designer would automatically close.

In Visual Basic .NET, a UserControl can be added to a form without closing the UserControl Designer; however, the UserControl must be built before it can be added to a form. Simply closing the UserControl designer does not build the control. Any subsequent changes to the UserControl are not reflected on the form designer until the UserControl is rebuilt; a UserControl is automatically rebuilt each time you debug your test application.

In Visual Basic 6.0, the **ReadProperties** and **WriteProperties** events were used to retrieve or save a UserControl's values to a **PropertyBag** object.

In Visual Basic .NET, the **PropertyBag** object is no longer supported, and the **ReadProperties** and **WriteProperties** events no longer exist. Instead, serialization is used to save and retrieve properties in either a binary or SOAP format.

See Also

PropertyBag Changes in Visual Basic .NET | Walkthrough: Persisting an Object in Visual Basic .NET |
Control Authoring for Windows Forms | The contents of the template file were not upgraded |
WebClass state management is not supported | WebClass event is not supported

PropertyBag Changes in Visual Basic .NET

In Visual Basic 6.0, the **PropertyBag** object was used to persist an object's data between instances, allowing you to store values and retrieve them the next time the object was instantiated. Although you could set an object's properties to a default value at design time, any values entered at run time were lost when the object was destroyed. The **PropertyBag** object allowed a copy of the object to be persisted in a binary format and retrieved for later reuse. For example, a class used to calculate loans might use a **PropertyBag** object to persist an interest rate between instances, rather than entering it each time the class is used.

In Visual Basic .NET, the **PropertyBag** object no longer exists, but you can still persist an object's data using the **System.Runtime.Serialization** and **System.IO** namespaces. In addition to persisting an object in binary format, you can also persist an object in a SOAP format.

To persist an object in Visual Basic .NET, you first mark the class as **Serializable**. When instantiating the object, you use a **Stream** class and a **BinaryFormatter** or **SoapFormatter** class to retrieve the object's data from a file. Before destroying the object, you again use the **Stream** and formatter classes to write the object's values to the file.

For an example of how to persist an object, see Walkthrough: Persisting an Object.

See Also

UserControl Changes in Visual Basic .NET | System.Runtime.Serialization Namespace | System.IO Namespace

Forms Changes in Visual Basic .NET

In Visual Basic 6.0 (and all earlier versions of Visual Basic), Windows applications were creating using forms that were an integral part of the Visual Basic runtime. In Visual Basic .NET, Windows applications are created using Windows Forms, which are provided by the .NET Framework.

Windows Forms and Windows Forms controls provide a richer, more extensible architecture for developing Windows applications; however, there are a number of differences between Windows Forms and controls and Visual Basic 6.0 Forms and controls. The following topics are designed to help you discover the differences.

In This Section

Forms Task Changes in Visual Basic .NET
Describes differences in common tasks for working with forms.
Control Changes in Visual Basic .NET
Provides links to topics describing the differences between Visual Basic 6.0 and Windows Forms controls and objects.
Control Property, Method, and Event Changes in Visual Basic .NET
Provides links to topics describing differences in properties, methods, and events for forms, controls, and other objects.
Control Array Changes in Visual Basic .NET
Describes the differences in control arrays between Visual Basic 6.0 and Visual Basic .NET.
Form Positioning Changes in Visual Basic .NET
Describes differences in setting a form's screen location between Visual Basic 6.0 and Visual Basic .NET.
Menu Changes in Visual Basic .NET
Describes differences in creating menus between Visual Basic 6.0 and Visual Basic .NET.
MDI Changes in Visual Basic .NET
Describes differences in multiple-document interface (MDI) applications between Visual Basic 6.0 and Visual Basic .NET.
Dialog Box Changes in Visual Basic .NET
Describes differences in creating dialog boxes between Visual Basic 6.0 and Visual Basic .NET.
Color Changes in Visual Basic .NET
Describes differences in color handling between Visual Basic 6.0 and Visual Basic .NET.
Help Support Changes in Visual Basic .NET
Describes differences in implementing Help between Visual Basic 6.0 and Visual Basic .NET.
Printing Changes in Visual Basic .NET
Describes differences in printing support for applications between Visual Basic 6.0 and Visual Basic .NET.

Related Sections

Event Changes in Visual Basic .NET
Describes differences in writing event handlers between Visual Basic 6.0 and Visual Basic .NET.
Drag and Drop Changes in Visual Basic .NET
Describes differences in implementing drag and drop between Visual Basic 6.0 and Visual Basic .NET.
Graphics Changes in Visual Basic .NET
Describes changes in graphics methods between Visual Basic 6.0 and Visual Basic .NET.
Introduction to Windows Forms
Introduces basic concepts related to forms.
What's New in Windows Forms and Controls
Describes new features for Windows Forms.
Windows Applications Namespaces in Visual Studio
Introduces the namespaces related to Windows Forms.
Introduction to Visual Basic .NET for Visual Basic Veterans
Provides links to additional topics describing differences between Visual Basic 6.0 and Visual Basic .NET.

Forms Task Changes in Visual Basic .NET

There are many differences between Windows Forms and Visual Basic 6.0 Forms. The following table provides a list of tasks that can be performed with forms and their methodologies. You can use this list to transition your existing Visual Basic 6.0 Forms projects to the new .NET Framework model for Windows Forms.

Task	Visual Basic 6.0 Forms	Windows Forms
Set the Tab Indexing for all the controls on a form.	Select each control on the form and set its **TabIndex** property to the next number in the tab order for your form.	From the **View** menu choose **Tab Order**. The form displays a numeral next to each control that represents the control's TabIndex property setting. Click on each control in the desired tab order. For more information, see Setting the Tab Order on Windows Forms.
Determine the modifier key (ALT, SHIFT, CTRL) pressed during keyboard and mouse events.	Use the parameters provided by the mouse event handler.	Compare the Form class's ModifierKeys method against the modifier key value, found in the Keys enumeration. For more information, see Determining Which Modifier Key Was Pressed
Align or size controls based on the location and size of a single control.	Click on each control to align or size. The last control selected is used to determine the location and size of all controls selected.	Click on each control to align or size. The first control selected is used to determine the location and size of all controls selected. For more information, see Aligning Multiple Controls on Windows Forms.
Change the name of a form.	Change the name specified in the **Name** property of the form in the Properties window.	Click on the form's class file in the Solution Explorer and then change the **Filename** property in the Properties window.
Change the run-time display position of a form.	Use the Form Layout window to position the form, or set its **Startup** property in the Properties window.	With the form selected in the Windows Forms Designer, set the StartPosition property in the Properties window.
Display a form as a dialog box.	In code, use the **Show** method and specify that the form be displayed modally using the **vbModal** value.	In code, use the ShowDialog method of the form. For details, see Displaying Dialog Boxes for Windows Forms.
Group or layer controls on a form.	Layer the controls within the **Frame** control. To move the controls as a group, select them all.	Place the controls within a GroupBox control. Moving the group box moves the controls as well.
Iterate through the controls on a form.	Use a control array.	Use the Controls Collection. For details, see Collections in Visual Basic .NET.

See Also

What's New in Windows Forms and Controls | Introduction to Visual Basic .NET for Visual Basic Veterans | Form Positioning Changes in Visual Basic .NET

Control Changes in Visual Basic .NET

New controls and objects in Visual Basic .NET replace the intrinsic controls, objects, and collections in Visual Basic 6.0. In some cases, the names have changed; in many cases the properties, methods, and events are different.

The following topics present mappings between Visual Basic 6.0 controls and objects and their Visual Basic .NET equivalents.

In This Section

App Object Changes in Visual Basic .NET

CheckBox Control Changes in Visual Basic .NET

Clipboard Object Changes in Visual Basic .NET

ComboBox Control Changes in Visual Basic .NET

CommandButton Control Changes in Visual Basic .NET

Controls Collection Changes in Visual Basic .NET

Data Control Changes in Visual Basic .NET

DataGrid Control Changes in Visual Basic .NET

DirListBox Control Changes in Visual Basic .NET

DriveListBox Control Changes in Visual Basic .NET

FileListBox Control Changes in Visual Basic .NET

Font Object Changes in Visual Basic .NET

Form Object Changes in Visual Basic .NET

Forms Collection Changes in Visual Basic .NET

Frame Control Changes in Visual Basic .NET

HScrollBar Control Changes in Visual Basic .NET

Image Control Changes in Visual Basic .NET

Label Control Changes in Visual Basic .NET

Line Control Changes in Visual Basic .NET

ListBox Control Changes in Visual Basic .NET

MDIForm Object Changes in Visual Basic .NET

Menu Object Changes in Visual Basic .NET

OLE Container Control Changes in Visual Basic .NET

OptionButton Control Changes in Visual Basic .NET

PictureBox Control Changes in Visual Basic .NET

Printer Object Changes in Visual Basic .NET

Printers Collection Changes in Visual Basic .NET

RDO Data Control Changes in Visual Basic .NET

Screen Object Changes in Visual Basic .NET

Shape Control Changes in Visual Basic .NET

TextBox Control Changes in Visual Basic .NET

Timer Control Changes in Visual Basic .NET

UserControl Changes in Visual Basic .NET

VBControlExtender Object Changes in Visual Basic .NET

VScrollBar Control Changes in Visual Basic .NET

Related Sections

Windows Forms Controls
Provides links to topics about the controls designed specifically to work with Windows Forms.

Control Property, Method, and Event Changes in Visual Basic .NET
Provides links to topics describing differences in properties, methods, and events for forms, controls, and other objects.

Controls and Programmable Objects Compared in Different Languages and Libraries
Provides a comparison of various different controls and objects.

What's New in Windows Forms and Controls
Describes new features for Windows Forms.

App Object Changes in Visual Basic .NET

The **App** object in Visual Basic 6.0 was a global object used to set or retrieve information about the application. There is no direct equivalent for the **App** object in Visual Basic .NET; however, most of the properties can be mapped to equivalent properties in the .NET Framework.

> **Note** The **App** object's Version Information properties are replaced by Assembly attributes in Visual Basic .NET. Version Information properties were set in the Project Properties dialog box; Assembly attributes are set by editing the AssemblyInfo file in the Visual Basic .NET project. For more information, see Attributes.

App Object Properties

Visual Basic 6.0	Visual Basic .NET Equivalent
Comments	**AssemblyDescription** attribute
CompanyName	**AssemblyCompany** attribute
EXEName	**System.Reflection.Assembly.GetExecutingAssembly.Location** **Note** The **Location** property returns the full path including the file name; the **EXEName** property returned only the file name.
FileDescription	**System.Diagnostics.FileVersionInfo.GetVersionInfo(System.Reflection.Assembly.GetExecutingAssembly.Location).FileDescription**
HelpFile	No equivalent. For more information, see Help Support Changes in Visual Basic .NET.
HInstance	**System.Runtime.InteropServices.Marshal.GetHINSTANCE(System.Reflection.Assembly.GetExecutingAssembly.GetModules()(0)).ToInt32**
LegalCopyright	**System.Diagnostics.FileVersionInfo.GetVersionInfo(System.Reflection.Assembly.GetExecutingAssembly.Location).LegalCopyright**
LegalTrademarks	**System.Diagnostics.FileVersionInfo.GetVersionInfo(System.Reflection.Assembly.GetExecutingAssembly.Location).LegalTrademarks**
LogEvent method	No equivalent. Logging in Visual Basic .NET is handled via Event Logs. For more information, see Creating and Removing Custom Event Logs.
LogMode	No equivalent. Logging in Visual Basic .NET is handled via Event Logs. For more information, see Creating and Removing Custom Event Logs.
LogPath	No equivalent. Logging in Visual Basic .NET is handled via Event Logs. For more information, see Creating and Removing Custom Event Logs.
Major	**System.Diagnostics.FileVersionInfo.GetVersionInfo(System.Reflection.Assembly.GetExecutingAssembly.Location).FileMajorPart** **Note** The format for version numbers is different in Visual Basic .NET. For more information, see Version Numbering Changes in Visual Basic .NET.
Minor	**System.Diagnostics.FileVersionInfo.GetVersionInfo(System.Reflection.Assembly.GetExecutingAssembly.Location).FileMinorPart** **Note** The format for version numbers is different in Visual Basic .NET. For more information, see Version Numbering Changes in Visual Basic .NET.
NonModalAllowed	No equivalent. This was a read-only property related to ActiveX .dll files. The .NET common language runtime automatically manages this behavior.
OleRequestPendingMsgText	No equivalent. This property relates to OLE automation, which is not supported by Visual Basic .NET.
OleRequestPendingMsgTitle	No equivalent. This property relates to OLE automation, which is not supported by Visual Basic .NET.
OleRequestPendingTimeout	No equivalent. This property relates to OLE automation, which is not supported by Visual Basic .NET.
OleServerBusyMsgText	No equivalent. This property relates to OLE automation, which is not supported by Visual Basic .NET.
OleServerBusyMsgTitle	No equivalent. This property relates to OLE automation, which is not supported by Visual Basic .NET.

OleServerBusyRaiseError	No equivalent. This property relates to OLE automation, which is not supported by Visual Basic .NET.
OleServerBusyTimeout	No equivalent. This property relates to OLE automation, which is not supported by Visual Basic .NET.
Path	**System.Reflection.Assembly.GetExecutingAssembly.Location** **Note** The **Location** property returns the full path including the file name; the **Path** property returned only the path.
PrevInstance	No equivalent. For more information, see PrevInstance Property Changes in Visual Basic .NET.
ProductName	**System.Diagnostics.FileVersionInfo.GetVersionInfo(System.Reflection.Assembly.GetExecutingAssembly.Location).ProductName**
RetainedProject	No equivalent. Visual Basic .NET does not have the ability to retain a project in memory.
Revision	**System.Diagnostics.FileVersionInfo.GetVersionInfo(System.Reflection.Assembly.GetExecutingAssembly.Location).FileBuildPart** – or – **System.Diagnostics.FileVersionInfo.GetVersionInfo(System.Reflection.Assembly.GetExecutingAssembly.Location).FilePrivatePart** **Note** The format for version numbers is different in Visual Basic .NET. For more information, see Version Numbering Changes in Visual Basic .NET.
StartLogging method	No equivalent. Logging in Visual Basic .NET is handled via Event Logs. For more information, see Creating and Removing Custom Event Logs.
StartMode	No equivalent. In Visual Basic 6.0, this property was used to start an application as an ActiveX component. ActiveX component creation is not supported in Visual Studio .NET.
TaskVisible	No equivalent. For more information, see TaskVisible Property Changes in Visual Basic .NET.
ThreadID	No equivalent. The threading model is different in Visual Basic .NET.
Title	No equivalent. In Visual Basic .NET, the **Text** property of the main form in the application determines the title that will be shown in the Task Manager.
UnattendedApp	No equivalent. For unattended applications in Visual Basic .NET, choose a Console Application project.

See Also

Control Changes in Visual Basic .NET | Screen Object Changes in Visual Basic .NET

CheckBox Control Changes in Visual Basic .NET

The **CheckBox** control in Visual Basic 6.0 is replaced by the Windows Forms **CheckBox** control in Visual Basic .NET. The names of some properties, methods, events, and constants are different, and in some cases there are differences in behavior.

The following tables list Visual Basic 6.0 properties, methods, and events and their Visual Basic .NET equivalents. Where applicable, constants are indented beneath the property or method. All Visual Basic .NET constants map to the **System.Windows.Forms** namespace unless otherwise noted.

Links are provided as necessary to topics explaining differences in behavior. Where there is no direct equivalent in Visual Basic .NET, links are provided to topics that present alternatives.

CheckBox Properties

Visual Basic 6.0	Visual Basic .NET Equivalent
Alignment **0 - Left Justify** **1 - Right Justify**	**CheckAlign** **System.Drawing.ContentAlignment.MiddleLeft** **System.Drawing.ContentAlignment.MiddleRight**
Appearance **0 – Flat** **1 - 3D**	**FlatStyle** **FlatStyle.Flat** **FlatStyle.Standard**
BackColor	**BackColor** **Note** Colors are handled differently in Visual Basic .NET. For more information, see Color Behavior Changes in Visual Basic .NET. For a list of constants, see Color Changes in Visual Basic .NET.
Caption	**Text**
CausesValidation	**CausesValidation**
Container	**Parent**
DataChanged	No equivalent. For more information, see Data Changes in Visual Basic .NET.
DataField	No equivalent. For more information, see Data Changes in Visual Basic .NET.
DataFormat	No equivalent. For more information, see Data Changes in Visual Basic .NET.
DataMember	No equivalent. For more information, see Data Changes in Visual Basic .NET.
DataSource	No equivalent. For more information, see Data Changes in Visual Basic .NET.
DisabledPicture	No equivalent. For more information, see Style Property Changes in Visual Basic .NET.
DownPicture	No equivalent. For more information, see Style Property Changes in Visual Basic .NET.
DragIcon	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
DragMode	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
Enabled	**Enabled**
Font	**Font** **Note** Fonts are handled differently in Visual Basic .NET. For more information, see Font Changes in Visual Basic .NET.
FontBold	**Font** **Note** Fonts are handled differently in Visual Basic .NET. For more information, see Font Changes in Visual Basic .NET.

FontItalic	**Font** **Note** Fonts are handled differently in Visual Basic .NET. For more information, see Font Changes in Visual Basic .NET.
FontName	**Font** **Note** Fonts are handled differently in Visual Basic .NET. For more information, see Font Changes in Visual Basic .NET.
FontSize	**Font** **Note** Fonts are handled differently in Visual Basic .NET. For more information, see Font Changes in Visual Basic .NET.
FontStrikethru	**Font** **Note** Fonts are handled differently in Visual Basic .NET. For more information, see Font Changes in Visual Basic .NET.
FontUnderline	**Font** **Note** Fonts are handled differently in Visual Basic .NET. For more information, see Font Changes in Visual Basic .NET.
ForeColor	**ForeColor** **Note** Colors are handled differently in Visual Basic .NET. For more information, see Color Behavior Changes in Visual Basic .NET. For a list of constants, see Color Changes in Visual Basic .NET.
Height	**Height, Size** **Note** Coordinates are handled differently in Visual Basic .NET. For more information, see Coordinate System Changes in Visual Basic .NET.
HelpContextID	No equivalent. For more information, see Help Support Changes in Visual Basic .NET.
HWnd	**Handle**
Index	No equivalent. For more information, see Control Array Changes in Visual Basic .NET.
Left	**Left** **Note** Coordinates are handled differently in Visual Basic .NET. For more information, see Coordinate System Changes in Visual Basic .NET.
MaskColor	No equivalent. For more information, see MaskColor Changes in Visual Basic .NET.
MouseIcon	No equivalent. For more information, see Cannot set a custom MousePointer.
MousePointer	**Cursor** For a list of constants, see MousePointer Changes in Visual Basic .NET.
Name	**Name**
OLEDropMode	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
Parent	**FindForm** method
Picture	**Image**
RightToLeft **True** **False**	**RightToLeft** **RightToLeft.Yes** **RightToLeft.No**

Style **0 – Standard** **1 - Graphical**	**Appearance** **Note** There are differences in how the Graphical style is handled in Visual Basic .NET. For more information, see Style Property Changes in Visual Basic .NET. **Appearance.Normal** **Appearance.Button**
TabIndex	**TabIndex**
TabStop	**TabStop**
Tag	**Tag**
ToolTipText	**ToolTip** component For more information, see ToolTip Changes in Visual Basic .NET.
Top	**Top** **Note** Coordinates are handled differently in Visual Basic .NET. For more information, see Coordinate System Changes in Visual Basic .NET.
UseMaskColor	No equivalent. For more information, see MaskColor Changes in Visual Basic .NET.
Value **0 – vbUnchecked** **1 – vbChecked** **2 - vbGrayed**	**CheckState** **CheckState.Unchecked** **CheckState.Checked** **CheckState.Indeterminate**
Visible	**Visible**
WhatsThisHelpID	No equivalent. For more information, see Help Support Changes in Visual Basic .NET.
Width	**Width, Size** **Note** Coordinates are handled differently in Visual Basic .NET. For more information, see Coordinate System Changes in Visual Basic .NET.

CheckBox Methods

Visual Basic 6.0	Visual Basic .NET Equivalent
Drag	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
Move	**SetBounds** **Note** Coordinates are handled differently in Visual Basic .NET. For more information, see Coordinate System Changes in Visual Basic .NET.
OLEDrag	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
Refresh	**Refresh**
SetFocus	**Focus**
ShowWhatsThis	No equivalent. For more information, see Help Support Changes in Visual Basic .NET.
ZOrder **0 – vbBringToFront** **1 - vbSendToBack**	**BringToFront()** or **SendToBack()** functions **BringToFront()** **SendToBack()**

CheckBox Events

Visual Basic 6.0	Visual Basic .NET Equivalent
Click	**CheckStateChanged** **Note** In Visual Basic 6.0, the **Click** event was raised when the **CheckBox** state was changed programmatically. In Visual Studio .NET, the **Click** event is not raised; therefore, you should use the **CheckStateChanged** event.
DragDrop	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
DragOver	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
GotFocus	**Enter**
KeyDown	**KeyDown**
KeyPress	**KeyPress**
KeyUp	**KeyUp**
LostFocus	**Leave**
MouseDown	**MouseDown**
MouseMove	**MouseMove**
MouseUp	**MouseUp**
OLECompleteDrag	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
OLEDragDrop	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
OLEDragOver	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
OLEGiveFeedback	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
OLESetData	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
OLEStartDrag	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
Validate	**Validating**

See Also

Introduction to the Windows Forms CheckBox Control

Clipboard Object Changes in Visual Basic .NET

The **Clipboard** object in Visual Basic 6.0 has no direct equivalent in Visual Basic .NET. Although there is no direct mapping, the functionality of the **Clipboard** object can be duplicated using the **System.Windows.Forms.Clipboard** namespace.

During upgrade, any code that references the **Clipboard** object is not upgraded and must be rewritten. The following example shows how to modify code that uses the **Clipboard** object:

```
' Visual Basic 6.0
Clipboard.Clear
Clipboard.SetText "hello", vbCFText

If Clipboard.GetFormat(vbCFText) Then
   Text1.Text = Clipboard.GetText(vbCFText)
End If
```

This can be rewritten as:

```
' Visual Basic .NET
Dim datobj As New System.Windows.Forms.DataObject

datobj.SetData System.Windows.Forms.DataFormats.Text, "hello"
System.Windows.Forms.Clipboard.SetDataObject datobj

If System.Windows.Forms.Clipboard.GetDataObject.GetDataPresent( _
   System.Windows.Forms.DataFormats.Text) Then
   Text1.Text = System.Windows.Forms.Clipboard.GetDataObject.GetData( _
   System.Windows.Forms.DataFormats.Text)
End If
```

Clipboard Constant Equivalents

Visual Basic 6.0	Visual Basic .NET Equivalent
vbCFBitmap	**System.Windows.Forms.DataFormats.Bitmap**
vbCFDIB	**System.Windows.Forms.DataFormats.DIB**
vbCFEMetafile	**System.Windows.Forms.DataFormats.EnhancedMetafile**
vbCFFiles	**System.Windows.Forms.DataFormats.FileDrop**
vbCFLink	No equivalent. For more information, see Dynamic Data Exchange Changes in Visual Basic .NET.
vbCFMetafile	**System.Windows.Forms.DataFormats.MetafilePict**
vbCFPalette	**System.Windows.Forms.DataFormats.Palette**
vbCFRTF	**System.Windows.Forms.DataFormats.Rtf**
vbCFText	**System.Windows.Forms.DataFormats.Text**

Note In addition to the formats listed above, Visual Basic .NET supports several clipboard formats that were not supported in Visual Basic 6.0.

See Also

Drag-and-Drop Operations and Clipboard Support | DataObject Class

ComboBox Control Changes in Visual Basic .NET

The **ComboBox** control in Visual Basic 6.0 is replaced by the Windows Forms **ComboBox** control in Visual Basic .NET. The names of some properties, methods, events, and constants are different, and in some cases there are differences in behavior.

The following tables list Visual Basic 6.0 properties, methods, and events and their Visual Basic .NET equivalents. Where applicable, constants are indented beneath the property or method. All Visual Basic .NET constants map to the **System.Windows.Forms** namespace unless otherwise noted.

Links are provided as necessary to topics explaining differences in behavior. Where there is no direct equivalent in Visual Basic .NET, links are provided to topics that present alternatives.

ComboBox Properties

Visual Basic 6.0	Visual Basic .NET Equivalent
Appearance	No equivalent. For more information, see Appearance and BorderStyle Property Changes in Visual Basic .NET.
BackColor	**BackColor** **Note** Colors are handled differently in Visual Basic .NET. For more information, see Color Behavior Changes in Visual Basic .NET. For a list of constants, see Color Changes in Visual Basic .NET.
CausesValidation	**CausesValidation**
Container	**Parent**
DataChanged	No equivalent. For more information, see Data Changes in Visual Basic .NET.
DataField	No equivalent. For more information, see Data Changes in Visual Basic .NET.
DataFormat	No equivalent. For more information, see Data Changes in Visual Basic .NET.
DataMember	No equivalent. For more information, see Data Changes in Visual Basic .NET.
DataSource	No equivalent. For more information, see Data Changes in Visual Basic .NET.
DragIcon	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
DragMode	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
Enabled	**Enabled**
Font	**Font** **Note** Fonts are handled differently in Visual Basic .NET. For more information, see Font Changes in Visual Basic .NET.
FontBold	**Font** **Note** Fonts are handled differently in Visual Basic .NET. For more information, see Font Changes in Visual Basic .NET.
FontItalic	**Font** **Note** Fonts are handled differently in Visual Basic .NET. For more information, see Font Changes in Visual Basic .NET.
FontName	**Font** **Note** Fonts are handled differently in Visual Basic .NET. For more information, see Font Changes in Visual Basic .NET.
FontSize	**Font** **Note** Fonts are handled differently in Visual Basic .NET. For more information, see Font Changes in Visual Basic .NET.
FontStrikethru	**Font** **Note** Fonts are handled differently in Visual Basic .NET. For more information, see Font Changes in Visual Basic .NET.

FontUnderline	**Font** **Note** Fonts are handled differently in Visual Basic .NET. For more information, see Font Changes in Visual Basic .NET.
ForeColor	**ForeColor** **Note** Colors are handled differently in Visual Basic .NET. For more information, see Color Behavior Changes in Visual Basic .NET. For a list of constants, see Color Changes in Visual Basic .NET.
Height	**Height, Size** **Note** Coordinates are handled differently in Visual Basic .NET. For more information, see Coordinate System Changes in Visual Basic .NET.
HelpContextID	No equivalent. For more information, see Help Support Changes in Visual Basic .NET.
HWnd	**Handle**
Index	No equivalent. For more information, see Control Array Changes in Visual Basic .NET.
IntegralHeight	**IntegralHeight**
ItemData	No equivalent. For more information, see ItemData property cannot be upgraded.
Left	**Left** **Note** Coordinates are handled differently in Visual Basic .NET. For more information, see Coordinate System Changes in Visual Basic .NET.
List	**Items**
ListCount	**Items.Count**
ListIndex	**SelectedIndex**
Locked	No equivalent. For more information, see Locked Property Changes in Visual Basic .NET.
MouseIcon	No equivalent. For more information, see Cannot set a custom MousePointer.
MousePointer	**Cursor** For a list of constants, see MousePointer Changes in Visual Basic .NET.
Name	**Name**.
NewIndex	No equivalent. For more information, see NewIndex property cannot be upgraded.
OLEDragMode	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
OLEDropMode	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
Parent	**FindForm** method
RightToLeft **True** **False**	**RightToLeft** **RightToLeft.Yes** **RightToLeft.No**
SelLength	**SelectionLength**
SelStart	**SelectionStart**
SelText	**SelectedText**
Sorted	**Sorted**

Style **0 - DropDown Combo** **1 - Simple Combo** **2 - DropDown List**	**DropDownStyle** **ComboBoxStyle.DropDown** **ComboBoxStyle.Simple** **ComboBoxStyle.DropDownList**
TabIndex	**TabIndex**
TabStop	**TabStop**
Tag	**Tag**
Text	**Text**
ToolTipText	**ToolTip** component For more information, see ToolTip Changes in Visual Basic .NET.
Top	**Top** **Note** Coordinates are handled differently in Visual Basic .NET. For more information, see Coordinate System Changes in Visual Basic .NET.
TopIndex	No equivalent. For more information, see TopIndex Property and Scroll Event Changes in Visual Basic .NET.
Visible	**Visible**
WhatsThisHelpID	No equivalent. For more information, see Help Support Changes in Visual Basic .NET.
Width	**Width, Size** **Note** Coordinates are handled differently in Visual Basic .NET. For more information, see Coordinate System Changes in Visual Basic .NET.

ComboBox Methods

Visual Basic 6.0	Visual Basic .NET Equivalent
AddItem	**Items.Insert** or **Items.Add**
Clear	**Items.Clear**
Drag	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
Move	**SetBounds** **Note** Coordinates are handled differently in Visual Basic .NET. For more information, see Coordinate System Changes in Visual Basic .NET.
OLEDrag	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
Refresh	**Refresh**
RemoveItem	**Items.Remove**
SetFocus	**Focus**
ShowWhatsThis	No equivalent. For more information, see Help Support Changes in Visual Basic .NET.
ZOrder **0 – vbBringToFront** **1 - vbSendToBack**	**BringToFront()** or **SendToBack()** functions **BringToFront()** **SendToBack()**

ComboBox Events

Visual Basic 6.0	Visual Basic .NET Equivalent

Change	**TextChanged** **Note** The behavior of the **TextChanged** event is slightly different. For more information, see Change Event Changes in Visual Basic .NET.
Click	**SelectedIndexChanged**
DblClick	No equivalent. Use the **SelectedIndexChanged** or **TextChanged** events.
DragDrop	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
DragOver	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
DropDown	**DropDown**
GotFocus	**Enter**
KeyDown	**KeyDown**
KeyPress	**KeyPress**
KeyUp	**KeyUp**
LostFocus	**Leave**
OLECompleteDrag	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
OLEDragDrop	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
OLEDragOver	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
OLEGiveFeedback	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
OLESetData	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
OLEStartDrag	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
Scroll	No equivalent. For more information, see TopIndex Property and Scroll Event Changes in Visual Basic .NET.
Validate	**Validating**

See Also

Introduction to the Windows Forms ComboBox Control | Change Event Changes in Visual Basic .NET

CommandButton Control Changes in Visual Basic .NET

The **CommandButton** control in Visual Basic 6.0 is replaced by the Windows Forms **Button** control in Visual Basic .NET. The names of some properties, methods, events, and constants are different, and in some cases there are differences in behavior.

The following tables list Visual Basic 6.0 properties, methods, and events and their Visual Basic .NET equivalents. Where applicable, constants are indented beneath the property or method. All Visual Basic .NET constants map to the **System.Windows.Forms** namespace unless otherwise noted.

Links are provided as necessary to topics explaining differences in behavior. Where there is no direct equivalent in Visual Basic .NET, links are provided to topics that present alternatives.

CommandButton Properties

Visual Basic 6.0	Visual Basic .NET Equivalent
Appearance	No equivalent. For more information, see Appearance and BorderStyle Property Changes in Visual Basic .NET.
BackColor	**BackColor** **Note** Colors are handled differently in Visual Basic .NET. For more information, see Color Behavior Changes in Visual Basic .NET. For a list of constants, see Color Changes in Visual Basic .NET.
Cancel	**Form.CancelButton**
Caption	**Text**
CausesValidation	**CausesValidation**
Container	**Parent**
Default	**Form.AcceptButton**
DisabledPicture	No equivalent. For more information, see Style Property Changes in Visual Basic .NET.
DownPicture	No equivalent. For more information, see Style Property Changes in Visual Basic .NET.
DragIcon	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
DragMode	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
Enabled	**Enabled**
Font	**Font** **Note** Fonts are handled differently in Visual Basic .NET. For more information, see Font Changes in Visual Basic .NET.
FontBold	**Font** **Note** Fonts are handled differently in Visual Basic .NET. For more information, see Font Changes in Visual Basic .NET.
FontItalic	**Font** **Note** Fonts are handled differently in Visual Basic .NET. For more information, see Font Changes in Visual Basic .NET.
FontName	**Font** **Note** Fonts are handled differently in Visual Basic .NET. For more information, see Font Changes in Visual Basic .NET.
FontSize	**Font** **Note** Fonts are handled differently in Visual Basic .NET. For more information, see Font Changes in Visual Basic .NET.

FontStrikethru	**Font** **Note** Fonts are handled differently in Visual Basic .NET. For more information, see Font Changes in Visual Basic .NET.
FontUnderline	**Font** **Note** Fonts are handled differently in Visual Basic .NET. For more information, see Font Changes in Visual Basic .NET.
Height	**Height, Size** **Note** Coordinates are handled differently in Visual Basic .NET. For more information, see Coordinate System Changes in Visual Basic .NET.
HelpContextID	No equivalent. For more information, see Help Support Changes in Visual Basic .NET.
HWnd	**Handle**
Index	No equivalent. For more information, see Control Array Changes in Visual Basic .NET.
Left	**Left** **Note** Coordinates are handled differently in Visual Basic .NET. For more information, see Coordinate System Changes in Visual Basic .NET.
MaskColor	No equivalent. For more information, see MaskColor Changes in Visual Basic .NET.
MouseIcon	No equivalent. For more information, see Cannot set a custom MousePointer.
MousePointer	**Cursor** For a list of constants, see MousePointer Changes in Visual Basic .NET.
Name	**Name**
OLEDropMode	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
Parent	**FindForm** method
Picture	**Image**
RightToLeft **True** **False**	**RightToLeft** **RightToLeft.Yes** **RightToLeft.No**
Style	**No equivalent.** For more information, see Style Property Changes in Visual Basic .NET.
TabIndex	**TabIndex**
TabStop	**TabStop**
Tag	**Tag**
ToolTipText	**ToolTip** component For more information, see ToolTip Changes in Visual Basic .NET.
Top	**Top** **Note** Coordinates are handled differently in Visual Basic .NET. For more information, see Coordinate System Changes in Visual Basic .NET.
UseMaskColor	No equivalent. For more information, see MaskColor Changes in Visual Basic .NET.
Value	No equivalent. The **PerformClick** method is the same as setting the **Value** to **true**; for any other use of the **Value** property there is no equivalent.
Visible	**Visible**
WhatsThisHelpID	No equivalent. For more information, see Help Support Changes in Visual Basic .NET.

Width	**Width, Size** **Note** Coordinates are handled differently in Visual Basic .NET. For more information, see Coordinate System Changes in Visual Basic .NET.

CommandButton Methods

Visual Basic 6.0	Visual Basic .NET Equivalent
Drag	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
Move	**SetBounds** **Note** Coordinates are handled differently in Visual Basic .NET. For more information, see Coordinate System Changes in Visual Basic .NET.
OLEDrag	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
Refresh	**Refresh**
SetFocus	**Focus**
ShowWhatsThis	No equivalent. For more information, see Help Support Changes in Visual Basic .NET.
ZOrder **0 – vbBringToFront** **1 - vbSendToBack**	**BringToFront()** or **SendToBack()** functions **BringToFront()** **SendToBack()**

CommandButton Events

Visual Basic 6.0	Visual Basic .NET Equivalent
Click	**Click**
DragDrop	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
DragOver	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
GotFocus	**Enter**
KeyDown	**KeyDown**
KeyPress	**KeyPress**
KeyUp	**KeyUp**
LostFocus	**Leave**
MouseDown	**MouseDown**
MouseMove	**MouseMove**
MouseUp	**MouseUp**
OLECompleteDrag	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
OLEDragDrop	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
OLEDragOver	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
OLEGiveFeedback	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
OLESetData	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
OLEStartDrag	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.

See Also

Introduction to the Windows Forms Button Control

Controls Collection Changes in Visual Basic .NET

In Visual Basic 6.0, the **Controls** collection was a collection whose elements represented the controls on a form or container control. In Visual Basic .NET, the **System.Windows.Forms.Control.ControlCollection** class replaces the **Controls** collection. Forms have a default **ControlCollection** class that can be accessed using the syntax `Me.Controls`.

There are some differences in behavior:

- In Visual Basic 6.0, the **Add** method of the **Controls** collection was late-bound; controls were created in the **Add** method by specifying the control class as an argument. In Visual Basic .NET. the **Add** method of a **Controls** collection class requires that the controls have already been created before being added to the collection.

  > **Note** Because of this and other differences between Visual Basic 6.0 and Visual Basic .NET Controls collections, calls to the **Add** method are not upgraded. You must add code to recreate your application's behavior using the new **Controls.Add** method.

- In Visual Basic 6.0, the **Timer** and **Menu** controls were members of the **Controls** collection. In Visual Basic .NET they are replaced by the **Timer** and **MainMenu** components; components are not members of the **ControlCollection** class.
- The **Controls** collection included controls that were children of a container control (for example, controls sited on a **Frame** control); the **ControlCollection** class does not.
- Members of the **Controls** collection could be referenced using either the index or the name of the control; the **ControlCollection** class only supports the index.
- The **Remove** method of the **Controls** collection could only be used for controls that had been added using the **Add** method; the **ControlCollection** class does not have this restriction.

The following example illustrates some of the differences:

```
' Visual Basic 6.0
Private Sub Command1_Click()
    Dim c As Control
    Set c = Controls.Add("VB.TextBox", "Text1")
    c.Visible = True
    c.Text = "Hello"
    If Controls.Count > 1 Then
        MsgBox (Controls("Text1").Text)
    End If
    ' The following line causes a compilation error.
    Controls.Remove (Command1)
End Sub

' Visual Basic .NET
Private Sub button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles
button1.Click
    Dim textBox1 As New System.Windows.Forms.TextBox
    textBox1.Text = "Hello"
    Me.Controls.Add(textBox1)
    If Me.Controls.Count > 1 Then
        MsgBox(Me.Controls(Me.Controls.IndexOf(textBox1)).Text)
    End If
    Me.Controls.Remove(button1)
End Sub
```

See Also

Control.Controls Property | Form.ControlCollection Class

Data Control Changes in Visual Basic .NET

The Visual Basic 6.0 Data Control was used as a mechanism for binding controls to a database using DAO. Visual Basic .NET has no equivalent for the Data Control; the data-binding architecture has changed and DAO is no longer supported. For more information, see Data Changes in Visual Basic .NET.

See Also

Things to Consider Before Upgrading | Data Changes in Visual Basic .NET

DataGrid Control Changes in Visual Basic .NET

The **DataGrid** control in Visual Basic 6.0 is replaced by the Windows Forms **DataGrid** control in Visual Basic .NET. The names of some properties, methods, events, and constants are different, and in some cases there are differences in behavior. The following table lists Visual Basic 6.0 properties and their Visual Basic .NET equivalents.

The Visual Basic .NET **DataGrid** control does not need data-specific methods or events because all actions are performed through the data source. Because of this separation of presentation and data functionality, the data source can be changed with or without user interface input. Also, multiple controls bound to the same data source will always stay in sync.

Properties for viewing and navigation in the **DataGrid** (such as **TabAction**, **EnterAction**, **AllowArrows**, **WrapCellPointer**, and **Scrollable**) are no longer needed. For example, the grid functions as though the **Scrollable** property is set to **true**: If more data exists than can be displayed, a scroll bar appears automatically. Excel-style navigation through the grid is the default, allowing the user to move forward with the TAB key and backward with the SHIFT+TAB key combination. For details, see Keyboard Shortcuts for the Windows Forms DataGrid Control.

DataGrid Properties

Visual Basic 6.0	**Visual Basic .NET Equivalent**
AddNewMode	No equivalent.
Align	**Dock** and **Anchor**
AllowAddNew	On **DataView** object; accessed through **AllowNew** property of **DataView** class.
AllowArrows	No longer an adjustable property: arrow navigation always allowed.
AllowDelete	On **DataView** object; accessed through **AllowDelete** property of **DataView** class.
AllowRowSizing	No longer an adjustable property: row sizing always allowed.
AllowUpdate	On **DataView** object; accessed through **AllowEdit** property of **DataView** class.
Appearance	**BorderStyle**
ApproxCount	On **BindingManagerBase** object, returned by **BindingContext** object; accessed through **Count** property of the **BindingManagerBase** class.
BackColor	**BackColor** affects only odd-numbered rows; **AlternatingBackColor** affects only even-numbered rows. **Note** Colors are handled differently in Visual Basic .NET. For more information, see Color Behavior Changes in Visual Basic .NET. For a list of constants, see Color Changes in Visual Basic .NET.
Bookmark	You can now directly access any item.
Caption	**CaptionText**
CausesValidation	**CausesValidation**
Col	On the **DataGridCell** structure; accessed by **DataGrid.CurrentCell.ColumnNumber**
ColumnHeaders	**ColumnHeadersVisible** **RowHeadersVisible**
Columns	On the **DataGridTableStyle** object; accessed by **DataGrid.TableStyle.GridColumnStyles**.
Container	**Parent**; inherited from **Control**.
CurrentCellModified	**RowState** property on **DataRow** object.

CurrentCellVisible	If **CurrentCell** properties are set to a cell that is not visible, the grid will scroll to show that cell (similar to **CurrentCellVisible** = **true**).
DataBindings	**DataBindings**
DataChanged	On **DataTable** object; accessed by **GetChanges** method of **DataTable** class.
DataFormats	**Format** property on **DataGridTextBoxColumn** object.
DataMember	**DataMember**
DataSource	**DataSource**
DefColWidth	**PreferredColumnWidth**
DragIcon	See GiveFeedbackEventArgs Class.
DragMode	See Control.DoDragDrop Method.
EditActive	No equivalent.
Enabled	**Enabled**
ErrorText	On **DataRow** object; accessed by **RowError** property of **DataRow** class.
FirstRow	Set **Position** property to 0 on **BindingManagerBase** object
Font	**Font** **Note** Fonts are handled differently in Visual Basic .NET. For more information, see Font Changes in Visual Basic .NET
ForeColor	**ForeColor** **Note** Colors are handled differently in Visual Basic .NET. For more information, see Color Behavior Changes in Visual Basic .NET. For a list of constants, see Color Changes in Visual Basic .NET.
HeadFont	**HeaderFont** **Note** Fonts are handled differently in Visual Basic .NET. For more information, see Font Changes in Visual Basic .NET
HeadLines	No equivalent.
Height	**Height**, inherited from **Control** class **Note** Coordinates are handled differently in Visual Basic .NET. For more information, see Coordinate System Changes in Visual Basic .NET.
HelpContextID	On **HelpProvider** extender control; accessed by **SetHelpKeyword** method of **HelpProvider** control.
hWnd	**Handle**
hWndEditor	No equivalent.
Index	No equivalent.
Left	**Left**
LeftCol	No equivalent.
MarqueeStyle	Cell is highlighted when selected
Name	Not available at run time

Object	No equivalent.
Parent	**Parent**, inherited from **Control** class
RecordSelectors	**RowHeadersVisible**
RightToLeft	**RightToLeft**
Row	On **DataGridCell** structure; accessed by **DataGrid.CurrentCell.RowNumber**.
RowDividerStyle	**GridLineStyle**
RowHeight	No equivalent.
ScrollBars	No longer an adjustable property; the default behavior is **ScrollBars** = **true**.
SelBookmarks	No equivalent.
SelEndCol	No equivalent.
SelLength	No equivalent.
SelStart	No equivalent.
SelStartCol	No equivalent.
SelText	No equivalent.
Split	No equivalent.
Splits	No equivalent.
TabAcrossSplits	No equivalent.
TabAction	No longer an adjustable property. The default behavior is for the focus to move to the next column when the TAB key is pressed (**dbgColumnNavigation**). You can move the focus to the next control on the form (**dbgControlNavigation**) by pressing CTRL+TAB.
TabIndex	**TabIndex**
TabStop	**TabStop**
Tag	No equivalent.
Text	**Item**, when set to the current cell. For example: `Dim SelectedCellText As String = DataGrid1.Item(DataGrid1.CurrentCell)`
ToolTipText	On **ToolTip** extender control; accessed by **SetToolTip** method of **ToolTip** control
Top	**Top**
Visible	**Visible**
VisibleCols	**VisibleColumnCount**
VisibleRows	**VisibleRowCount**
WhatsThisHelpID	On **HelpProvider** extender control; accessed by **SetHelpString** method of **HelpProvider** control
Width	**Width**, inherited from **Control** class **Note** Coordinates are handled differently in Visual Basic .NET. For more information, see Coordinate System Changes in Visual Basic .NET.
WrapCellPointer	No longer an adjustable property; default behavior is **WrapCellPointer** = **true**.

See Also

Keyboard Shortcuts for the Windows Forms DataGrid Control | Introduction to the Windows Forms DataGrid Control | What's New in Windows Forms and Controls | DataGrid Control (Windows Forms) | Controls and Programmable Objects Compared in Different Languages and Libraries | Data Changes in Visual Basic .NET | Control Changes in Visual Basic .NET | Windows Forms Data Architecture

DirListBox Control Changes in Visual Basic .NET

The Visual Basic 6.0 **DirListBox** control was typically used to display directories and paths in a File Open or Save dialog box. Visual Basic .NET has no equivalent for the **DirListBox** control; however, the Windows Forms **OpenFileDialog** and **SaveFileDialog** controls provide the ability to create standard Windows dialog boxes for working with files.

During upgrade, any existing **DirListBox** controls are upgraded to the **VB6.DirListBox** control that is provided as a part of the compatibility library (Microsoft.VisualBasic.Compatibility).

See Also

Introduction to the Windows Forms OpenFileDialog Component | Introduction to the Windows Forms SaveFileDialog Component

DriveListBox Control Changes in Visual Basic .NET

The Visual Basic 6.0 **DriveListBox** control was typically used to select or change drives in a File Open or Save dialog box. Visual Basic .NET has no equivalent for the **DriveListBox** control; however, the Windows Forms **OpenFileDialog** and **SaveFileDialog** controls provide the ability to create standard Windows dialog boxes for working with files.

During upgrade, any existing **DriveListBox** controls are upgraded to the **VB6.DriveListBox** control that is provided as a part of the compatibility library (Microsoft.VisualBasic.Compatibility).

See Also

Introduction to the Windows Forms OpenFileDialog Component | Introduction to the Windows Forms SaveFileDialog Component

FileListBox Control Changes in Visual Basic .NET

The Visual Basic 6.0 **FileListBox** control was typically used to select files in a File Open or Save dialog box. Visual Basic .NET has no equivalent for the **FileListBox** control; however, the Windows Forms **OpenFileDialog** and **SaveFileDialog** controls provide the ability to create standard Windows dialog boxes for working with files.

During upgrade, any existing **FileListBox** controls are upgraded to the **VB6.FileListBox** control that is provided as a part of the compatibility library (Microsoft.VisualBasic.Compatibility).

See Also

Introduction to the Windows Forms OpenFileDialog Component | Introduction to the Windows Forms SaveFileDialog Component

Font Object Changes in Visual Basic .NET

In Visual Basic 6.0, **Font** properties could be set via a **stdFont** object rather than setting the properties directly. The Visual Basic 6.0 **stdFont** object is replaced in Visual Basic .NET by the **System.Drawing.Font** class; there is no direct mapping of the properties and methods. For more information, see Font Changes in Visual Basic .NET.

See Also

Font Changes in Visual Basic .NET

Form Object Changes in Visual Basic .NET

The **Form** object in Visual Basic 6.0 is replaced by the **System.Windows.Forms.Form** class in Visual Basic .NET. The names of some properties, methods, events, and constants are different, and in some cases there are differences in behavior.

The following tables list Visual Basic 6.0 properties, methods, and events and their Visual Basic .NET equivalents. Where applicable, constants are indented beneath the property or method. All Visual Basic .NET constants map to the **System.Windows.Forms** namespace unless otherwise noted.

Links are provided as necessary to topics explaining differences in behavior. Where there is no direct equivalent in Visual Basic .NET, links are provided to topics that present alternatives.

Form Properties

Visual Basic 6.0	Visual Basic .NET Equivalent
ActiveControl	**ActiveControl**
Appearance	No equivalent. For more information, see Appearance and BorderStyle Property Changes in Visual Basic .NET.
AutoRedraw	No equivalent. For more information, see Graphics Changes in Visual Basic .NET.
BackColor	**BackColor** **Note** Colors are handled differently in Visual Basic .NET. For more information, see Color Behavior Changes in Visual Basic .NET. For a list of constants, see Color Changes in Visual Basic .NET.
BorderStyle **0 – None** **1 – Fixed Single** **2 – Sizable** **3 – Fixed Dialog** **4 – Fixed Tool Window** **5 – Sizable ToolWindow**	**FormBorderStyle** **0 – FormBorderStyle.None** **1 – FormBorderStyle.FixedSingle** **4 – FormBorderStyle.Sizable** **3 – FormBorderStyle.FixedDialog** **5 – FormBorderStyle.FixedToolWindow** **6 – FormBorderStyle.SizableToolWindow**
Caption	**Text**
ClipControls	No equivalent. For more information, see Graphics Changes in Visual Basic .NET.
ControlBox	**ControlBox**
Controls	**Controls** **Note** There are differences in the **Controls** collection in Visual Basic .NET. For more information, see Controls Collection Changes in Visual Basic .NET.
Count	**Controls.Count** **Note** There are differences in the **Controls** collection in Visual Basic .NET. For more information, see Controls Collection Changes in Visual Basic .NET.
CurrentX	No equivalent. For more information, see Graphics Changes in Visual Basic .NET.
CurrentY	No equivalent. For more information, see Graphics Changes in Visual Basic .NET.
DrawMode	No equivalent. For more information, see Graphics Changes in Visual Basic .NET.
DrawStyle	No equivalent. For more information, see Graphics Changes in Visual Basic .NET.
DrawWidth	No equivalent. For more information, see Graphics Changes in Visual Basic .NET.
Enabled	**Enabled**
FillColor	No equivalent. For more information, see Graphics Changes in Visual Basic .NET.
FillStyle	No equivalent. For more information, see Graphics Changes in Visual Basic .NET.

Font	**Font** **Note** Fonts are handled differently in Visual Basic .NET. For more information, see Font Changes in Visual Basic .NET.
FontBold	**Font** **Note** Fonts are handled differently in Visual Basic .NET. For more information, see Font Changes in Visual Basic .NET.
FontItalic	**Font** **Note** Fonts are handled differently in Visual Basic .NET. For more information, see Font Changes in Visual Basic .NET.
FontName	**Font** **Note** Fonts are handled differently in Visual Basic .NET. For more information, see Font Changes in Visual Basic .NET.
FontSize	**Font** **Note** Fonts are handled differently in Visual Basic .NET. For more information, see Font Changes in Visual Basic .NET.
FontStrikethru	**Font** **Note** Fonts are handled differently in Visual Basic .NET. For more information, see Font Changes in Visual Basic .NET.
FontUnderline	**Font** **Note** Fonts are handled differently in Visual Basic .NET. For more information, see Font Changes in Visual Basic .NET.
ForeColor	**ForeColor** **Note** Colors are handled differently in Visual Basic .NET. For more information, see Color Behavior Changes in Visual Basic .NET. For a list of constants, see Color Changes in Visual Basic .NET.
HasDC	No equivalent. For more information, see Graphics Changes in Visual Basic .NET.
HDC	No equivalent. For more information, see Graphics Changes in Visual Basic .NET.
Height	**Height, Size** **Note** Coordinates are handled differently in Visual Basic .NET. For more information, see Coordinate System Changes in Visual Basic .NET.
HelpContextID	No equivalent. For more information, see Help Support Changes in Visual Basic .NET.
HWnd	**Handle**
Icon	**Icon**
Image	No equivalent. The **Image** property in Visual basic 6.0 returned a handle to a bitmap; images in Visual Basic .NET do not have handles.
KeyPreview	**KeyPreview**
Left	**Left** **Note** Coordinates are handled differently in Visual Basic .NET. For more information, see Coordinate System Changes in Visual Basic .NET.
LinkMode	No equivalent. For more information, see Dynamic Data Exchange Changes in Visual Basic .NET.
LinkTopic	No equivalent. For more information, see Dynamic Data Exchange Changes in Visual Basic .NET.
MaxButton	**MaximizeBox**

MDIChild	**MDIParent** **Note** MDI behavior is different in Visual Basic .NET. For more information, see MDI Changes in Visual Basic .NET.
MinButton	**MinimizeBox**
MouseIcon	No equivalent. For more information, see Cannot set a custom MousePointer.
MousePointer	**Cursor** For a list of constants, see MousePointer Changes in Visual Basic .NET.
Moveable	No equivalent. For more information, see Moveable Property Changes in Visual Basic .NET.
Name	**Name**
NegotiateMenus	No equivalent. For more information, see Menu Changes in Visual Basic .NET.
OLEDropMode	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
Palette	No equivalent. For more information, see Palette Changes in Visual Basic .NET.
PaletteMode	No equivalent. For more information, see Palette Changes in Visual Basic .NET.
Picture	**BackgroundImage** **Note** In Visual Basic 6.0, the **Picture** was displayed in the top left corner of the form; in Visual Basic .NET, the **BackgroundImage** is tiled.
RightToLeft **True** **False**	**RightToLeft** **RightToLeft.Yes** **RightToLeft.No**
ScaleHeight	No equivalent. For more information, see Coordinate System Changes in Visual Basic .NET.
ScaleLeft	No equivalent. For more information, see Coordinate System Changes in Visual Basic .NET.
ScaleMode	No equivalent. For more information, see Coordinate System Changes in Visual Basic .NET.
ScaleTop	No equivalent. For more information, see Coordinate System Changes in Visual Basic .NET.
ScaleWidth	No equivalent. For more information, see Coordinate System Changes in Visual Basic .NET.
ShowInTaskbar	**ShowInTaskBar**
StartUpPosition **0 – Manual** **1 – CenterOwner** **2 – CenterScreen** **3 – Windows Default**	**StartPositionMapping** **0 – FormStartPosition.Manual** **4 – FormStartPosition.CenterParent** **1 – FormStartPosition.CenterScreen** **2 – FormStartPosition.WindowsDefaultLocation**
Tag	**Tag**
Top	**Top** **Note** Coordinates are handled differently in Visual Basic .NET. For more information, see Coordinate System Changes in Visual Basic .NET.
Visible	**Visible**
WhatsThisButton	**HelpButton** **Note** There are differences in Help behavior in Visual Basic .NET. For more information, see Help Support Changes in Visual Basic .NET.
WhatsThisHelp	No equivalent. For more information, see Help Support Changes in Visual Basic .NET.
Width	**Width, Size** **Note** Coordinates are handled differently in Visual Basic .NET. For more information, see Coordinate System Changes in Visual Basic .NET.

WindowState	**WindowState**

Form Methods

Visual Basic 6.0	**Visual Basic .NET Equivalent**
Circle	No equivalent. For more information, see Graphics Changes in Visual Basic .NET.
Cls	No equivalent. For more information, see Graphics Changes in Visual Basic .NET.
Hide	**Hide**
Line	No equivalent. For more information, see Graphics Changes in Visual Basic .NET.
Move	**SetBounds** **Note** Coordinates are handled differently in Visual Basic .NET. For more information, see Coordinate System Changes in Visual Basic .NET.
OLEDrag	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
PaintPicture	No equivalent. For more information, see Graphics Changes in Visual Basic .NET.
Point	No equivalent. For more information, see Graphics Changes in Visual Basic .NET.
PopupMenu	No equivalent. For more information, see Menu Changes in Visual Basic .NET.
PrintForm	No equivalent. For more information, see Printing Changes in Visual Basic .NET.
Pset	No equivalent. For more information, see Graphics Changes in Visual Basic .NET.
Refresh	**Refresh**
Scale	No equivalent. For more information, see Coordinate System Changes in Visual Basic .NET.
ScaleX	No equivalent. For more information, see Coordinate System Changes in Visual Basic .NET.
ScaleY	No equivalent. For more information, see Coordinate System Changes in Visual Basic .NET.
SetFocus	**Activate**
Show	**Show** or **ShowDialog**
TextHeight	No equivalent. For more information, see Graphics Changes in Visual Basic .NET.
TextWidth	No equivalent. For more information, see Graphics Changes in Visual Basic .NET.
ValidateControls	**Validate** **Note** The **Validate** method returns **true** or **false**; **ValidateControls** raised an error if validation failed .
WhatsThisMode	No equivalent. For more information, see Help Support Changes in Visual Basic .NET.
ZOrder **0 – vbBringToFront** **1 - vbSendToBack**	**BringToFront()** or **SendToBack()** functions **Note** In Visual Basic 6.0, the **ZOrder** method positioned the form relative to other forms in the applicat ion; in Visual Basic .NET, forms are positioned relative to all windows on the screen. **BringToFront()** **SendToBack()**

Form Events

Visual Basic 6.0	**Visual Basic .NET Equivalent**
Activate	**Activated** **Note** In Visual Basic 6.0, the **Activate** event was raised only when switching between forms in the applicat ion; in Visual Basic .NET, the **Activated** event also is raised when switching from other applications.
Click	**Click** **Note** The **Click** event is not supported for MDI Forms.

DblClick	**DoubleClick**
Deactivate	**Deactivate** **Note** In Visual Basic 6.0, the **Deactivate** event was raised only when switching between forms in the application; in Visual Basic .NET, it also is raised when switching to other applications.
DragDrop	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
DragOver	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
GotFocus	**Enter**
Initialize	**New** method **Note** The behavior of the **New** method is different than the **Initialize** event. For more information, see Form Event Changes in Visual Basic .NET.
KeyDown	**KeyDown**
KeyPress	**KeyPress**
KeyUp	**KeyUp**
LinkClose	No equivalent. For more information, see Dynamic Data Exchange Changes in Visual Basic .NET.
LinkError	No equivalent. For more information, see Dynamic Data Exchange Changes in Visual Basic .NET.
LinkExecute	No equivalent. For more information, see Dynamic Data Exchange Changes in Visual Basic .NET.
LinkOpen	No equivalent. For more information, see Dynamic Data Exchange Changes in Visual Basic .NET.
Load	**Load**
LostFocus	**LostFocus**
MouseDown	**MouseDown** **Note** The **MouseDown** event is not supported for MDI Forms.
MouseMove	**MouseMove** **Note** The **MouseMove** event is not supported for MDI Forms.
MouseUp	**MouseUp** **Note** The **MouseUp** event is not supported for MDI Forms.
OLECompleteDrag	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
OLEDragDrop	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
OLEDragOver	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
OLEGiveFeedback	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
OLESetData	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
OLEStartDrag	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
Paint	**Paint**

QueryUnload	**Closing** **Note** In Visual Basic 6.0, **QueryUnload** took two arguments, Cancel and UnloadMode. In Visual Basic .NET, Cancel can be replaced by using **System.ComponentModel.CancelEventArgs.Cancel**. There is no replacement for UnloadMode.
Resize	**Resize**
Terminate	**Dispose** method **Note** The behavior of the **Dispose** method is different than the **Terminate** event. For more information, see Form Event Changes in Visual Basic .NET.
Unload	**Closed** **Note** The behavior of the **Closed** event is different than the **Unload** event. For more information, see Form Event Changes in Visual Basic .NET.

See Also

Windows Forms | Forms Task Changes in Visual Basic .NET

Forms Collection Changes in Visual Basic .NET

In Visual Basic 6.0, the **Forms** collection was a collection whose elements represented each loaded form in an application. There is no equivalent for the **Forms** collection in Visual Basic .NET.

If you need the functionality of the **Forms** collection, you can create your own collection class and add or remove references to forms as the forms are loaded or unloaded. For more information, see Walkthrough: Creating Your Own Collection Class.

See Also

Form Object Changes in Visual Basic .NET | Walkthrough: Creating Your Own Collection Class

Frame Control Changes in Visual Basic .NET

In Visual Basic 6.0, the **Frame** control was used as a container for grouping controls. In Visual Basic .NET, the **Frame** control is replaced by either the **GroupBox** control or the **Panel** control.

The **GroupBox** control is the equivalent of a **Frame** control with a **BorderStyle** property of **1 – Fixed Single**. It has a visible border and optionally a caption.

The **Panel** control is the equivalent of a **Frame** control with a **BorderStyle** property of **0 – None.** It has no border or caption.

The following tables list Visual Basic 6.0 properties, methods, and events and their Visual Basic .NET equivalents. Where applicable, constants are indented beneath the property or method. All Visual Basic .NET constants map to the **System.Windows.Forms** namespace unless otherwise noted.

Links are provided as necessary to topics explaining differences in behavior. Where there is no direct equivalent in Visual Basic .NET, links are provided to topics that present alternatives.

Frame Properties

Visual Basic 6.0	**Visual Basic .NET Equivalent**
Appearance **0 – Flat** **1 - 3D**	**FlatStyle** (**GroupBox** control only) **Note** The **Panel** control does not have a direct equivalent. For more information, see Appearance and BorderStyle Property Changes in Visual Basic .NET. **FlatStyle.Flat** **FlatStyle.Standard**
BackColor	**BackColor** **Note** Colors are handled differently in Visual Basic .NET. For more information, see Color Behavior Changes in Visual Basic .NET. For a list of constants, see Color Changes in Visual Basic .NET.
BorderStyle **0 – None** **1 – Fixed Single**	**BorderStyle** (**GroupBox** control only) **Note** The **Panel** control does not have a direct equivalent. For more information, see Appearance and BorderStyle Property Changes in Visual Basic .NET. **0 – BorderStyle.None** **1 – BorderStyle.FixedSingle**
Caption	**Text** (**GroupBox** control only) **Note** The **Panel** control does not have a **Text** property.
ClipControls	No equivalent. For more information, see Graphics Changes in Visual Basic .NET.
Container	**Parent**
DragIcon	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
DragMode	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
Enabled	**Enabled**
Font	**Font** **Note** Fonts are handled differently in Visual Basic .NET. For more information, see Font Changes in Visual Basic .NET.
FontBold	**Font** **Note** Fonts are handled differently in Visual Basic .NET. For more information, see Font Changes in Visual Basic .NET.

FontItalic	**Font** **Note** Fonts are handled differently in Visual Basic .NET. For more information, see Font Changes in Visual Basic .NET.
FontName	**Font** **Note** Fonts are handled differently in Visual Basic .NET. For more information, see Font Changes in Visual Basic .NET.
FontSize	**Font** **Note** Fonts are handled differently in Visual Basic .NET. For more information, see Font Changes in Visual Basic .NET.
FontStrikethru	**Font** **Note** Fonts are handled differently in Visual Basic .NET. For more information, see Font Changes in Visual Basic .NET.
FontUnderline	**Font** **Note** Fonts are handled differently in Visual Basic .NET. For more information, see Font Changes in Visual Basic .NET.
ForeColor	**ForeColor** **Note** Colors are handled differently in Visual Basic .NET. For more information, see Color Behavior Changes in Visual Basic .NET. For a list of constants, see Color Changes in Visual Basic .NET.
Height	**Height, Size** **Note** Coordinates are handled differently in Visual Basic .NET. For more information, see Coordinate System Changes in Visual Basic .NET.
HelpContextID	No equivalent. For more information, see Help Support Changes in Visual Basic .NET.
HWnd	**Handle**
Index	No equivalent. For more information, see Control Array Changes in Visual Basic .NET.
Left	**Left** **Note** Coordinates are handled differently in Visual Basic .NET. For more information, see Coordinate System Changes in Visual Basic .NET.
MouseIcon	No equivalent. For more information, see Cannot set a custom MousePointer.
MousePointer	**Cursor** For a list of constants, see MousePointer Changes in Visual Basic .NET.
Name	**Name**
OLEDropMode	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
Parent	**FindForm** method
RightToLeft **True** **False**	**RightToLeft** **RightToLeft.Yes** **RightToLeft.No**
TabIndex	**TabIndex**
Tag	**Tag**
ToolTipText	**ToolTip** component For more information, see ToolTip Changes in Visual Basic .NET.

Top	**Top** **Note** Coordinates are handled differently in Visual Basic .NET. For more information, see Coordinate System Changes in Visual Basic .NET.
Visible	**Visible**
WhatsThisHelpID	No equivalent. For more information, see Help Support Changes in Visual Basic .NET.
Width	**Width, Size** **Note** Coordinates are handled differently in Visual Basic .NET. For more information, see Coordinate System Changes in Visual Basic .NET.

Frame Methods

Visual Basic 6.0	Visual Basic .NET Equivalent
Drag	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
Move	**SetBounds** **Note** Coordinates are handled differently in Visual Basic .NET. For more information, see Coordinate System Changes in Visual Basic .NET.
OLEDrag	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
Refresh	**Refresh**
ShowWhatsThis	No equivalent. For more information, see Help Support Changes in Visual Basic .NET.
ZOrder **0 – vbBringToFront** **1 - vbSendToBack**	**BringToFront()** or **SendToBack()** functions **BringToFront()** **SendToBack()**

Frame Events

Visual Basic 6.0	Visual Basic .NET Equivalent
Click	**Click** (**Panel** only, **GroupBox** has no equivalent)
DblClick	**DoubleClick** (**Panel** only, **GroupBox** has no equivalent)
DragDrop	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
DragOver	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
MouseDown	**MouseDown**
MouseMove	**MouseMove**
MouseUp	**MouseUp**
OLECompleteDrag	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
OLEDragDrop	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
OLEDragOver	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
OLEGiveFeedback	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
OLESetData	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
OLEStartDrag	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.

See Also

Introduction to the Windows Forms Panel Control | Introduction to the Windows Forms GroupBox Control

HScrollBar Control Changes in Visual Basic .NET

The **HScrollBar** control in Visual Basic 6.0 is replaced by the Windows Forms **HScrollBar** control in Visual Basic .NET. The names of some properties, methods, events, and constants are different, and in some cases there are differences in behavior.

The following tables list Visual Basic 6.0 properties, methods, and events and their Visual Basic .NET equivalents. Where applicable, constants are indented beneath the property or method. All Visual Basic .NET constants map to the **System.Windows.Forms** namespace unless otherwise noted.

Links are provided as necessary to topics explaining differences in behavior. Where there is no direct equivalent in Visual Basic .NET, links are provided to topics that present alternatives.

HScrollBar Properties

Visual Basic 6.0	**Visual Basic .NET Equivalent**
CausesValidation	**CausesValidation**
Container	**Parent**
DragIcon	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
DragMode	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
Enabled	**Enabled**
Height	**Height, Size** **Note** Coordinates are handled differently in Visual Basic .NET. For more information, see Coordinate System Changes in Visual Basic .NET.
HelpContextID	No equivalent. For more information, see Help Support Changes in Visual Basic .NET.
HWnd	**Handle**
Index	No equivalent. For more information, see Control Array Changes in Visual Basic .NET.
LargeChange	**LargeChange** **Note** The default value in Visual Basic 6.0 was 1, in Visual Basic .NET the default value is 10.
Left	**Left** **Note** Coordinates are handled differently in Visual Basic .NET. For more information, see Coordinate System Changes in Visual Basic .NET.
Max	**Maximum** **Note** The default value in Visual Basic 6.0 was 32767, in Visual Basic .NET the default value is 100.
Min	**Minimum** **Note** In Visual Basic 6.0 **Min** could be set to a value greater than **Max**; in Visual Basic .NET this is not allowed.
MouseIcon	No equivalent. For more information, see Cannot set a custom MousePointer.
MousePointer	**Cursor** For a list of constants, see MousePointer Changes in Visual Basic .NET.
Name	**Name**
Parent	**FindForm** method
RightToLeft **True** **False**	**RightToLeft** **RightToLeft.Yes** **RightToLeft.No**

SmallChange	**SmallChange**
TabIndex	**TabIndex**
TabStop	**TabStop**
Tag	**Tag**
Top	**Top** **Note** Coordinates are handled differently in Visual Basic .NET. For more information, see Coordinate System Changes in Visual Basic .NET.
Value	**Value** **Note** In Visual Basic 6.0, when the **Value** was changed, the **Change** event was raised. In Visual Basic .NET the **Change** event is not raised; use the **ValueChanged** event instead.
Visible	**Visible**
WhatsThisHelpID	No equivalent. For more information, see Help Support Changes in Visual Basic .NET.
Width	**Width, Size** **Note** Coordinates are handled differently in Visual Basic .NET. For more information, see Coordinate System Changes in Visual Basic .NET.

HScrollBar Methods

Visual Basic 6.0	Visual Basic .NET Equivalent
Drag	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
Move	**SetBounds** **Note** Coordinates are handled differently in Visual Basic .NET. For more information, see Coordinate System Changes in Visual Basic .NET.
Refresh	**Refresh**
SetFocus	**Focus**
ShowWhatsThis	No equivalent. For more information, see Help Support Changes in Visual Basic .NET.
ZOrder **0 – vbBringToFront** **1 - vbSendToBack**	**BringToFront()** or **SendToBack()** functions **BringToFront()** **SendToBack()**

HScrollBar Events

Visual Basic 6.0	Visual Basic .NET Equivalent
Change	**ValueChanged** **Note** The behavior of the **Change** and **Scroll** events have changed in Visual Basic .NET. For more information, see Code was changed from an event to a procedure.
DragDrop	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
DragOver	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
GotFocus	**Enter**
KeyDown	**KeyDown**
KeyPress	**KeyPress**
KeyUp	**KeyUp**
LostFocus	**Leave**

Scroll	**Scroll** **Note** The behavior of the **Change** and **Scroll** events have changed in Visual Basic .NET. For more inform ation, see Code was changed from an event to a procedure.
Validate	**Validating**

See Also

Introduction to the Windows Forms HScrollBar and VScrollBar Controls

Image Control Changes in Visual Basic .NET

The **Image** control in Visual Basic 6.0 has no direct equivalent in Visual Basic .NET. The **Image** control was a lightweight control; Visual Basic .NET does not support lightweight controls.

During upgrade, **Image** controls are upgraded to the Visual Basic .NET Windows Forms **PictureBox** control. The properties, methods, and events for the **Image** control were a subset of those for the **PictureBox** control (with the exception of the **Stretch** property). For more information, see PictureBox Control Changes in Visual Basic .NET.

The following table shows the mapping between the **Stretch** property and the equivalent for the Visual Basic .NET **PictureBox** control.

Stretch property equivalents

Visual Basic 6.0	Visual Basic .NET Equivalent
Stretch = **False**	**SizeMode** = **System.Windows.Forms.PictureBoxSizeMode.Normal**
Stretch = **True**	**SizeMode** = **System.Windows.Forms.PictureBoxSizeMode.StretchImage**

See Also

PictureBox Control Changes in Visual Basic .NET | Introduction to the Windows Forms PictureBox Control

Label Control Changes in Visual Basic .NET

The **Label** control in Visual Basic 6.0 is replaced by the Windows Forms **Label** control in Visual Basic .NET. The names of some properties, methods, events, and constants are different, and in some cases there are differences in behavior.

The following tables list Visual Basic 6.0 properties, methods, and events and their Visual Basic .NET equivalents. Where applicable, constants are indented beneath the property or method. All Visual Basic .NET constants map to the **System.Windows.Forms** namespace unless otherwise noted.

Links are provided as necessary to topics explaining differences in behavior. Where there is no direct equivalent in Visual Basic .NET, links are provided to topics that present alternatives.

Label Properties

Visual Basic 6.0	**Visual Basic .NET Equivalent**
Alignment **0 - Left Justify** **1 - Right Justify** **2 - Center**	**TextAlign** **System.Drawing.ContentAlignment.TopLeft** **System.Drawing.ContentAlignment.TopRight** **System.Drawing.ContentAlignment.TopCenter**
Appearance	No equivalent. For more information, see Appearance and BorderStyle Property Changes in Visual Basic .NET.
AutoSize	**AutoSize**
BackColor	**BackColor** **Note** Colors are handled differently in Visual Basic .NET. For more information, see Color Behavior Changes in Visual Basic .NET. For a list of constants, see Color Changes in Visual Basic .NET.
BackStyle	No equivalent. For more information, see BackStyle Property Changes in Visual Basic .NET.
BorderStyle **0 – None** **1 – Fixed Single**	**BorderStyle** **0 – BorderStyle.None** **1 – BorderStyle.FixedSingle**
Caption	**Text**
Container	**Parent**
DataChanged	No equivalent. For more information, see Data Changes in Visual Basic .NET.
DataField	No equivalent. For more information, see Data Changes in Visual Basic .NET.
DataFormat	No equivalent. For more information, see Data Changes in Visual Basic .NET.
DataMember	No equivalent. For more information, see Data Changes in Visual Basic .NET.
DataSource	No equivalent. For more information, see Data Changes in Visual Basic .NET.
DragIcon	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
DragMode	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
Enabled	**Enabled**
Font	**Font** **Note** Fonts are handled differently in Visual Basic .NET. For more information, see Font Changes in Visual Basic .NET.
FontBold	**Font** **Note** Fonts are handled differently in Visual Basic .NET. For more information, see Font Changes in Visual Basic .NET.

FontItalic	**Font** **Note** Fonts are handled differently in Visual Basic .NET. For more information, see Font Changes in Visual Basic .NET.
FontName	**Font** **Note** Fonts are handled differently in Visual Basic .NET. For more information, see Font Changes in Visual Basic .NET.
FontSize	**Font** **Note** Fonts are handled differently in Visual Basic .NET. For more information, see Font Changes in Visual Basic .NET.
FontStrikethru	**Font** **Note** Fonts are handled differently in Visual Basic .NET. For more information, see Font Changes in Visual Basic .NET.
FontUnderline	**Font** **Note** Fonts are handled differently in Visual Basic .NET. For more information, see Font Changes in Visual Basic .NET.
ForeColor	**ForeColor** **Note** Colors are handled differently in Visual Basic .NET. For more information, see Color Behavior Changes in Visual Basic .NET. For a list of constants, see Color Changes in Visual Basic .NET.
Height	**Height, Size** **Note** Coordinates are handled differently in Visual Basic .NET. For more information, see Coordinate System Changes in Visual Basic .NET.
Index	No equivalent. For more information, see Control Array Changes in Visual Basic .NET.
Left	**Left** **Note** Coordinates are handled differently in Visual Basic .NET. For more information, see Coordinate System Changes in Visual Basic .NET.
LinkItem	No equivalent. For more information, see Dynamic Data Exchange Changes in Visual Basic .NET.
LinkMode	No equivalent. For more information, see Dynamic Data Exchange Changes in Visual Basic .NET.
LinkTimeOut	No equivalent. For more information, see Dynamic Data Exchange Changes in Visual Basic .NET.
LinkTopic	No equivalent. For more information, see Dynamic Data Exchange Changes in Visual Basic .NET.
MouseIcon	No equivalent. For more information, see Cannot set a custom MousePointer.
MousePointer	**Cursor** For a list of constants, see MousePointer Changes in Visual Basic .NET.
Name	**Name**
OLEDropMode	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
Parent	**FindForm** method
RightToLeft **True** **False**	**RightToLeft** **RightToLeft.Yes** **RightToLeft.No**
TabIndex	**TabIndex**
Tag	**Tag**

ToolTipText	**ToolTip** component For more information, see ToolTip Changes in Visual Basic .NET.
Top	**Top** **Note** Coordinates are handled differently in Visual Basic .NET. For more information, see Coordinate System Changes in Visual Basic .NET.
UseMnemonic	**UseMnemonic**
Visible	**Visible**
WhatsThisHelpID	No equivalent. For more information, see Help Support Changes in Visual Basic .NET.
Width	**Width, Size** **Note** Coordinates are handled differently in Visual Basic .NET. For more information, see Coordinate System Changes in Visual Basic .NET.
WordWrap	No equivalent. **Note** Text in the Visual Basic .NET **Label** control wraps by default and cannot be changed.

Label Methods

Visual Basic 6.0	Visual Basic .NET Equivalent
Drag	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
LinkExecute	No equivalent. For more information, see Dynamic Data Exchange Changes in Visual Basic .NET.
LinkPoke	No equivalent. For more information, see Dynamic Data Exchange Changes in Visual Basic .NET.
LinkRequest	No equivalent. For more information, see Dynamic Data Exchange Changes in Visual Basic .NET.
LinkSend	No equivalent. For more information, see Dynamic Data Exchange Changes in Visual Basic .NET.
Move	**SetBounds** **Note** Coordinates are handled differently in Visual Basic .NET. For more information, see Coordinate System Changes in Visual Basic .NET.
OLEDrag	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
Refresh	**Refresh**
ShowWhatsThis	No equivalent. For more information, see Help Support Changes in Visual Basic .NET.
ZOrder **0 – vbBringToFront** **1 - vbSendToBack**	**BringToFront()** or **SendToBack()** functions **BringToFront()** **SendToBack()**

Label Events

Visual Basic 6.0	Visual Basic .NET Equivalent
Change	No equivalent. There are individual events (**TextChanged**, **SizeChanged**, and so forth) that are raised when properties are changed.
Click	**Click**
DblClick	**DoubleClick**
DragDrop	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
DragOver	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
LinkClose	No equivalent. For more information, see Dynamic Data Exchange Changes in Visual Basic .NET.
LinkError	No equivalent. For more information, see Dynamic Data Exchange Changes in Visual Basic .NET.
LinkNotify	No equivalent. For more information, see Dynamic Data Exchange Changes in Visual Basic .NET.
LinkOpen	No equivalent. For more information, see Dynamic Data Exchange Changes in Visual Basic .NET.
MouseDown	**MouseDown**

MouseMove	**MouseMove**
MouseUp	**MouseUp**
OLECompleteDrag	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
OLEDragDrop	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
OLEDragOver	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
OLEGiveFeedback	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
OLESetData	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
OLEStartDrag	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.

See Also

Introduction to the Windows Forms Label Control

Line Control Changes in Visual Basic .NET

The **Line** control in Visual Basic 6.0 has no equivalent in Visual Basic .NET. During upgrade, vertical or horizontal **Line** controls are replaced with the Windows Forms **Label** control, with the **Text** property set to an empty string, the **BorderStyle** property set to **none**, and the **BackColor**, **Width,** and **Height** properties set to match the original control.

Line controls that were not vertical or horizontal are not upgraded. You can replace the **Line** control using the graphics functions built into the .NET Framework. For more information, see Graphics Changes in Visual Basic .NET.

See Also

Graphics Changes in Visual Basic .NET

ListBox Control Changes in Visual Basic .NET

The **ListBox** control in Visual Basic 6.0 is replaced by either the Windows Forms **ListBox** control or the **CheckedListBox** control in Visual Basic .NET. The names of some properties, methods, events, and constants are different, and in some cases there are differences in behavior.

During upgrade, if the **Style** property of a Visual Basic 6.0 **Listbox** is set to **1 – Checkbox**, it is upgraded to the **CheckedListBox** control; otherwise, it is upgraded to the **ListBox** control.

The following tables list Visual Basic 6.0 properties, methods, and events and their Visual Basic .NET equivalents. Where applicable, constants are indented beneath the property or method. All Visual Basic .NET constants map to the **System.Windows.Forms** namespace unless otherwise noted.

Links are provided as necessary to topics explaining differences in behavior. Where there is no direct equivalent in Visual Basic .NET, links are provided to topics that present alternatives.

ListBox Properties

Visual Basic 6.0	Visual Basic .NET Equivalent
Appearance	No equivalent. For more information, see Appearance and BorderStyle Property Changes in Visual Basic .NET.
BackColor	**BackColor** **Note** Colors are handled differently in Visual Basic .NET. For more information, see Color Behavior Changes in Visual Basic .NET. For a list of constants, see Color Changes in Visual Basic .NET.
CausesValidation	**CausesValidation**
Columns	**MultiColumn** and **ColumnWidth** **Note** The **Columns** property took an integer specifying the number of columns. The **MultiColumn** property takes a Boolean value, and the **ColumnWidth** property takes an integer specifying the width in pixels. If the width of the **ListBox** control is changed at run time, you may also need to set the **ColumnWidth** property for each column.
Container	**Parent**
DataChanged	No equivalent. For more information, see Data Changes in Visual Basic .NET.
DataField	No equivalent. For more information, see Data Changes in Visual Basic .NET.
DataFormat	No equivalent. For more information, see Data Changes in Visual Basic .NET.
DataMember	No equivalent. For more information, see Data Changes in Visual Basic .NET.
DataSource	No equivalent. For more information, see Data Changes in Visual Basic .NET.
DragIcon	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.

DragMode	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
Enabled	**Enabled**
Font	**Font** **Note** Fonts are handled differently in Visual Basic .NET. For more information, see Font Changes in Visual Basic .NET.
FontBold	**Font** **Note** Fonts are handled differently in Visual Basic .NET. For more information, see Font Changes in Visual Basic .NET.
FontItalic	**Font** **Note** Fonts are handled differently in Visual Basic .NET. For more information, see Font Changes in Visual Basic .NET.
FontName	**Font** **Note** Fonts are handled differently in Visual Basic .NET. For more information, see Font Changes in Visual Basic .NET.
FontSize	**Font** **Note** Fonts are handled differently in Visual Basic .NET. For more information, see Font Changes in Visual Basic .NET.
FontStrikethru	**Font** **Note** Fonts are handled differently in Visual Basic .NET. For more information, see Font Changes in Visual Basic .NET.
FontUnderline	**Font** **Note** Fonts are handled differently in Visual Basic .NET. For more information, see Font Changes in Visual Basic .NET.
ForeColor	**ForeColor** **Note** Colors are handled differently in Visual Basic .NET. For more information, see Color Behavior Changes in Visual Basic .NET. For a list of constants, see Color Changes in Visual Basic .NET.
Height	**Height, Size** **Note** Coordinates are handled differently in Visual Basic .NET. For more information, see Coordinate System Changes in Visual Basic .NET.
HelpContextID	No equivalent. For more information, see Help Support Changes in Visual Basic .NET.
HWnd	**Handle**
Index	No equivalent. For more information, see Control Array Changes in Visual Basic .NET.
IntegralHeight	**IntegralHeight**
ItemData	No equivalent. For more information, see ItemData property cannot be upgraded.

Left	**Left** **Note** Coordinates are handled differently in Visual Basic .NET. For more information, see Coordinate System Changes in Visual Basic .NET.
List	**Items**
ListCount	**Items.Count**
ListIndex	**SelectedIndex**
MouseIcon	No equivalent. For more information, see Cannot set a custom MousePointer.
MousePointer	**Cursor** For a list of constants, see MousePointer Changes in Visual Basic .NET.
MultiSelect **0 – None** **1 – Simple** **2 – Extended**	**SelectionMode** **SelectionMode.One** **SelectionMode.MultiSimple** **SelectionMode.MultiExtended**
Name	**Name**
NewIndex	No equivalent. For more information, see NewIndex property cannot be upgraded.
OLEDragMode	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
OLEDropMode	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
Parent	**FindForm** method
RightToLeft **True** **False**	**RightToLeft** **RightToLeft.Yes** **RightToLeft.No**
SelCount	**SelectedItems.Count**
Selected	**GetSelected, SetSelected** (**ListBox** control) **GetItemChecked, SetItemChecked** (**CheckedListBox** control)
Sorted	**Sorted**

Style **0 - Standard** **1 - Checkbox**	No equivalent property. **0 – Standard** maps to the **ListBox** control, **1 – Checkbox** maps to the **CheckedListBox** control.
TabIndex	**TabIndex**
TabStop	**TabStop**
Tag	**Tag**
Text	**Text**
ToolTipText	**ToolTip** component For more information, see ToolTip Changes in Visual Basic .NET.
Top	**Top** **Note** Coordinates are handled differently in Visual Basic .NET. For more information, see Coordinate System Changes in Visual Basic .NET.
TopIndex	**TopIndex**
Visible	**Visible**
WhatsThisHelpID	No equivalent. For more information, see Help Support Changes in Visual Basic .NET.
Width	**Width, Size** **Note** Coordinates are handled differently in Visual Basic .NET. For more information, see Coordinate System Changes in Visual Basic .NET.

ListBox Methods

Name	Visual Basic .NET Equivalent
AddItem	**Items.Insert** or **Items.Add**
Clear	**Items.Clear**
Drag	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
Move	**SetBounds** **Note** Coordinates are handled differently in Visual Basic .NET. For more information, see Coordinate System Changes in Visual Basic .NET.
OLEDrag	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
Refresh	**Refresh**
RemoveItem	**Items.Remove**
SetFocus	**Focus**
ShowWhatsThis	No equivalent. For more information, see Help Support Changes in Visual Basic .NET.

ZOrder **0 – vbBringToFront** **1 - vbSendToBack**	**BringToFront()** or **SendToBack()** functions **BringToFront()** **SendToBack()**

ListBox Events

Visual Basic 6.0	**Visual Basic .NET Equivalent**
Click	**SelectedIndexChanged** **Note** In Visual Basic 6.0, the **Click** event was raised when the **CheckBox** state was changed programmatically. In Visual Studio .NET, the **Click** event is not raised; therefore, you should use the **CheckStateChanged** event.
DblClick	No equivalent. Use the **SelectedIndexChanged** or **TextChanged** events.
DragDrop	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
DragOver	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
GotFocus	**Enter**
ItemCheck	**ItemCheck** (**CheckedListBox** only) **Note** In Visual Basic 6.0, when an event was raised, the check state was already changed. In Visual Basic .NET, the check state has not yet changed. You can get the pending value from the **ItemCheckEventArgs** argument passed to the event.
KeyDown	**KeyDown**
KeyPress	**KeyPress**
KeyUp	**KeyUp**
LostFocus	**Leave**
MouseDown	**MouseDown**
MouseMove	**MouseMove**
MouseUp	**MouseUp**
OLECompleteDrag	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
OLEDragDrop	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
OLEDragOver	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
OLEGiveFeedback	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
OLESetData	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
OLEStartDrag	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
Scroll	No equivalent. For more information, see TopIndex Property and Scroll Event Changes in Visual Basic .NET.
Validate	**Validating**

See Also

Introduction to the Windows Forms ListBox Control

MDIForm Object Changes in Visual Basic .NET

In Visual Basic 6.0, an **MDIForm** was a special type of form that acted as a container for an MDI application. In VisualBasic .NET, an **MdiForm** is any form with the **IsMdiContainer** property set to **true**.

In Visual Basic 6.0, setting the **MDIChild** property on a child form identified it as a child of the MDI form; an MDI application could have only one MDI form. In Visual Basic .NET, the **MdiParent** property replaces the **MDIChild** property and identifies the MDI form that will contain the child form; MDI applications can have multiple MDI containers.

The following table lists Visual Basic 6.0 properties and methods and their Visual Basic .NET equivalents. Only properties and methods that are unique to the **MDIForm** object are listed — for other properties, methods, and events see Form Object Changes in Visual Basic .NET.

MDIForm Properties and Methods

Visual Basic 6.0	**Visual Basic .NET Equivalent**
ActiveForm property	**ActiveMdiCHild**
Arrange method **0 - vbCascade** **1 – vbTileHorizontal** **2 – vbTileVertical** **3 – vbArrangeIcons**	**LayoutMdi** **0 - MdiLayout.Cascade** **1 - MdiLayout.TileHorizontal** **2 - MdiLayout.TileVertical** **3 - MdiLayout.ArrangeIcons**
AutoShowChildren property	No equivalent. For more information, see MDI form Show behavior has changed.
BackColor property	No equivalent. The **BackColor** for an MDI form will always be the default.
Picture property	No equivalent. MDI forms in Visual Basic .NET cannot display pictures.

See Also

MDI Changes in Visual Basic .NET | Form Object Changes in Visual Basic .NET | Forms Task Changes in Visual Basic .NET

Menu Object Changes in Visual Basic .NET

In Visual Basic 6.0, a **Menu** object was created using the Menu Editor. **Menu** objects were tied to the form for which they were created, but they could be modified at run time or displayed as context menus.

In Visual Basic .NET, the **Menu** object is replaced by the **MainMenu** and **ContextMenu** controls. Menus may be created using the Windows Forms Designer or they may be created in code.

The following tables list Visual Basic 6.0 properties, methods, and events and their Visual Basic .NET equivalents. Where there are no direct equivalents, links are provided to additional information. Unless otherwise noted, all objects are in the **System.Windows.Forms** namespace.

Menu Properties

Visual Basic 6.0	Visual Basic .NET Equivalent
Caption	**Text**
Checked	**Checked**
Enabled	**Enabled**
HelpContextID	No equivalent. For details, see Help Support Changes in Visual Basic .NET.
Index	No direct equivalent. Use the **MenuItemCollection** class.
Name	**Name**
NegotiatePosition	No equivalent. This property was only used for OLE linking and embedding.
Parent	**GetMainMenu().GetForm()**
Shortcut	**Shortcut**
Tag	No equivalent. Tags for menu items are not supported.
Visible	**Visible**
WindowList	**MdiList**

Menu Events

Visual Basic 6.0	Visual Basic .NET Equivalent
Click	**Click** or **Popup**. **Note** The **Popup** event is raised prior to the **Click** event when a menu item has children.

See Also

Menu Changes in Visual Basic .NET | Control Changes in Visual Basic .NET | Menu.MenuItemCollection Class | MenuItem.Popup Event

OLE Container Control Changes in Visual Basic .NET

In Visual Basic 6.0, the **OLE Container** control was used to add insertable OLE objects to your forms. There is no **OLE Container** control in Visual Basic .NET. If you need the functionality of the **OLE Container** control, in some cases you can substitute the Visual Basic 6.0 **WebBrowser** ActiveX control and use it as an **OLE Container**.

See Also

Things to Consider Before Upgrading | Adding ActiveX Controls to Windows Forms

OptionButton Control Changes in Visual Basic .NET

The **OptionButton** control in Visual Basic 6.0 is replaced by the Windows Forms **RadioButton** control in Visual Basic .NET. The names of some properties, methods, events, and constants are different, and in some cases there are differences in behavior.

The following tables list Visual Basic 6.0 properties, methods, and events and their Visual Basic .NET equivalents. Where applicable, constants are indented beneath the property or method. All Visual Basic .NET constants map to the **System.Windows.Forms** namespace unless otherwise noted.

Links are provided as necessary to topics explaining differences in behavior. Where there is no direct equivalent in Visual Basic .NET, links are provided to topics that present alternatives.

OptionButton Properties

Visual Basic 6.0	Visual Basic .NET Equivalent
Alignment **0 - Left Justify** **1 - Right Justify**	**TextAlign** **System.Drawing.ContentAlignment.MiddleLeft** **System.Drawing.ContentAlignment.MiddleRight**
Appearance **0 – Flat** **1 – 3D**	**FlatStyle** **FlatStyle.Flat** **FlatStyle.Standard**
BackColor	**BackColor** **Note** Colors are handled differently in Visual Basic .NET. For more information, see Color Behavior Changes in Visual Basic .NET. For a list of constants, see Color Changes in Visual Basic .NET.
Caption	**Text**
CausesValidation	**CausesValidation**
Container	**Parent**
DisabledPicture	No equivalent. For more information, see Style Property Changes in Visual Basic .NET.
DownPicture	No equivalent. For more information, see Style Property Changes in Visual Basic .NET.
DragIcon	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
DragMode	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
Enabled	**Enabled**
Font	**Font** **Note** Fonts are handled differently in Visual Basic .NET. For more information, see Font Changes in Visual Basic .NET.
FontBold	**Font** **Note** Fonts are handled differently in Visual Basic .NET. For more information, see Font Changes in Visual Basic .NET.
FontItalic	**Font** **Note** Fonts are handled differently in Visual Basic .NET. For more information, see Font Changes in Visual Basic .NET.

FontName	**Font** **Note** Fonts are handled differently in Visual Basic .NET. For more information, see Font Changes in Visual Basic .NET.
FontSize	**Font** **Note** Fonts are handled differently in Visual Basic .NET. For more information, see Font Changes in Visual Basic .NET.
FontStrikethru	**Font** **Note** Fonts are handled differently in Visual Basic .NET. For more information, see Font Changes in Visual Basic .NET.
FontUnderline	**Font** **Note** Fonts are handled differently in Visual Basic .NET. For more information, see Font Changes in Visual Basic .NET.
ForeColor	**ForeColor** **Note** Colors are handled differently in Visual Basic .NET. For more information, see Color Behavior Changes in Visual Basic .NET. For a list of constants, see Color Changes in Visual Basic .NET.
Height	**Height, Size** **Note** Coordinates are handled differently in Visual Basic .NET. For more information, see Coordinate System Changes in Visual Basic .NET.
HelpContextID	No equivalent. For more information, see Help Support Changes in Visual Basic .NET.
HWnd	**Handle**
Index	No equivalent. For more information, see Control Array Changes in Visual Basic .NET.
Left	**Left** **Note** Coordinates are handled differently in Visual Basic .NET. For more information, see Coordinate System Changes in Visual Basic .NET.
MaskColor	No equivalent. For more information, see MaskColor Changes in Visual Basic .NET.
MouseIcon	No equivalent. For more information, see Cannot set a custom MousePointer.
MousePointer	**Cursor** For a list of constants, see MousePointer Changes in Visual Basic .NET.
Name	**Name**
OLEDropMode	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
Parent	**FindForm** method
Picture	**Image**
RightToLeft **True** **False**	**RightToLeft** **RightToLeft.Yes** **RightToLeft.No**

Style **0 – Standard** **1 - Graphical**	**Appearance** **Note** There are differences in how the Graphical style is handled in Visual Basic .NET. For more information, see Style Property Changes in Visual Basic .NET. **Appearance.Normal** **Appearance.Button**
TabIndex	**TabIndex**
TabStop	**TabStop**
Tag	**Tag**
ToolTipText	**ToolTip** component For more information, see ToolTip Changes in Visual Basic .NET.
Top	**Top** **Note** Coordinates are handled differently in Visual Basic .NET. For more information, see Coordinate System Changes in Visual Basic .NET.
UseMaskColor	No equivalent. For more information, see MaskColor Changes in Visual Basic .NET.
Value	**Checked**
Visible	**Visible**
WhatsThisHelpID	No equivalent. For more information, see Help Support Changes in Visual Basic .NET.
Width	**Width, Size** **Note** Coordinates are handled differently in Visual Basic .NET. For more information, see Coordinate System Changes in Visual Basic .NET.

OptionButton Methods

Visual Basic 6.0	Visual Basic .NET Equivalent
Drag	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
Move	**SetBounds** **Note** Coordinates are handled differently in Visual Basic .NET. For more information, see Coordinate System Changes in Visual Basic .NET.
OLEDrag	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
Refresh	**Refresh**
SetFocus	**Focus**
ShowWhatsThis	No equivalent. For more information, see Help Support Changes in Visual Basic .NET.
ZOrder **0 – vbBringToFront** **1 - vbSendToBack**	**BringToFront()** or **SendToBack()** functions **BringToFront()** **SendToBack()**

OptionButton Events

Visual Basic 6.0	Visual Basic .NET Equivalent
Click	**CheckedChanged** **Note** In Visual Basic 6.0, the **Click** event was raised only when the state was changed to **true**; in Visual Basic .NET, it is raised for both **true** and **false**.
DblClick	No equivalent. Use the **Click** event.

DragDrop	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
DragOver	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
GotFocus	**Enter**
KeyDown	**KeyDown**
KeyPress	**KeyPress**
KeyUp	**KeyUp**
LostFocus	**Leave**
MouseDown	**MouseDown**
MouseMove	**MouseMove**
MouseUp	**MouseUp**
OLECompleteDrag	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
OLEDragDrop	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
OLEDragOver	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
OLEGiveFeedback	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
OLESetData	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
OLEStartDrag	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
Validate	**Validating**

See Also

Introduction to the Windows Forms RadioButton Control

PictureBox Control Changes in Visual Basic .NET

The **PictureBox** control in Visual Basic 6.0 is replaced by the Windows Forms **PictureBox** control in Visual Basic .NET. The names of some properties, methods, events, and constants are different, and in some cases there are differences in behavior.

The Visual Basic 6.0 **PictureBox** control was a container control; the Visual Basic .NET **PictureBox** control is not. During upgrade, **PictureBox** controls that contain controls are upgraded to **Panel** controls.

The following tables list Visual Basic 6.0 properties, methods, and events and their Visual Basic .NET equivalents. Where applicable, constants are indented beneath the property or method. All Visual Basic .NET constants map to the **System.Windows.Forms** namespace unless otherwise noted.

Links are provided as necessary to topics explaining differences in behavior. Where there is no direct equivalent in Visual Basic .NET, links are provided to topics that present alternatives.

PictureBox Properties

Visual Basic 6.0	**Visual Basic .NET Equivalent**
Align **0 – None** **1 – Top** **2 – Bottom** **3 – Left** **4 – Right**	**Dock** **DockStyle.None** **DockStyle.Top** **DockStyle.Bottom** **DockStyle.Left** **DockStyle.Right**
Appearance	No equivalent. For more information, see Appearance and BorderStyle Property Changes in Visual Basic .NET.
AutoRedraw	No equivalent. For more information, see Graphics Changes in Visual Basic .NET.
AutoSize **False** **True**	**PictureBoxSizeMode** **Note** For **PictureBox** controls that are upgraded to **Panel** controls, there is no equivalent for **AutoSize**. **PictureBoxSizeMode.Normal** **PictureBoxSizeMode.AutoSize**
BackColor	**BackColor** **Note** Colors are handled differently in Visual Basic .NET. For more information, see Color Behavior Changes in Visual Basic .NET. For a list of constants, see Color Changes in Visual Basic .NET.
BorderStyle **0 – None** **1 – Fixed Single**	**Borderstyle** **Note** In Visual Basic 6.0, the default was **1 – Fixed Single**, in Visual Basic .NET the default is **None**. **BorderStyle.None** **BorderStyle.FixedSingle**
CausesValidation	**CausesValidation**
ClipControls	No equivalent. For more information, see Graphics Changes in Visual Basic .NET.
Container	**Parent**
CurrentX	No equivalent. For more information, see Graphics Changes in Visual Basic .NET.
CurrentY	No equivalent. For more information, see Graphics Changes in Visual Basic .NET.

DataChanged	No equivalent. For more information, see Data Changes in Visual Basic .NET.
DataField	No equivalent. For more information, see Data Changes in Visual Basic .NET.
DataFormat	No equivalent. For more information, see Data Changes in Visual Basic .NET.
DataMember	No equivalent. For more information, see Data Changes in Visual Basic .NET.
DataSource	No equivalent. For more information, see Data Changes in Visual Basic .NET.
DragIcon	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
DragMode	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
DrawMode	No equivalent. For more information, see Graphics Changes in Visual Basic .NET.
DrawStyle	No equivalent. For more information, see Graphics Changes in Visual Basic .NET.
DrawWidth	No equivalent. For more information, see Graphics Changes in Visual Basic .NET.
Enabled	**Enabled**
FillColor	No equivalent. For more information, see Graphics Changes in Visual Basic .NET.
FillStyle	No equivalent. For more information, see Graphics Changes in Visual Basic .NET.
Font	**Font** **Note** Fonts are handled differently in Visual Basic .NET. For more information, see Font Changes in Visual Basic .NET.
FontBold	**Font** **Note** Fonts are handled differently in Visual Basic .NET. For more information, see Font Changes in Visual Basic .NET.
FontItalic	**Font** **Note** Fonts are handled differently in Visual Basic .NET. For more information, see Font Changes in Visual Basic .NET.
FontName	**Font** **Note** Fonts are handled differently in Visual Basic .NET. For more information, see Font Changes in Visual Basic .NET.
FontSize	**Font** **Note** Fonts are handled differently in Visual Basic .NET. For more information, see Font Changes in Visual Basic .NET.
FontStrikethru	**Font** **Note** Fonts are handled differently in Visual Basic .NET. For more information, see Font Changes in Visual Basic .NET.
FontTransparent	No equivalent. For more information, see Graphics Changes in Visual Basic .NET.
FontUnderline	**Font** **Note** Fonts are handled differently in Visual Basic .NET. For more information, see Font Changes in Visual Basic .NET.
ForeColor	**ForeColor** **Note** Colors are handled differently in Visual Basic .NET. For more information, see Color Behavior Changes in Visual Basic .NET. For a list of constants, see Color Changes in Visual Basic .NET.
HasDC	No equivalent. For more information, see Graphics Changes in Visual Basic .NET.
HDC	No equivalent. For more information, see Graphics Changes in Visual Basic .NET.

Height	**Height, Size** **Note** Coordinates are handled differently in Visual Basic .NET. For more information, see Coordinate System Changes in Visual Basic .NET.
HelpContextID	No equivalent. For more information, see Help Support Changes in Visual Basic .NET.
HWnd	**Handle**
Image	No equivalent. For more information, see Graphics Changes in Visual Basic .NET. **Note** In Visual Basic .NET, the **Image** property is the equivalent of the Visual Basic 6.0 **Picture** property.
Index	No equivalent. For more information, see Control Array Changes in Visual Basic .NET.
Left	**Left** **Note** Coordinates are handled differently in Visual Basic .NET. For more information, see Coordinate System Changes in Visual Basic .NET.
LinkItem	No equivalent. For more information, see Dynamic Data Exchange Changes in Visual Basic .NET.
LinkMode	No equivalent. For more information, see Dynamic Data Exchange Changes in Visual Basic .NET.
LinkTimeOut	No equivalent. For more information, see Dynamic Data Exchange Changes in Visual Basic .NET.
LinkTopic	No equivalent. For more information, see Dynamic Data Exchange Changes in Visual Basic .NET.
MouseIcon	No equivalent. For more information, see Cannot set a custom MousePointer.
MousePointer	**Cursor** For a list of constants, see MousePointer Changes in Visual Basic .NET.
Name	**Name**
OLEDragMode	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
OLEDropMode	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
Parent	**FindForm** method
Picture	**Image** **Note** For Visual Basic 6.0 **PictureBox** controls that were upgraded to **Panel** controls, the **Picture** property is mapped to the **BackgroundPicture** property.
RightToLeft **True** **False**	**RightToLeft** **RightToLeft.Yes** **RightToLeft.No**
ScaleHeight	No equivalent. For more information, see Coordinate System Changes in Visual Basic .NET.
ScaleLeft	No equivalent. For more information, see Coordinate System Changes in Visual Basic .NET.
ScaleMode	No equivalent. For more information, see Coordinate System Changes in Visual Basic .NET.
ScaleTop	No equivalent. For more information, see Coordinate System Changes in Visual Basic .NET.
ScaleWidth	No equivalent. For more information, see Coordinate System Changes in Visual Basic .NET.
TabIndex	**TabIndex**
TabStop	**TabStop**
Tag	**Tag**
ToolTipText	**ToolTip** component For more information, see ToolTip Changes in Visual Basic .NET.

Top	**Top** **Note** Coordinates are handled differently in Visual Basic .NET. For more information, see Coordinate System Changes in Visual Basic .NET.
Visible	**Visible**
WhatsThisHelpID	No equivalent. For more information, see Help Support Changes in Visual Basic .NET.
Width	**Width, Size** **Note** Coordinates are handled differently in Visual Basic .NET. For more information, see Coordinate System Changes in Visual Basic .NET.

PictureBox Methods

Visual Basic 6.0	Visual Basic .NET Equivalent
Circle	No equivalent. For more information, see Graphics Changes in Visual Basic .NET.
Cls	No equivalent. For more information, see Graphics Changes in Visual Basic .NET.
Drag	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
Line	No equivalent. For more information, see Graphics Changes in Visual Basic .NET.
LinkExecute	No equivalent. For more information, see Dynamic Data Exchange Changes in Visual Basic .NET.
LinkPoke	No equivalent. For more information, see Dynamic Data Exchange Changes in Visual Basic .NET.
LinkRequest	No equivalent. For more information, see Dynamic Data Exchange Changes in Visual Basic .NET.
LinkSend	No equivalent. For more information, see Dynamic Data Exchange Changes in Visual Basic .NET.
Move	**SetBounds** **Note** Coordinates are handled differently in Visual Basic .NET. For more information, see Coordinate System Changes in Visual Basic .NET.
OLEDrag	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
PaintPicture	No equivalent. For more information, see Graphics Changes in Visual Basic .NET.
Point	No equivalent. For more information, see Graphics Changes in Visual Basic .NET.
Print	No equivalent. For more information, see Graphics Changes in Visual Basic .NET.
Pset	No equivalent. For more information, see Graphics Changes in Visual Basic .NET
Refresh	**Refresh**
Scale	No equivalent. For more information, see Coordinate System Changes in Visual Basic .NET.
ScaleX	No equivalent. For more information, see Coordinate System Changes in Visual Basic .NET.
ScaleY	No equivalent. For more information, see Coordinate System Changes in Visual Basic .NET.
SetFocus	**Activate**
ShowWhatsThis	No equivalent. For more information, see Help Support Changes in Visual Basic .NET.
TextHeight	No equivalent. For more information, see Graphics Changes in Visual Basic .NET.
TextWidth	No equivalent. For more information, see Graphics Changes in Visual Basic .NET.
ZOrder **0 – vbBringToFront** **1 - vbSendToBack**	**BringToFront()** or **SendToBack()** functions **BringToFront()** **SendToBack()**

PictureBox Events

Visual Basic 6.0	Visual Basic .NET Equivalent
Change	No equivalent. There are individual events (**BackgroundImageChanged**, **SizeChanged**, and so forth) that are raised when some properties are changed.
Click	**Click**
DblClick	**DoubleClick**
DragDrop	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.

DragOver	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
GotFocus	**Enter**
KeyDown	No equivalent. The Visual Basic .NET **PictureBox** control is not a container control and does not receive keystrokes.
KeyPress	No equivalent. The Visual Basic .NET **PictureBox** control is not a container control and does not receive keystrokes.
KeyUp	No equivalent. The Visual Basic .NET **PictureBox** control is not a container control and does not receive keystrokes.
LinkClose	No equivalent. For more information, see Dynamic Data Exchange Changes in Visual Basic .NET.
LinkError	No equivalent. For more information, see Dynamic Data Exchange Changes in Visual Basic .NET.
LinkNotify	No equivalent. For more information, see Dynamic Data Exchange Changes in Visual Basic .NET.
LinkOpen	No equivalent. For more information, see Dynamic Data Exchange Changes in Visual Basic .NET.
LostFocus	**Leave**
MouseDown	**MouseDown**
MouseMove	**MouseMove**
MouseUp	**MouseUp**
OLECompleteDrag	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
OLEDragDrop	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
OLEDragOver	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
OLEGiveFeedback	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
OLESetData	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
OLEStartDrag	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
Paint	**Paint**
Resize	**Resize**
Validate	**Validating**

See Also

Introduction to the Windows Forms PictureBox Control

Printer Object Changes in Visual Basic .NET

The **Printer** object in Visual Basic 6.0 is replaced by the **PrintDocument** component in Visual Basic .NET. The behavior is different, but in most cases the functionality can be duplicated.

The following tables list Visual Basic 6.0 properties, methods, and events and their Visual Basic .NET equivalents. Where there are no direct equivalents, links are provided to additional information. Unless otherwise noted, all objects are in the **System.Drawing** namespace.

Printer Properties

Visual Basic 6.0	Visual Basic .NET Equivalent
ColorMode	No longer necessary. This property controlled whether output would be printed in monochrome on a color printer; it is now up to the printer to expose this as an advanced option. The **SupportsColor** property of the **PrinterSettings** class can be used to determine the color capabilities of a printer.
Copies	**Copies** property of the **PrinterSettings** class.
CurrentX	No equivalent. Replaced by location and dimension arguments of various methods of the **Graphics** class.
CurrentY	No equivalent. Replaced by location and dimension arguments of various methods of the **Graphics** class.
DeviceName	**PrinterName** property of the **PrinterSettings** class.
DrawMode	No equivalent. For details, see Graphics Changes in Visual Basic .NET.
DrawStyle	No equivalent. For details, see Graphics Changes in Visual Basic .NET.
DrawWidth	No equivalent. For details, see Graphics Changes in Visual Basic .NET.
DriverName	No equivalent. No longer needed; printer drivers are managed by Windows.
Duplex	**Duplex** property of the **PrinterSettings** class.
FillColor	No equivalent. For details, see Graphics Changes in Visual Basic .NET.
FillStyle	No equivalent. For details, see Graphics Changes in Visual Basic .NET.
Font	No equivalent. For details, see Font Changes in Visual Basic .NET.
FontBold	No equivalent. For details, see Font Changes in Visual Basic .NET.
FontCount	No equivalent. For details, see Font Changes in Visual Basic .NET.
FontItalic	No equivalent. For details, see Font Changes in Visual Basic .NET.
FontName	No equivalent. For details, see Font Changes in Visual Basic .NET.
Fonts	No equivalent. For details, see Font Changes in Visual Basic .NET.
FontSize	No equivalent. For details, see Font Changes in Visual Basic .NET.
FontStrikeThru	No equivalent. For details, see Font Changes in Visual Basic .NET.
FontTransparent	No equivalent. For details, see Font Changes in Visual Basic .NET.

FontUnderline	No equivalent. For details, see Font Changes in Visual Basic .NET.
ForeColor	No equivalent. For details, see Color Behavior Changes in Visual Basic .NET.
hDC	No longer necessary. An instance of a **PrintDocument** component is the equivalent of a device context. **Note** Graphics objects have **GetHDC** and **ReleaseHDC** methods that can be used in advanced scenarios.
Height	**PaperSize** property of the **PageSettings** class.
Orientation	**Landscape** property of the **PageSettings** class.
Page	No equivalent. The current page number is not tracked; however, you can easily do this by setting a variable in the **Begin Print** event and incrementing it in the **PrintPage** event.
PaperBin	**PaperSources** property of the **PrinterSettings** class.
PaperSize	**PaperSize** property of the **PageSettings** class.
Port	No longer necessary. The **PrintPreviewDialog** control automatically sets port information.
PrintQuality	**PrinterResolutions** property of the **PrinterSettings** class.
RightToLeft	No longer necessary. The direction of printing is controlled by the localization settings in Windows.
ScaleHeight	No equivalent. For details, see Coordinate System Changes in Visual Basic .NET.
ScaleLeft	No equivalent. For details, see Coordinate System Changes in Visual Basic .NET.
ScaleMode	No equivalent. For details, see Coordinate System Changes in Visual Basic .NET.
ScaleTop	No equivalent. For details, see Coordinate System Changes in Visual Basic .NET.
ScaleWidth	No equivalent. For details, see Coordinate System Changes in Visual Basic .NET.
TrackDefault	No direct equivalent. The **IsDefaultPrinter** property of the **PrinterSettings** class can be used to determine if a printer is the default, but printing is no longer halted if the default printer changes.
TwipsPerPixelX	No longer necessary. Measurements in Visual Basic .NET are always in pixels.
TwipsPerPixelY	No longer necessary. Measurements in Visual Basic .NET are always in pixels.
Width	**PaperSize** property of the **PageSettings** class.
Zoom	No longer necessary. If the printer has zoom capabilities, settings are automatically exposed in the Print dialog box.

Printer Methods

Visual Basic 6.0	Visual Basic .NET Equivalent
Circle	**Graphics.DrawEllipse**
EndDoc	**Print**
KillDoc	**PrintEventArgs.Cancel**
Line	**Graphics.DrawLine**
NewPage	**PrintPageEventArgs.HasMorePages**
PaintPicture	**Graphics.DrawImage**
PSet	**Graphics.DrawLine**
Scale	No equivalent. For details, see Coordinate System Changes in Visual Basic .NET.
ScaleX	No equivalent. For details, see Coordinate System Changes in Visual Basic .NET.

ScaleY	No equivalent. For details, see Coordinate System Changes in Visual Basic .NET.
TextHeight	**Font** parameter of **Graphics.DrawString**
TextWidth	**Font** parameter of **Graphics.DrawString**

See Also

Printing Changes in Visual Basic .NET | Windows Forms Print Support | Graphics Class | PageSettings Class | PrinterSettings Class

Printers Collection Changes in Visual Basic .NET

In Visual Basic 6.0, the **Printers** collection was used to return information about available printers on a system. In Visual Basic .NET, the **PrintDialog** control returns information about available printers; properties for a given printer can be retrieved using the **PrinterSettings** class.

The **Printers** collection had two properties: **Item** and **Count**. These were standard collection properties; the **PrintDialog** control has no equivalent properties.

See Also

Printing Changes in Visual Basic .NET | Windows Forms Print Support |
Introduction to the Windows Forms PrintDialog Component | PrinterSettings Class

RDO Data Control Changes in Visual Basic .NET

The Visual Basic 6.0 RDO Data Control was used as a mechanism for binding controls to a database using RDO. Visual Basic .NET has no equivalent for the Data Control; the data-binding architecture has changed and RDO is no longer supported. For more information, see Data Changes in Visual Basic .NET.

See Also

Things to Consider Before Upgrading | Data Changes in Visual Basic .NET

Screen Object Changes in Visual Basic .NET

In Visual Basic 6.0, the **Screen** object provided global information about your application, and in some case allowed you to set global properties. In Visual Basic .NET, there is no equivalent object, but most of the functionality can be duplicated using the .NET Framework.

The following table lists Visual Basic 6.0 properties their Visual Basic .NET equivalents. Links are provided as necessary to topics explaining differences in behavior. Where there is no direct equivalent in Visual Basic .NET, links are provided to topics that present alternatives.

Screen Properties

Visual Basic 6.0	Visual Basic .NET Equivalent
ActiveControl	**System.Windows.Forms.Application.ActiveForm.ActiveControl**
ActiveForm	**System.Windows.Forms.Application.ActiveForm**
FontCount	No equivalent. The behavior for enumerating fonts is different. For more information, see Font Changes in Visual Basic .NET.
Fonts	**System.Drawing.FontFamilies** **Note** The behavior for enumerating fonts is different. For more information, see Font Changes in Visual Basic .NET.
Height	**System.Windows.Forms.Screen.PrimaryScreen.Bounds.Height**
MouseIcon	No equivalent. For more information, see Cannot set a custom MousePointer.
MousePointer	**System.Windows.Forms.Cursor** **Note** In Visual Basic 6.0, changing **Screen.MousePointer** changed the cursor's appearance until **Screen.MousePointer** was changed again. In Visual Basic .NET, it remains changed only until Windows messages are processed again (until the next DoEvents call or until the program's processing is done).
TwipsPerPixelX	No equivalent. In Visual Basic .NET, coordinates are in pixels and twips are not used as a unit of measurement.
TwipsPerPixelY	No equivalent. In Visual Basic .NET, coordinates are in pixels and twips are not used as a unit of measurement.
Width	**System.Windows.Forms.Screen.PrimaryScreen.Bounds.Width**

See Also

App Object Changes in Visual Basic .NET

Shape Control Changes in Visual Basic .NET

The **Shape** control in Visual Basic 6.0 has no equivalent in Visual Basic .NET. During upgrade, **Shape** controls are replaced with the Windows Forms **Label** control, with the **BorderStyle** property set to **FixedSingle**, and the **BackColor**, **Width**, and **Height** properties set to match the original control.

> **Note** It may be necessary to use the **ZOrder** method to achieve the same visual layout as the original application.

Shape controls with the **Shape** property set to 2-Oval or 3-Circle cannot be upgraded and are replaced with a **Label** control as a placeholder. You can replace the **Shape** control using the graphics functions built into the .NET Framework. For more information, see Graphics Changes in Visual Basic .NET.

See Also

Graphics Changes in Visual Basic .NET

TextBox Control Changes in Visual Basic .NET

The **TextBox** control in Visual Basic 6.0 is replaced by the Windows Forms **TextBox** control in Visual Basic .NET. The names of some properties, methods, events, and constants are different, and in some cases there are differences in behavior.

The following tables list Visual Basic 6.0 properties, methods, and events and their Visual Basic .NET equivalents. Where applicable, constants are indented beneath the property or method. All Visual Basic .NET constants map to the **System.Windows.Forms** namespace unless otherwise noted.

Links are provided as necessary to topics explaining differences in behavior. Where there is no direct equivalent in Visual Basic .NET, links are provided to topics that present alternatives.

TextBox Properties

Visual Basic 6.0	Visual Basic .NET Equivalent
Alignment **0 - Left Justify** **1 - Right Justify** **2 - Center**	**TextAlign** **HorizontalAlignment.Left** **HorizontalAlignment.Right** **HorizontalAlignment.Center**
Appearance	No equivalent. For more information, see Appearance and BorderStyle Property Changes in Visual Basic .NET.
BackColor	**BackColor** **Note** Colors are handled differently in Visual Basic .NET. For more information, see Color Behavior Changes in Visual Basic .NET. For a list of constants, see Color Changes in Visual Basic .NET.
BorderStyle **0 – None** **1 – Fixed Single**	**BorderStyle** **BorderStyle.None** **BorderStyle.FixedSingle**
CausesValidation	**CausesValidation**
Container	**Parent**
DataChanged	No equivalent. For more information, see Data Changes in Visual Basic .NET.
DataField	No equivalent. For more information, see Data Changes in Visual Basic .NET.
DataFormat	No equivalent. For more information, see Data Changes in Visual Basic .NET.
DataMember	No equivalent. For more information, see Data Changes in Visual Basic .NET.
DataSource	No equivalent. For more information, see Data Changes in Visual Basic .NET.
DragIcon	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
DragMode	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
Enabled	**Enabled**
Font	**Font** **Note** Fonts are handled differently in Visual Basic .NET. For more information, see Font Changes in Visual Basic .NET.

FontBold	**Font** **Note** Fonts are handled differently in Visual Basic .NET. For more information, see Font Changes in Visual Basic .NET.
FontItalic	**Font** **Note** Fonts are handled differently in Visual Basic .NET. For more information, see Font Changes in Visual Basic .NET.
FontName	**Font** **Note** Fonts are handled differently in Visual Basic .NET. For more information, see Font Changes in Visual Basic .NET.
FontSize	**Font** **Note** Fonts are handled differently in Visual Basic .NET. For more information, see Font Changes in Visual Basic .NET.
FontStrikethru	**Font** **Note** Fonts are handled differently in Visual Basic .NET. For more information, see Font Changes in Visual Basic .NET.
FontUnderline	**Font** **Note** Fonts are handled differently in Visual Basic .NET. For more information, see Font Changes in Visual Basic .NET.
ForeColor	**ForeColor** **Note** Colors are handled differently in Visual Basic .NET. For more information, see Color Behavior Changes in Visual Basic .NET. For a list of constants, see Color Changes in Visual Basic .NET.
Height	**Height, Size** **Note** Coordinates are handled differently in Visual Basic .NET. For more information, see Coordinate System Changes in Visual Basic .NET.
HelpContextID	No equivalent. For more information, see Help Support Changes in Visual Basic .NET.
HideSelection	**HideSelection**
HWnd	**Handle**
Index	No equivalent. For more information, see Control Array Changes in Visual Basic .NET.
Left	**Left** **Note** Coordinates are handled differently in Visual Basic .NET. For more information, see Coordinate System Changes in Visual Basic .NET.
LinkItem	No equivalent. For more information, see Dynamic Data Exchange Changes in Visual Basic .NET.
LinkMode	No equivalent. For more information, see Dynamic Data Exchange Changes in Visual Basic .NET.
LinkTimeOut	No equivalent. For more information, see Dynamic Data Exchange Changes in Visual Basic .NET.
LinkTopic	No equivalent. For more information, see Dynamic Data Exchange Changes in Visual Basic .NET.
Locked	**ReadOnly**
MaxLength	**MaxLength** **Note** In Visual Basic 6.0 it was not possible to programmatically set the **Text** property to a length greater than the **MaxLength** value; in Visual Basic .NET this is now possible.

MouseIcon	No equivalent. For more information, see Cannot set a custom MousePointer.
MousePointer	**Cursor** For a list of constants, see MousePointer Changes in Visual Basic .NET.
MultiLine	**MultiLine**
Name	**Name**
OLEDragMode	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
OLEDropMode	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
Parent	**FindForm** method
PasswordChar	**PasswordChar** **Note** In Visual Basic 6.0 the **PasswordChar** property was of type String; in Visual Basic .NET it is of type Char.
RightToLeft **True** **False**	**RightToLeft** **RightToLeft.Yes** **RightToLeft.No**
ScrollBars **0 – None** **1 – Horizontal** **2 – Vertical** **3 – Both**	**ScrollBars** **ScrollBars.None** **ScrollBars.Horizontal** **ScrollBars.Vertical** **ScrollBars.Both**
SelLength	**SelectionLength**
SelStart	**SelectionStart**
SelText	**SelectedText**
TabIndex	**TabIndex**
TabStop	**TabStop**
Tag	**Tag**
Text	**Text**
ToolTipText	**ToolTip** component For more information, see ToolTip Changes in Visual Basic .NET.
Top	**Top** **Note** Coordinates are handled differently in Visual Basic .NET. For more information, see Coordinate System Changes in Visual Basic .NET.
Visible	**Visible**
WhatsThisHelpID	No equivalent. For more information, see Help Support Changes in Visual Basic .NET.
Width	**Width, Size** **Note** Coordinates are handled differently in Visual Basic .NET. For more information, see Coordinate System Changes in Visual Basic .NET.

TextBox Methods

Visual Basic 6.0	Visual Basic .NET Equivalent
Drag	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
LinkExecute	No equivalent. For more information, see Dynamic Data Exchange Changes in Visual Basic .NET.

LinkPoke	No equivalent. For more information, see Dynamic Data Exchange Changes in Visual Basic .NET.
LinkRequest	No equivalent. For more information, see Dynamic Data Exchange Changes in Visual Basic .NET.
LinkSend	No equivalent. For more information, see Dynamic Data Exchange Changes in Visual Basic .NET.
Move	**SetBounds** **Note** Coordinates are handled differently in Visual Basic .NET. For more information, see Coordinate System Changes in Visual Basic .NET.
OLEDrag	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
Refresh	**Refresh**
SetFocus	**Focus**
ShowWhatsThis	No equivalent. For more information, see Help Support Changes in Visual Basic .NET.
ZOrder **0 – vbBringToFront** **1 - vbSendToBack**	**BringToFront()** or **SendToBack()** functions **BringToFront()** **SendToBack()**

TextBox Events

Visual Basic 6.0	Visual Basic .NET Equivalent
Change	**TextChanged**
Click	No equivalent. Use **MouseUp** as an alternative.
DblClick	**DoubleClick**
DragDrop	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
DragOver	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
GotFocus	**Enter**
KeyDown	**KeyDown**
KeyPress	**KeyPress**
KeyUp	**KeyUp**
LinkClose	No equivalent. For more information, see Dynamic Data Exchange Changes in Visual Basic .NET.
LinkError	No equivalent. For more information, see Dynamic Data Exchange Changes in Visual Basic .NET.
LinkNotify	No equivalent. For more information, see Dynamic Data Exchange Changes in Visual Basic .NET.
LinkOpen	No equivalent. For more information, see Dynamic Data Exchange Changes in Visual Basic .NET.
LostFocus	**Leave**
MouseDown	**MouseDown**
MouseMove	**MouseMove**
MouseUp	**MouseUp**
OLECompleteDrag	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
OLEDragDrop	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
OLEDragOver	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
OLEGiveFeedback	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
OLESetData	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
OLEStartDrag	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
Validate	**Validating**

See Also

Introduction to the Windows Forms TextBox Control

Timer Control Changes in Visual Basic .NET

The **Timer** control in Visual Basic 6.0 is replaced by the Windows Forms **Timer** component in Visual Basic .NET. The **Timer** component is placed in the tray instead of on the form at design time, and the behavior of the **Interval** property has changed. For more information, see Timer Interval property behavior has changed.

The following table lists Visual Basic 6.0 properties and events and their Visual Basic .NET equivalents.

Timer Properties and Events

Visual Basic 6.0	**Visual Basic .NET Equivalent**
Enabled property	**Enabled**
Index property	No equivalent property. The **Timer** is a component and cannot be a part of a control array.
Interval property	**Interval** **Note** The behavior of the **Interval** property has changed. For more information, see Timer Interval property behavior has changed.
Name property	No equivalent property. For more information, see Name Property Changes in Visual Basic .NET.
Parent property	No equivalent property. The **Timer** is a component and cannot have a parent.
Tag property	No equivalent. For more information, see Tag Property Changes in Visual Basic .NET.
Timer event	**Tick**

See Also

Introduction to the Windows Forms Timer Component

VBControlExtender Object Changes in Visual Basic .NET

In Visual Basic 6.0, the **VBControlExtender** object was used to provide access to properties of a host form to an ActiveX control. In Visual Basic .NET, the forms architecture is different; an extender is no longer needed.

The following tables list Visual Basic 6.0 properties, methods, and events and their Visual Basic .NET equivalents. Where applicable, constants are indented beneath the property or method. All Visual Basic .NET constants map to the **System.Windows.Forms** namespace unless otherwise noted.

Links are provided as necessary to topics explaining differences in behavior. Where there is no direct equivalent in Visual Basic .NET, links are provided to topics that present alternatives.

VBControlExtender Properties

Visual Basic 6.0	Visual Basic .NET Equivalent
Align **0 – None** **1 – Top** **2 – Bottom** **3 – Left** **4 – Right**	**Dock** **DockStyle.None** **DockStyle.Top** **DockStyle.Bottom** **DockStyle.Left** **DockStyle.Right**
Cancel	No equivalent.
CausesValidation	**CausesValidation**
Container	**Parent**
DataBindings	No equivalent. For more information, see Data Changes in Visual Basic .NET.
DataChanged	No equivalent. For more information, see Data Changes in Visual Basic .NET.
DataField	No equivalent. For more information, see Data Changes in Visual Basic .NET.
DataFormat	No equivalent. For more information, see Data Changes in Visual Basic .NET.
DrawMember	No equivalent. For more information, see Data Changes in Visual Basic .NET.
DataSource	No equivalent. For more information, see Data Changes in Visual Basic .NET.
Default	**AcceptButton**
DragIcon	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
DragMode	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
Enabled	**Enabled**
Height	**Height**, **Size** **Note** Coordinates are handled differently in Visual Basic .NET. For more information, see Coordinate System Changes in Visual Basic .NET.
HelpContextID	No equivalent. For more information, see Help Support Changes in Visual Basic .NET.
Index	No equivalent. For more information, see Control Array Changes in Visual Basic .NET.
Left	**Left** **Note** Coordinates are handled differently in Visual Basic .NET. For more information, see Coordinate System Changes in Visual Basic .NET.
Name	**Name**
Parent	**FindForm** method
TabIndex	**TabIndex**
TabStop	**TabStop**
Tag	**Tag**

ToolTipText	**ToolTip** component For more information, see ToolTip Changes in Visual Basic .NET.
Top	**Top** **Note** Coordinates are handled differently in Visual Basic .NET. For more information, see Coordinate System Changes in Visual Basic .NET.
Visible	**Visible**
WhatsThisHelpID	No equivalent. For more information, see Help Support Changes in Visual Basic .NET.
Width	**Width**, **Size** **Note** Coordinates are handled differently in Visual Basic .NET. For more information, see Coordinate System Changes in Visual Basic .NET.

VBControlExtender Methods

Visual Basic 6.0	**Visual Basic .NET Equivalent**
Drag	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
Move	**SetBounds** **Note** Coordinates are handled differently in Visual Basic .NET. For more information, see Coordinate System Changes in Visual Basic .NET.
SetFocus	**Activate**
ShowWhatsThis	No equivalent. For more information, see Help Support Changes in Visual Basic .NET.
Zorder **0 – vbBringToFront** **1 - vbSendToBack**	**BringToFront()** or **SendToBack()** functions **BringToFront()** **SendToBack()** **Note** In Visual Basic 6.0, the **ZOrder** method is positioned the form relative to other forms in the application; in Visual Basic .NET, forms are positioned relative to all windows on the screen.

VBControlExtender Events

Visual Basic 6.0	**Visual Basic .NET Equivalent**
DragDrop	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
DragOver	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
GotFocus	**Enter**
LostFocus	**LostFocus**
Validate	**Validating**

See Also

Adding ActiveX Controls to Windows Forms | System.Windows.Forms Namespace

VScrollBar Control Changes in Visual Basic .NET

The **VScrollBar** control in Visual Basic 6.0 is replaced by the Windows Forms **VScrollBar** control in Visual Basic .NET. The names of some properties, methods, events, and constants are different, and in some cases there are differences in behavior.

The following tables list Visual Basic 6.0 properties, methods, and events and their Visual Basic .NET equivalents. Where applicable, constants are indented beneath the property or method. All Visual Basic .NET constants map to the **System.Windows.Forms** namespace unless otherwise noted.

Links are provided as necessary to topics explaining differences in behavior. Where there is no direct equivalent in Visual Basic .NET, links are provided to topics that present alternatives.

VScrollBar Properties

Visual Basic 6.0	Visual Basic .NET Equivalent
CausesValidation	**CausesValidation**
Container	**Parent**
DragIcon	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
DragMode	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
Enabled	**Enabled**
Height	**Height, Size** **Note** Coordinates are handled differently in Visual Basic .NET. For more information, see Coordinate System Changes in Visual Basic .NET.
HelpContextID	No equivalent. For more information, see Help Support Changes in Visual Basic .NET.
HWnd	**Handle**
Index	No equivalent. For more information, see Control Array Changes in Visual Basic .NET.
LargeChange	**LargeChange** **Note** The default value in Visual Basic 6.0 was 1; in Visual Basic .NET the default value is 10.
Left	**Left** **Note** Coordinates are handled differently in Visual Basic .NET. For more information, see Coordinate System Changes in Visual Basic .NET.
Max	**Maximum** **Note** The default value in Visual Basic 6.0 was 32767; in Visual Basic .NET the default value is 100.
Min	**Minimum** **Note** In Visual Basic 6.0 **Min** could be set to a value greater than **Max**; in Visual Basic .NET this is not allowed.
MouseIcon	No equivalent. For more information, see Cannot set a custom MousePointer.
MousePointer	**Cursor** For a list of constants, see MousePointer Changes in Visual Basic .NET.
Name	**Name**
Parent	**FindForm** method
RightToLeft **True** **False**	**RightToLeft** **RightToLeft.Yes** **RightToLeft.No**

SmallChange	**SmallChange**
TabIndex	**TabIndex**
TabStop	**TabStop**
Tag	**Tag**
Top	**Top** **Note** Coordinates are handled differently in Visual Basic .NET. For more information, see Coordinate System Changes in Visual Basic .NET.
Value	**Value** **Note** In Visual Basic 6.0, when the **Value** was changed, the **Change** event was raised. In Visual Basic .NET the **Change** event is not raised; use the **ValueChanged** event instead.
Visible	**Visible**
WhatsThisHelpID	No equivalent. For more information, see Help Support Changes in Visual Basic .NET.
Width	**Width, Size** **Note** Coordinates are handled differently in Visual Basic .NET. For more information, see Coordinate System Changes in Visual Basic .NET.

VScrollBar Methods

Visual Basic 6.0	Visual Basic .NET Equivalent
Drag	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
Move	**SetBounds** **Note** Coordinates are handled differently in Visual Basic .NET. For more information, see Coordinate System Changes in Visual Basic .NET.
Refresh	**Refresh**
SetFocus	**Focus**
ShowWhatsThis	No equivalent. For more information, see Help Support Changes in Visual Basic .NET.
ZOrder **0 – vbBringToFront** **1 - vbSendToBack**	**BringToFront()** or **SendToBack()** functions **BringToFront()** **SendToBack()**

VScrollBar Events

Visual Basic 6.0	Visual Basic .NET Equivalent
Change	**ValueChanged** **Note** The behavior of the **Change** and **Scroll** events have changed in Visual Basic .NET. For more information, see Code was changed from an event to a procedure.
DragDrop	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
DragOver	No equivalent. For more information, see Drag and Drop Changes in Visual Basic .NET.
GotFocus	**Enter**
KeyDown	**KeyDown**
KeyPress	**KeyPress**
KeyUp	**KeyUp**
LostFocus	**Leave**

Scroll	**Scroll** **Note** The behavior of the **Change** and **Scroll** events have changed in Visual Basic .NET. For more inform ation, see Code was changed from an event to a procedure.
Validate	**Validating**

See Also

Introduction to the Windows Forms HScrollBar and VScrollBar Controls

Control Property, Method, and Event Changes in Visual Basic .NET

Many of the properties, methods, and events for Visual Basic 6.0 controls and objects are replaced by Visual Basic .NET properties, methods, and events with different behavior.

The following topics describe the changes for properties, methods, and events.

In This Section

Appearance and BorderStyle Property Changes in Visual Basic .NET
Describes changes to the **Appearance** and **BorderStyle** properties of various controls.
BackStyle Property Changes in Visual Basic .NET
Describes changes to the **BackStyle** property of the **Label** control.
Change Event Changes in Visual Basic .NET
Describes changes to the **Change** event of the **ComboBox** control.
Coordinate System Changes in Visual Basic .NET
Describes changes to the **Height** and **Width** properties and the **Move** method.
Drag and Drop Changes in Visual Basic .NET
Describes changes to drag and drop properties, methods, and events.
Dynamic Data Exchange Changes in Visual Basic .NET
Describes changes to Dynamic Data Exchange (DDE).
Event Changes in Visual Basic .NET
Describes changes to event handling.
Font Changes in Visual Basic .NET
Describes changes to font handling and font properties.
Form Event Changes in Visual Basic .NET
Describes changes to various form events.
Graphics Changes in Visual Basic .NET
Describes changes to graphics properties, methods, and events.
Locked Property Changes in Visual Basic .NET
Describes changes to the **Locked** property of the **ComboBox** control.
MaxLength Property Changes in Visual Basic .NET
Describes changes to the **MaxLength** property of the **TextBox** control.
MousePointer Changes in Visual Basic .NET
Describes changes to the **MousePointer** property and constants.
Moveable Property Changes in Visual Basic .NET
Describes changes to the **Moveable** property of a form.
Name Property Changes in Visual Basic .NET
Describes changes to the **Name** property of **Timer** or **Menu** controls.
PrevInstance Property Changes in Visual Basic .NET
Describes changes to the **PrevInstance** property of the **App** object.
Style Property Changes in Visual Basic .NET
Describes changes to the **Style**, **Picture**, **Disabled Picture**, and **DownPicture** properties of various controls.
Tag Property Changes in Visual Basic .NET
Describes changes to the **Tag** property of **Timer** and **Menu** controls.
TaskVisible Property Changes in Visual Basic .NET
Describes changes to the **TaskVisible** property of the **App** object.
ToolTip Changes in Visual Basic .NET
Describes changes to the **ToolTip** property of various controls.
TopIndex Property and Scroll Event Changes in Visual Basic .NET
Describes changes to the **TopIndex** property and the **Scroll** event of **ComboBox** and **ListBox** controls.
Version Numbering Changes in Visual Basic .NET
Describes changes to the **Version** property of the **App** object.

Related Sections

Color Changes in Visual Basic .NET
Describes changes to color constants.
Constant Changes in Visual Basic .NET

Describes changes to various Visual Basic constants.

Control Changes in Visual Basic .NET

Describes changes to controls.

Data Changes in Visual Basic .NET

Describes changes to data access and data tools.

Forms Task Changes in Visual Basic .NET

Describes changes to forms.

Help Support Changes in Visual Basic .NET

Describes changes to help-related properties.

Appearance and BorderStyle Property Changes in Visual Basic .NET

In Visual Basic 6.0, the **Appearance** and/or the **BorderStyle** properties were used to change the appearance of controls. In Visual Basic .NET, the **Appearance** property no longer exists, and the **BorderStyle** property has an additional option.

Visual Basic 6.0 forms and the Visual Basic 6.0 **Frame**, **Image**, **Label**, **PictureBox**, and **TextBox** controls supported both the **Appearance** and **BorderStyle** properties. During upgrade, the properties are mapped as follows:

Visual Basic 6.0	Visual Basic .NET
Appearance = 0 – Flat **BorderStyle = 0 – None**	**BorderStyle = BorderStyle.None**
Appearance = 0 – Flat **BorderStyle = 1 – Fixed Single**	**BorderStyle = BorderStyle.FixedSingle**
Appearance = 1 – 3D **BorderStyle = 0 – None**	**BorderStyle = BorderStyle.None**
Appearance = 1 – 3D **BorderStyle = 1 – Fixed Single**	**BorderStyle = BorderStyle.Fixed3D**

> **Note** The VisualBasic .NET **BorderStyle** constants map to the **System.Windows.Forms** namespace.

For the **Frame** control, the mappings apply only to **Frame** controls that are upgraded to **Panel** controls; there is no **BorderStyle** property for **GroupBox** controls. For more information, see Frame Control Changes in Visual Basic .NET.

The Visual Basic 6.0 **ComboBox**, **CommandButton**, and **ListBox** controls supported only the **Appearance** property. There is no equivalent for the **Appearance** property in Visual Basic .NET. During upgrade, the **Appearance** property is not mapped; if the **Appearance** property is set in code, an upgrade warning will be added.

> **Note** The Visual Basic .NET **Button** control supports the **FlatStyle** property that can be used to provide a flat appearance. The **ListBox** control can be made to appear flat by setting the **BorderStyle** property to **FixedSingle**. The **ComboBox** control never actually supported a flat appearance in Visual Basic 6.0, so there is no equivalent.

See Also

ComboBox Control Changes in Visual Basic .NET | CommandButton Control Changes in Visual Basic .NET | Form Object Changes in Visual Basic .NET | Frame Control Changes in Visual Basic .NET | Label Control Changes in Visual Basic .NET | PictureBox Control Changes in Visual Basic .NET | TextBox Control Changes in Visual Basic .NET | ButtonBase.FlatStyle Property | UserControl Changes in Visual Basic .NET

Change Event Changes in Visual Basic .NET

In Visual Basic 6.0, the **Change** event of the **ComboBox** control was raised when the text in the text-entry portion of the control was modified; it was not raised when an item was selected from the list portion of the control. Programmatically changing the text of an item in the list did not raise the **Change** event.

When a project is migrated to Visual Basic .NET, the **Change** event of the **ComboBox** control is mapped to the **TextChanged** event of the Visual Basic .NET **ComboBox** control. The behavior of the **TextChanged** event differs from that of the **Change** event; this difference may cause unintended results in your code.

The Visual Basic .NET **TextChanged** event is raised every time the text changes for any reason: when the text-entry portion is modified, when an item is selected from the list, when a list item is modified programmatically, and when the **AddItem** method is called.

The following example illustrates the behavior differences:

```
' Visual Basic 6.0
Private Sub Form_Load()
   ' Does not raise the Change event.
   Combo1.AddItem "A"
End Sub
Private Sub Form_Click()
   ' Does not raise the Change event.
   Combo1.List(0) = "B"
End If

' Visual Basic .NET
Private Sub New()
   ' Raises the TextChanged event.
   ComboBox1.Items.Add "A"
End Sub
Protected Sub Form1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs))
   ' Uses the SelectItem string method from the VB6 compatibility library
   ' because there is no equivalent method in Visual Basic .NET.
   ' Raises the TextChanged event.
   VB6.SetItemString(ComboBox1, 0, "B")
End Sub
```

What to do next

- Set a breakpoint in the **TextChanged** event procedure and run your code to determine where it is being raised. Modify your code as necessary.

See Also

ComboBox Control Changes in Visual Basic .NET | UserControl Changes in Visual Basic .NET

Coordinate System Changes in Visual Basic .NET

In Visual Basic 6.0, coordinates for forms and controls were expressed in twips; in Visual Basic .NET, coordinates are expressed in pixels. During upgrade the **Height** and **Width** properties of forms and controls are automatically converted to pixels.

Visual Basic .NET also provides a **Size** property that allows you to change both height and width in a single step. The following example shows how to change the **Size** property of a form at run time:

```
' Resizes the form to 640 pixels wide by 480 pixels high.
Me.Size = New System.Drawing.Size(640, 480)
```

Visual Basic 6.0 used the **Move** method to change the location of a form or control at run time. In Visual Basic .NET, **Move** is replaced by **SetBounds**; the coordinates are expressed in pixels. The following example demonstrates using **SetBounds** to move controls:

```
' Visual Basic 6.0
' Move and resize to 1200 by 800 twips.
CommandButton1.Move 0, 0, 1200, 800
' Move and retain original size.
CommandButton2.Move 2000, 1000
```

Visual Basic .NET equivalent

```
' Visual Basic .NET
' Move and resize to 120 by 80 pixels.
button1.SetBounds(0, 0, 120, 80)
' Move and retain original size.
button2.SetBounds(20, 10, 0, 0, BoundsSpecified.X Or BoundsSpecified.Y)
```

Visual Basic 6.0 also provided a **ScaleMode** property that allowed you to define a different coordinate system. **ScaleMode** and its related properties and methods (**Scale**, **ScaleHeight**, **ScaleLeft**, **ScaleTop**, **ScaleWidth**, **ScaleX**, **ScaleY**) are not supported in Visual Basic .NET. For more information, see ScaleMode is not supported.

See Also

Control Changes in Visual Basic .NET | UserControl Changes in Visual Basic .NET

Drag and Drop Changes in Visual Basic .NET

In Visual Basic 6.0, drag and drop could be accomplished by two different methods: standard drag and drop for dragging between controls on a form, and OLE drag and drop for dragging between forms and applications.

In Visual Basic .NET, a single model for drag and drop is used. It is similar to OLE drag and drop, but the names and parameters for drag-and-drop methods and events are different, and the event model is different. Drag-and-drop code cannot automatically be upgraded to Visual Basic .NET; it must be rewritten using the new model.

The following example shows the differences when dragging text from one **TextBox** control to another:

```
' Visual Basic 6.0
Private Sub Text1_MouseDown(Button As Integer, Shift As Integer, X As Single, Y As Single)
   Text1.Text = "Hello World"
   ' Begin dragging by calling the OLEDrag method.
   Text1.OLEDrag
End Sub

Private Sub Text1_OLEStartDrag(Data As DataObject, AllowedEffects As Long)
   ' Only allow copying.
   AllowedEffects = vbDropEffectCopy
   Data.Clear
   ' Populate the Data object with the text to copy and the format.
   Data.SetData Text1.Text, vbCFText
End Sub

Private Sub Text2_OLEDragOver(Data As DataObject, Effect As Long, Button As Integer, Shift As
Integer, X As Single, Y As Single, State As Integer)
   ' Make sure that the format is text.
   If Data.GetFormat(vbCFText) Then
      ' If it is text, enable dropping for the second TextBox.
      Text2.OLEDropMode = vbOLEDropManual
    End If
End Sub

Private Sub Text2_OLEDragDrop(Data As DataObject, Effect As Long, Button As Integer, Shift As
Integer, X As Single, Y As Single)
   ' Copy the text from the Data object to the second TextBox.
   Text2.Text = Data.GetData(vbCFText)
End Sub

' Visual Basic .NET
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles My
Base.Load
   ' Dropping must be enabled before the dragging occurs.
   TextBox2.AllowDrop = True
End Sub

Private Sub TextBox1_MouseDown(ByVal sender As Object, ByVal e As System.Windows.Forms.MouseE
ventArgs) Handles TextBox1.MouseDown
   TextBox1.Text = "Hello World"
   ' Begin dragging by calling the DoDragDrop method.
   ' OLEStartDrag is replaced by arguments on the method.
   TextBox1.DoDragDrop(TextBox1.Text, DragDropEffects.Copy)
End Sub

Private Sub TextBox2_DragEnter(ByVal sender As Object, ByVal e As System.Windows.Forms.DragEv
entArgs) Handles TextBox2.DragEnter
' Make sure that the format is text.
If (e.Data.GetDataPresent(DataFormats.Text)) Then
      ' Allow drop.
      e.Effect = DragDropEffects.Copy
   Else
      ' Do not allow drop.
      e.Effect = DragDropEffects.None
   End If
End Sub
```

```
Private Sub TextBox2_DragDrop(ByVal sender As Object, ByVal e As System.Windows.Forms.DragEve
ntArgs) Handles TextBox2.DragDrop
   ' Copy the text to the second TextBox.
   TextBox2.Text = e.Data.GetData(DataFormats.Text).ToString
End Sub
```

See Also

Drag-and-Drop Operations and Clipboard Support

Dynamic Data Exchange Changes in Visual Basic .NET

Dynamic Data Exchange (DDE) was an early technology that enabled the exchange of information between applications in Visual Basic 3.0. DDE was replaced by OLE Automation in later versions of Visual Basic; however, DDE was still supported for backward compatibility through version 6.0.

With the introduction of Visual Basic .NET, DDE is no longer supported. Visual Basic applications that need to exchange information with legacy applications that only support DDE should be maintained in Visual Basic 6.0.

The following DDE properties, methods, and events are no longer available:

Properties

LinkItem

LinkMode

LinkTimeOut

LinkTopic

Methods

LinkExecute

LinkPoke

LinkRequest

LinkSend

Events

LinkClose

LinkError

LinkExecute

LinkNotify

LinkOpen

In addition, the **Clipboard** object constant **vbCFLink** is no longer valid.

See Also

Form Object Changes in Visual Basic .NET | Label Control Changes in Visual Basic .NET | PictureBox Control Changes in Visual Basic .NET | TextBox Control Changes in Visual Basic .NET | Clipboard Object Changes in Visual Basic .NET | UserControl Changes in Visual Basic .NET

Event Changes in Visual Basic .NET

In Visual Basic 6.0, events were tied to specific objects and had their own event-handling code. For example, on a form with a button and a menu, each had its own **Click** event; you had to write code in the event handler for each even if they both performed exactly the same function.

In Visual Basic .NET, events are tied to event handlers via delegates, allowing you to create a single event handler for multiple objects. For more information, see Introduction to Event Handlers in Windows Forms.

In addition, the names and behavior of some events for various objects are different in Visual Basic .NET. For more information, see Control Changes in Visual Basic .NET.

See Also

Event Handling in Windows Forms

Font Changes in Visual Basic .NET

Fonts in Visual Basic 6.0 were handled in two different ways: as font properties of forms and controls, or as a **stdFont** object. In Visual Basic .NET, there is a single **Font** object: **System.Drawing.Font**.

Setting Font Properties

In Visual Basic 6.0, font properties could be set at run time either by assigning a **stdFont** object or by setting the properties directly:

```
' Visual Basic 6.0
' Uses the stdFont object.
Dim f As New stdFont
' Sets the stdFont object to the Arial font.
f.Name = "Arial"
' Assigns the stdFont to the font property.
Set Label1.Font = f
' Can still change properties at run time.
Label1.FontBold = True
Label1.Font.Italic = True
```

In Visual Basic .NET, fonts are read-only at run time — you cannot set the properties directly, but you can assign a new **Font** object:

```
' Visual Basic .NET
' Fails - property is read-only.
label1.Font = Arial
' Assign a Font object — Name and Size are required.
label1.Font = New System.Drawing.Font("Arial", 10)
' Assign additional attributes.
label1.Font = New System.Drawing.Font(label1.Font, FontStyle.Bold __
    Or FontStyle.Italic)
```

Font Inheritance

In Visual Basic 6.0, font properties had to be set individually for each control or form. In Visual Basic .NET, font properties are automatically inherited from their parent unless they are explicitly set for the child object. For example, if you had two label controls on a form and changed the font properties of the form to Arial, the label control's fonts would also change to Arial. If you subsequently changed the font of one label to Times Roman, further changes to the form's font would not override the label's font.

Font Compatibility

Visual Basic 6.0 supported raster fonts for backward compatibility; Visual Basic .NET supports only TrueType and OpenType fonts. During upgrade, raster fonts are converted to the default OpenType font, Microsoft Sans Serif. Formatting such as Bold or Italic is not preserved. For more information, see Only OpenType and TrueType fonts are supported.

Enumerating Fonts

In Visual Basic 6.0, you could use the **Screen.Fonts** collection along with **Screen.FontCount** property to enumerate the available screen fonts.

In Visual Basic .NET, the **Screen** object no longer exists; in order to enumerate available fonts on the system you should use the **System.Drawing.FontFamily** namespace.

```
' Visual Basic 6.0
Dim i As Integer
For i = 0 To Screen.FontCount - 1
   List1.AddItem Screen.Fonts(i)
Next i

' Visual Basic .NET
Dim ff As FontFamily
```

```
For Each ff In System.Drawing.FontFamily.Families
   listBox1.Items.Add(ff.Name)
Next
```

Note Visual Basic 6.0 enumerated all types of fonts. Visual Basic .NET only supports TrueType and OpenType fonts; other font types will not be enumerated. In addition, Visual Basic 6.0 enumerated each character set version within a font family (for instance, Arial, Arial Baltic, Arial Greek); Visual Basic .NET enumerates the font families.

See Also

Font Class | FontFamily.Families Property | Control Changes in Visual Basic .NET | UserControl Changes in Visual Basic .NET

Form Event Changes in Visual Basic .NET

There are some differences in form event behavior between Visual Basic 6.0 and Visual Basic .NET.

In Visual Basic 6.0, the **Initialize** event was used to execute code before a form was loaded. In Visual Basic .NET, initialization code must be added to the form constructor (**Sub New()**) after the call to **InitializeComponent()** as in the following example:

```
' Visual Basic 6.0
Private Sub Form_Initialize()
    frmSplash.Show
End Sub

' Visual Basic .NET
Public Sub New()
    MyBase.New()
    ' This call is required by the Windows Forms Designer.
    InitializeComponent()
    ' Add your initialization code here.
    FrmSplash.Show()
End Sub
```

In Visual Basic 6.0, the **Terminate** event was used to execute code after a form was unloaded. In Visual Basic .NET, the **Terminate** event is no longer supported. Termination code must be executed inside the **Dispose** method, before the call to **MyBase.Dispose()**.

> **Note** The **Dispose** method is called automatically for the main form in an application; you need to call it explicitly for any other form.

The following example demonstrates the differences. Note that a static variable is used to prevent the termination code from executing twice — this is necessary because the **Dispose** method is public and might be called outside of the form's code.

```
' Visual Basic 6.0
Private Sub Form_Terminate()
    MsgBox "The form was terminated"
End Sub

' Visual Basic .NET
Public Overrides Sub Dispose(ByVal Disposing As Boolean)

    Static fTerminated As Boolean
    If Not fTerminated Then
        Form_Terminate
        fTerminated = True
    End If
    ' The following code was automatically added by the Dispose method.
    MyBase.Dispose(Disposing)
    If Disposing Then
        If Not components Is Nothing Then
            Components.Dispose(Disposing)
        End If
    End If
End Sub
```

In Visual Basic 6.0, the **Unload** event had a Cancel argument; in Visual Basic .NET, it is replaced by the **Closed** event, which does not have a Cancel argument. If you need to cancel during unloading, use the **Closing** event instead.

In Visual Basic 6.0, MDI Forms supported mouse events. In Visual Basic .NET, the **Click**, **MouseDown**, **MouseMove**, and **MouseUp** events are no longer supported for MDI Forms, since the MDI Form has no client area to receive mouse events.

See Also

Form Object Changes in Visual Basic .NET | Form Constructor | Form.Closing Event

Graphics Changes in Visual Basic .NET

In Visual Basic 6.0, various graphics methods and properties were used to draw on a form or **PictureBox** control. Graphics in Visual Basic 6.0 were based on Windows Graphics Device Interface (GDI) APIs.

In Visual Basic .NET, graphics are provided by the **System.Drawing** namespace, which encapsulates the new GDI+ APIs. GDI+ expands upon the graphics capabilities of Visual Basic 6.0, but the methods are not compatible. When an application is upgraded from Visual Basic 6.0, graphics methods are not upgraded.

The following example shows the code used to draw a circle in the upper left corner of a form using Visual Basic 6.0 and Visual Basic .NET:

```
' Visual Basic 6.0
Private Sub Form_Click()
   ' Set the ScaleMode to pixels.
   Me.ScaleMode = 3
   Me.Circle (50, 50), 50, vbRed
End Sub

' Visual Basic .NET
Private Sub Form1_Click(ByVal sender As Object, ByVal e As System.EventArgs) Handles MyBase.C
lick
   ' Create a Pen object.
   Dim pen As New Drawing.Pen(System.Drawing.Color.Red, 1)
   Me.CreateGraphics.DrawEllipse(pen, 0, 0, 100, 100)
   Pen.Dispose
End Sub
```

The following table lists Visual Basic 6.0 graphics properties and methods and their Visual Basic .NET equivalents.

Visual Basic 6.0	**Visual Basic .NET Equivalent**
AutoRedraw Property	No equivalent. To persist graphics, put graphics methods in the **Paint** event.
Circle Method	Graphics.DrawEllipse Method
ClipControls Property	No equivalent. To repaint only part of an object, create a clip in the **Paint** event using the Graphics.SetClip Method method.
Cls Method	Graphics.Clear Method
CurrentX Property	The ***x*** parameter of various graphics methods. For example, DrawRectangle(pen, ***x***, y, width, height)
CurrentY Property	The ***y*** parameter of various graphics methods. For example, DrawRectangle(pen, x, ***y***, width, height)
DrawMode Property	Pen.Color Property
DrawStyle Property	Pen.PenType Property
DrawWidth Property	Pen.Width Property
FillColor Property	SolidBrush.Color Property
FillStyle Property	Pen.Brush Property
HasDC Property	No equivalent. Device contexts are no longer necessary with GDI+.
HDC Property	No equivalent. Device contexts are no longer necessary with GDI+..
Image Property	**Image** Property. In Visual Basic 6.0 the **Image** property returned a handle; in Visual Basic .NET it returns a **System.Drawing.Image** object.
Line Method	Graphics.DrawLine Method
PaintPicture Method	Graphics.DrawImage Method
Point Method	No equivalent for forms or controls. For bitmaps, use Bitmap.GetPixel Method.
Print Method	Graphics.DrawString Method
Pset Method	No equivalent for forms or controls. For bitmaps, use Bitmap.SetPixel Method.

TextHeight Property	Font.Height Property
TextWidth Property	Graphics.MeasureString Method

See Also

Creating Graphical Images with GDI+

MaxLength Property Changes in Visual Basic .NET

In Visual Basic 6.0, the **MaxLength** property of a **TextBox** control had absolute control over the length of a string that could be entered or displayed in the text box. Although you could programmatically enter a longer string, it would automatically be truncated to the length set in the **MaxLength** property.

In Visual Basic .NET, the length of a string entered programmatically will override the **MaxLength** property setting.

The following code illustrates the different behavior.

```
' Visual Basic 6.0
Text1.MaxLength = 5
Text1.Text = "Longer than five"
MsgBox Text1.Text       ' Displays "Longe".

' Visual Basic .NET
Text1.MaxLength = 5
Text1.Text = "Longer than five"
MsgBox Text1.Text       ' Displays "Longer than five".
```

What to do next

- Review the design of your application to determine if the intended behavior was to always limit the text in the text box to the length set in the **MaxLength** property.
  - If this was the intended behavior, find each occurrence of code that sets the **Text** property and add code to truncate the string similar to the following example.

    ```
    ' Old code
    Text1.Text = SomeString

    ' Replacement code
    If Len(SomeString) > Text1.MaxLength Then
       SomeString = Left(SomeString, Text1.MaxLength)
    End If

    Text1.Text = SomeString
    ```

  - If it was not the intended behavior, leave the code alone. The user will still be limited by the **MaxLength** setting, but you can programmatically assign strings that are longer than the **MaxLength** setting.

See Also

TextBox Control Changes in Visual Basic .NET | UserControl Changes in Visual Basic .NET

MousePointer Changes in Visual Basic .NET

In Visual Basic .NET, the **MousePointer** property has been replaced by the **Cursor** property, and the names of the **MousePointer** constants have changed.

Visual Basic 6.0 allowed you to use a bitmap as a custom **MousePointer**; this is no longer supported in Visual Basic .NET. You can set a custom **Cursor** at run time by assigning a cursor file:

```
' Visual Basic .NET
button1.Cursor = New System.Windows.Forms.Cursor("C:\mypath\mycursor.cur")
```

In addition, the icon MousePointer (vbIconPointer) is obsolete and is replaced by the default cursor during upgrade.

MousePointer Constant Equivalents

Visual Basic 6.0	Visual Basic .NET Equivalent
0 – vbDefault	**System.Windows.Forms.Cursors.Default**
1 – vbArrow	**System.Windows.Forms.Cursors.Arrow**
2 – vbCrossHair	**System.Windows.Forms.Cursors.Cross**
3 – vbIBeam	**System.Windows.Forms.Cursors.IBeam**
4 – vbIconPointer	Obsolete — replaced by **System.Windows.Forms.Cursors.Default**
5 – vbSizePointer	**System.Windows.Forms.Cursors.SizeAll**
6 – vbSizeNESW	**System.Windows.Forms.Cursors.SizeNESW**
7 – vbSizeNS	**System.Windows.Forms.Cursors.SizeNS**
8 – vbSizeNWSE	**System.Windows.Forms.Cursors.SizeNWSE**
9 – vbSizeWE	**System.Windows.Forms.Cursors.SizeWE**
10 – vbUpArrow	**System.Windows.Forms.Cursors.UpArrow**
11 – vbHourGlass	**System.Windows.Forms.Cursors.WaitCursor**
12 – vbNoDrop	**System.Windows.Forms.Cursors.No**
13 – vbArrowHourGlass	**System.Windows.Forms.Cursors.AppStarting**
14 – vbArrowQuestion	**System.Windows.Forms.Cursors.Help**
15 – vbSizeAll	**System.Windows.Forms.Cursors.SizeAll**
99 – vbCustom	No equivalent — for more information, see Cannot set a custom MousePointer

See Also

Control Changes in Visual Basic .NET | UserControl Changes in Visual Basic .NET

Moveable Property Changes in Visual Basic .NET

In Visual Basic 6.0, the **Moveable** property of a form could be set to **false** to prevent a user from moving the form at run time. Windows Forms in Visual Basic .NET do not have an equivalent property.

Although it is generally considered bad user interface design, you can achieve similar behavior in Visual Basic .NET by setting the **Form.FormBorderStyle** property to **none** and the **Form.ControlBox** property to **false.**

See Also

Form Object Changes in Visual Basic .NET | MDIForm Object Changes in Visual Basic .NET | Form.ControlBox Property

Style Property Changes in Visual Basic .NET

In Visual Basic 6.0, setting the **Style** property of a **CommandButton**, **CheckBox** or **OptionButton** control to **1 – Graphical** changed the appearance of the control to display images. The **Picture**, **DownPicture**, and **DisabledPicture** properties were used to assign images to be displayed in response to state changes. For example, when a **CheckBox** control was checked, the **DownPicture** image was displayed; if the control was disabled, the **DisabledPicture** image was displayed.

Visual Basic .NET no longer supports the **Style**, **Picture**, **DownPicture**, or **DisabledPicture** properties. The **Style** property is replaced by the **Appearance** property, the **Picture** property is replaced by the **Image** property, and the functionality of the **DownPicture** and **DisabledPicture** properties can be emulated using an **ImageList** control.

When an application is upgraded to Visual Basic .NET, if the **Style** property is set to **1 – Graphical**, the **Appearance** property of the upgraded control is set to **Button**, and any image assigned to the **Picture** property at design time is assigned to the **Image** property of the upgraded control.

What to do next

- Check your Visual Basic 6.0 application. If the **DownPicture** and **DisabledPicture** properties are not being set at design time or at run time, the behavior should be the same in Visual Basic .NET.
- If the **DownPicture** or **DisabledPicture** properties are being set, take the following steps to modify the upgraded application:

  Note The following example uses a **CheckBox** control; the same steps apply for an **OptionButton** control (replaced by **RadioButton** control in Visual Basic .NET).

  - Determine the file names and locations of the images that were assigned to the **Picture**, **DownPicture**, and **DisabledPicture** properties and, if necessary, copy them to your development computer.
  - Add an **ImageList** control from the Toolbox to your form.
  - In the Properties window, select the **Images** property.
  - In the Image Collection Editor, add the images for the **Picture**, **DownPicture**, and **DisabledPicture** properties, in that order.
  - If any of the properties were set at run time, remove the code.
  - If the properties were set a design time, add the following code to the **Load** event for the form:

    ```
    ' Assign the first image (Picture) to the Image property.
    checkBox1.Image = imageList1.Images(0)
    ```

  - To display the **DownPicture** image at run time, add the following code to the **CheckChanged** event for the **CheckBox** control:

    ```
    If checkBox1.Checked = True Then
       ' Assign the second image (DownPicture) to the Image property.
       checkBox1.Image = imageList1.Images(1)
    Else
       ' Assign the first image (Picture) to the Image property.
       checkBox1.Image = imageList1.Images(0)
    End If
    ```

  - To display the **DisablePicture** image at run time, add the following code to the **EnabledChanged** event for the **CheckBox** control:

    ```
    If checkBox1.Enabled = False Then
       ' Assign the third image (DisabledPicture) to the Image property.
       checkBox1.Image = imageList1.Images(2)
    ElseIf checkBox1.Checked = True Then
       ' Assign the second image (DownPicture) to the Image property
       checkBox1.Image = imageList1.Images(1)
    Else
       ' Assign the first image (Picture)to the Image property
       checkBox1.Image = imageList1.Images(0)
    ```

```
End If
```

The application should now behave exactly as it did in Visual Basic 6.0.

See Also

CheckBox Control Changes in Visual Basic .NET | CommandButton Control Changes in Visual Basic .NET | OptionButton Control Changes in Visual Basic .NET | UserControl Changes in Visual Basic .NET

ToolTip Changes in Visual Basic .NET

In Visual Basic 6.0, the **ToolTipText** property of a control was used to display a ToolTip at run time. If the property contained text, the ToolTip was displayed; if it was empty, no ToolTip was displayed.

In Visual Basic .NET, a single **ToolTip** control can be used to control the ToolTips for all controls on a form. The **SetToolTip** method is used to set the text for each ToolTip.

The following example demonstrates setting a ToolTip in Visual Basic 6.0 and Visual Basic .NET:

```
' Visual Basic 6.0
Button1.ToolTipText = "Saves changes"

' Visual Basic .NET
' Adds a ToolTip control (ToolTip1) to the form.
ToolTip1.SetToolTip(Button1, "Save changes")
```

See Also

Control Changes in Visual Basic .NET

TopIndex Property and Scroll Event Changes in Visual Basic .NET

In Visual Basic 6.0, the **TopIndex** property was used to return or set a value that specifies which item in a **ComboBox** or **ListBox** control is displayed in the topmost position. This property was commonly used to scroll the list without selecting an item.

In Visual Basic .NET, the **ComboBox** control no longer supports the **TopIndex** property. In most cases this should not present a problem, because setting the **TopIndex** property had no visible effect unless the **Style** property was set to **1 - SimpleCombo**. In this case, you can emulate the **ComboBox** SimpleCombo behavior by using a **ListBox** control and a **TextBox** control; the **ListBox** control still supports the **TopIndex** property.

The **Scroll** event in Visual Basic 6.0 was used in conjunction with the **TopIndex** property to perform actions when the list was scrolled. Visual Basic .NET does not support the **Scroll** event; however, in most cases the **SelectedIndexChanged** event should be a suitable replacement.

See Also

ComboBox Control Changes in Visual Basic .NET | ListBox Control Changes in Visual Basic .NET | ComboBox.SelectedIndexChanged Event | ListBox.SelectedIndexChanged Event | UserControl Changes in Visual Basic .NET

Version Numbering Changes in Visual Basic .NET

In Visual Basic 6.0, setting the **Major**, **Minor**, and **Revision** properties of the **App** object controlled the version number for an application. For example, setting **Major** to 1, **Minor** to 2, and **Revision** to 3 resulted in a version number of 1.2.0.3.

In Visual Basic .NET, the version number properties are replaced by the **AssemblyVersion** attribute, which has four parts: **Major**, **Minor**, **Revision**, and **Build**. For the above example, the resulting version number would be 1.2.3.*x*, where *x* is a build number that is incremented each time the assembly is built.

When an application is upgraded from Visual Basic 6.0, only the **Major** and **Mino**r properties are upgraded; Visual Basic .NET assigns new values for the **Revision** and **Build** properties.

See Also

App Object Changes in Visual Basic .NET | Attributes

Control Array Changes in Visual Basic .NET

In Visual Basic 6.0, control arrays could be used to specify a group of controls that shared a set of events. The controls had to be of the same type, and they had to have the same name.

In Visual Basic .NET, control arrays are no longer supported. Changes to the event model make control arrays unnecessary. Just as control arrays in Visual Basic 6.0 could share events, the event model in Visual Basic .NET allows any event handler to handle events from multiple controls. In effect, this allows you to create groups of controls of disparate types that share the same events.

For example, you might add two **Button** controls (Button1 and Button2) and a **CheckBox** control (CheckBox1) to a form, then create an event handler to handle the **Click** event for all three controls:

```
Private Sub MixedControls_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) H
andles Button1.Click, Button2.Click, CheckBox1.Click
```

Another feature of Visual Basic 6.0 control arrays was the ability to reference a control by its **Index** property. Although Visual Basic .NET controls do not have an **Index** property, you can duplicate the functionality using another common property such as the **TabIndex** or **Tag** property.

For example, you might set the **TabIndex** property for a group of controls using the new visual tab ordering capabilities of Windows Forms, and then use the **TabIndex** in a **Select Case** statement:

```
Private Sub MixedControls_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) H
andles Button1.Click, Button2.Click, CheckBox1.Click
   Select Case sender.TabIndex
      Case 0
         MsgBox("Button 1")
      Case 1
         MsgBox("Button 2")
      Case 2
         MsgBox("CheckBox 1")
   End Select
End Sub
```

See Also

Event Changes in Visual Basic .NET | Control Changes in Visual Basic .NET | Setting the Tab Order on Windows Forms

Form Positioning Changes in Visual Basic .NET

In Visual Basic 6.0, the initial screen location for a form could be set at design time using the Form Layout window or by setting the **Left** and **Top** properties in the Properties window. The **Left** and **Top** properties could also be set programmatically.

In Visual Basic .NET, the initial screen location is set at design time by setting the **Location** property, or programmatically by setting the **Location** property to a **Point** object.

In addition, Visual Basic 6.0 supported a **StartUpPosition** property that could be used to allow Windows to automatically set a location (for example, centered on the screen). The default setting was **Manual**, meaning that the **Left** and **Top** properties took precedence.

In Visual Basic .NET, the **StartPosition** property performs the same function. The default setting is now **WindowsDefaultLocation**, meaning that the **Location** property is ignored.

See Also

Setting the Screen Location of Windows Forms

Menu Changes in Visual Basic .NET

In Visual Basic 6.0, menus were created using the Menu Editor. Menus could not be created programmatically, but could be changed or added to programmatically.

In Visual Basic .NET, menus are created using **MainMenu** controls, which provide a graphical menu designer. Menus can also be created programmatically. For more information, see Menus in Windows Forms.

In Visual Basic 6.0, context menus were created by calling the **PopupMenu** method of a form or control. They could not be created programmatically; the **PopupMenu** method required that a menu already existed.

In Visual Basic .NET, context menus are created using **ContextMenu** controls; they can also be created programmatically. For more information, see Context Menus on Windows Forms.

In Visual Basic 6.0, the **NegotiateMenus** property of a form determined whether a menu from an object on the form would be merged with the form's menu. This property could only be set at design time.

In Visual Basic .NET, menus can be merged using the **MergeMenu** method of a **MainMenu** or **ContextMenu** control. For more information, see Merging Menu Items Programmatically.

See Also

Menu Object Changes in Visual Basic .NET | Menus and Context Menus | Forms Changes in Visual Basic .NET

MDI Changes in Visual Basic .NET

In Visual Basic 6.0, multiple-document interface (MDI) applications were created by adding an MDI form to a project and setting the **MDIChild** property of any child forms. In Visual Basic .NET, any form can be made an MDI parent by setting the **IsMdiContainer** property to **true**.

Properties and methods that applied to Visual Basic 6.0 MDI forms have changed in Visual Basic .NET. For more information, see MDIForm Object Changes in Visual Basic .NET.

The behavior of MDI applications has also changed. In Visual Basic 6.0, an MDI application that also contained a form that was not an MDI child would not end until that form was closed, even if the MDI parent was closed. In Visual Basic .NET, the application will end when the startup form is closed, regardless of any non-MDI forms in the application.

See Also

Multiple-Document Interface (MDI) Applications | Forms Changes in Visual Basic .NET |
Introduction to Visual Basic .NET for Visual Basic Veterans

Dialog Box Changes in Visual Basic .NET

In Visual Basic 6.0, a form could be displayed as a modal dialog box by calling the **Show** method with a parameter of **vbModal**.

In Visual Basic .NET, the **ShowDialog** method is used to display a form modally; the **Show** method is used to display a form non-modally.

In addition, the Visual Basic 6.0 **CommonDialog** control provided several predefined dialog boxes. In Visual Basic .NET, this control is replaced by the **ColorDialog** control, the **FontDialog** control, the **OpenFileDialog** control, the **PageSetupDialog** control, the **PrintDialog** control, the **PrintPreviewDialog** control, and the **SaveFileDialog** control.

See Also

Dialog Boxes in Windows Forms | Dialog-Box Controls and Components (Windows Forms)

Color Changes in Visual Basic .NET

In Visual Basic 6.0, colors were represented by a value of type **Long**; in Visual Basic .NET colors are of type **System.Drawing.Color**. When a Visual Basic 6.0 application is upgraded, colors are converted using the **System.Drawing.ColorTranslator.FromOle** class.

In Visual Basic 6.0, constants were provided for the eight standard colors as well as for system colors that could be used to map a color to the user's system preferences. In Visual Basic .NET, the system colors are of type **System.Drawing.SystemColors**.

Color Constant Equivalents

System Color Constant Equivalents

Translating Colors

Color Constant Equivalents

Visual Basic 6.0	Visual Basic .NET Equivalent
vbBlack	System.Drawing.Color.Black
vbRed	System.Drawing.Color.Red
vbGreen	System.Drawing.Color.Lime
vbYellow	System.Drawing.Color.Yellow
vbBlue	System.Drawing.Color.Blue
vbMagenta	System.Drawing.Color.Magenta
vbCyan	System.Drawing.Color.Cyan
vbWhite	System.Drawing.Color.White

System Color Constant Equivalents

Visual Basic 6.0	Visual Basic .NET Equivalent
vb3DDKShadow	System.Drawing.SystemColors.ControlDarkDark
vb3DFace	System.Drawing.SystemColors.Control
vb3DHighlight	System.Drawing.SystemColors.ControlLightLight
vb3DLight	System.Drawing.SystemColors.ControlLight
vb3DShadow	System.Drawing.SystemColors.ControlDark
vbActiveBorder	System.Drawing.SystemColors.ActiveBorder
vbActiveTitleBar	System.Drawing.SystemColors.ActiveCaption
vbActiveTitleBarText	System.Drawing.SystemColors.ActiveCaptionText
vbApplicationWorkspace	System.Drawing.SystemColors.AppWorkspace
vbButtonFace	System.Drawing.SystemColors.Control
vbButtonShadow	System.Drawing.SystemColors.ControlDark
vbButtonText	System.Drawing.SystemColors.ControlText
vbDesktop	System.Drawing.SystemColors.Desktop
vbGrayText	System.Drawing.SystemColors.GrayText
vbHighlight	System.Drawing.SystemColors.Highlight
vbHighlightText	System.Drawing.SystemColors.HighlightText
vbInactiveBorder	System.Drawing.SystemColors.InactiveBorder
vbInactiveCaptionText	System.Drawing.SystemColors.InactiveCaptionText
vbInactiveTitleBar	System.Drawing.SystemColors.InactiveCaption
vbInactiveTitleBarText	System.Drawing.SystemColors.InactiveCaptionText
vbInfoBackground	System.Drawing.SystemColors.Info
vbInfoText	System.Drawing.SystemColors.InfoText
vbMenuBar	System.Drawing.SystemColors.Menu
vbMenuText	System.Drawing.SystemColors.MenuText
vbScrollBars	System.Drawing.SystemColors.Scrollbar
vbTitleBarText	System.Drawing.SystemColors.ActiveCaptionText
vbWindowBackground	System.Drawing.SystemColors.Window
vbWindowFrame	System.Drawing.SystemColors.WindowFrame

vbWindowText	**System.Drawing.SystemColors.WindowText**

Translating Colors

To find an equivalent value for a Visual Basic 6.0 color that is not covered by the color constants, you can use the **System.Drawing.ColorTranslator** class.

The following example sets the **BackColor** property of a **TextBox** control to match the lightest green (&H00C0FFC0) from the Visual Basic 6.0 color picker:

```
' Visual Basic .NET
textBox1.BackColor = System.Drawing.ColorTranslator.FromOle(&H00C0FFC0)
```

See Also

Color Behavior Changes in Visual Basic .NET | Palette Changes in Visual Basic .NET | MaskColor Changes in Visual Basic .NET | Control Changes in Visual Basic .NET | UserControl Changes in Visual Basic .NET

Color Behavior Changes in Visual Basic .NET

In Visual Basic 6.0, the **BackColor** and **ForeColor** properties of a control had to be explicitly set at design time or at run time; colors could not be inherited. In Visual Basic .NET, unless a color is explicitly set at design time or at run time, it will inherit the color setting of its parent.

The following example of a form with two **CommandButton** controls, Command1 and Command2, demonstrates the differences in behavior:

```
' Visual Basic 6.0
' Command1's BackColor is left at its default (gray).
' Command2's BackColor is explicitly set.
Command2.BackColor = vbBlack
' Explicitly set the BackColor of the form.
Form1.BackColor = vbRed
```

After running the above code, the Command1's BackColor is still the default (gray), and Command2's BackColor is black.

The following example shows the behavior in Visual Basic .NET. In Visual Basic .NET, **CommandButton** controls are replaced by **Button** controls.

```
' Visual Basic .NET
' Command1's BackColor is left at its default (gray).
' Command2's BackColor is explicitly set.
Command2.BackColor = System.Drawing.Color.Black
' Explicitly set the BackColor of the form.
Me.BackColor = System.Drawing.Color.Red
```

After running the above code, the Command1's **BackColor** is red, and Command2's **BackColor** is black. Because the **BackColor** was not explicitly set for Command1, the color is inherited from the form.

Note that this will apply for any parent, not just forms. If the **Button** was contained in a **Panel** control within the form, changing the color of either the panel or the form would change the color of the **Button**.

What to do next

- Search for any code that explicitly sets the **BackColor** or **ForeColor** property at run time. If it is being set for a parent, explicitly set the color of the child controls at design time.

See Also

Control Changes in Visual Basic .NET | UserControl Changes in Visual Basic .NET

Palette Changes in Visual Basic .NET

In Visual Basic 6.0, the **Palette** and **PaletteMode** properties of a form were used to control the set of colors to be used when displaying images on a 256-color display.

> **Note** These properties had no effect on displays that support more than 256 colors.

Visual Basic .NET does not support the **Palette** or **PaletteMode** properties. Where Visual Basic 6.0 required a single palette per form, Visual Basic .NET supports a separate palette for each object. Palettes in Visual Basic .NET are set using the **Image.Palette** property in the **System.Drawing** namespace.

See Also

Form Object Changes in Visual Basic .NET | Image.Palette Property

MaskColor Changes in Visual Basic .NET

In Visual Basic 6.0, the **MaskColor** property of a **CheckBox**, **CommandButton**, or **OptionButton** control was used to define a color that would become transparent, allowing a background image to show through. In order to use this property, the **Style** property had to be set to **Graphical**, the **UseMaskColor** property had to be set to **true**, and a bitmap had to be assigned to the **Picture** property.

In Visual Basic .NET, there is no direct equivalent for the **MaskColor** property. You can, however, set transparency on a control using code similar to the following:

```
' Assumes a picture has been assigned to the BackgroundImage property.
Dim g As New System.Drawing.Bitmap(Button1.BackGroundImage)
g.MakeTransparent(System.Drawing.Color.White)
Button1.BackgroundImage = g
```

See Also

CheckBox Control Changes in Visual Basic .NET | CommandButton Control Changes in Visual Basic .NET | OptionButton Control Changes in Visual Basic .NET

Help Support Changes in Visual Basic .NET

In Visual Basic 6.0, Help support was implemented on a per-project basis by specifying a Help file name in the Project Properties dialog box. Each form and control had a **HelpContextID** property that was used to link to a specific topic in the Help file.

In Visual Basic .NET, Help support is implemented on a per-form basis by adding one or more **HelpProvider** components to a form. Each form and control has **HelpKeyword** and **HelpNavigator** properties that are used to link to a specific topic. For more information, see Providing Help in a Windows Application.

> **Note** In Visual Basic 6.0, Help could be provided using either HTML Help or the older Windows Help format. In Visual Basic .NET, only HTML Help is supported.

In Visual Basic 6.0, pop-up Help was implemented using the **WhatsThisButton** and **WhatsThisHelp** properties of a form. The What's This button appeared if **WhatsThisButton** was **true** and either the **MaxButton** or **MinButton** property was **false**. Setting the **WhatsThisMode** property in code enabled the button.

In Visual Basic .NET, pop-up Help is implemented using the **HelpButton** property of a form. The Help button only appears if the **HelpButton** property is set to **true** and both the **MaximizeBox** and **MinimizeBox** properties are set to **false**; the button is automatically enabled. For more information, see Displaying Pop-up Help.

In Visual Basic 6.0, ToolTips were implemented using the **ToolTipText** property of a control. In Visual Basic .NET, ToolTips are implemented by adding a **ToolTip** component to a form. For more information, see ToolTip Changes in Visual Basic .NET.

See Also

Application Assistance

Printing Changes in Visual Basic .NET

Visual Basic .NET provides greatly improved support for printing through classes that control the printing of documents, that allow users to modify print settings and choose printers, and that allow print preview.

Printer Object Changes

In Visual Basic 6.0, the **Printer** object was used for printing; it supported various graphics methods such as **Print**, **Line**, and **PaintPicture** to control what was printed.

In Visual Basic .NET, the **PrintDocument** component replaces the **Printer** object. Equivalent graphics methods include the **DrawString**, **DrawLine**, and **DrawImage** methods of the **Graphics** class.

For details, see Printer Object Changes in Visual Basic .NET.

Printers Collection Changes

In Visual Basic 6.0, the **Printers** collection was used to direct the output of a **Printer** object to a different printer. The **Printers** collection contained a list of available printers that matched the list displayed in the Windows Print dialog box.

In Visual Basic .NET, the **Printers** collection is replaced by the **PrintDialog** control, which provides a standard Windows Print dialog box.

For details, see Printers Collection Changes in Visual Basic .NET.

Other Printing Changes

In Visual Basic 6.0, the **PrintForm** method of a form could be used to send an image of the form to a printer. The **PrintForm** method is no longer supported in Visual Basic .NET. The results of the **PrintForm** method varied greatly depending on the screen resolution and the resolution of the printer; it was not the recommended method of printing. If you need to duplicate the functionality of the **PrintForm** method, you can automate the screen capture capabilities of a third-party graphics tool to capture and print an image of a form.

In Visual Basic 6.0, implementing print preview was only possible through third-party controls. In Visual Basic .NET, the **PrintPreviewDialog** control can be used to provide a standard Windows Print Preview dialog box. For more information, see Introduction to the Windows Forms PrintPreviewDialog Control.

In Visual Basic 6.0, no user interface for page setup was provided. In Visual Basic .NET, the **PageSetupDialog** control can be used to provide a standard Windows Page Setup dialog box. For more information, see Introduction to the Windows Forms PageSetupDialog Component.

In Visual Basic 6.0, printing text from a file required a considerable amount of code. In Visual Basic .NET, the **StreamReader** class can be used to pass the contents of a text file directly to a **PrintDocument** component. For details, see PrintDocument.Print Method.

See Also

Windows Forms Print Support | Printer Object Changes in Visual Basic .NET | Printers Collection Changes in Visual Basic .NET

Debugging Changes in Visual Basic .NET

Although the process of debugging an application in Visual Basic .NET is essentially the same as it was in Visual Basic 6.0, there are a few differences you will want to be aware of. In addition, there are several new features related to debugging in Visual Basic .NET. For more information, see What's New in the Visual Studio Debugger.

In Visual Basic 6.0, in most cases you could make changes to your code while in break mode and continue debugging without stopping and restarting. In Visual Basic .NET, any code changes in break mode require that the project be rebuilt before the code changes can take effect; edit and continue is no longer supported. For more information, see Limitations of Edit and Continue.

In Visual Basic 6.0, syntax errors in code would cause an error message dialog box to be displayed during debugging. In Visual Basic .NET, most syntax errors appear in the Task List window at design-time; run-time exceptions appear in the Output window. For more information, see Debugging Basics.

Some of the keyboard shortcuts for debugging are different in Visual Basic .NET. For example, in Visual Basic 6.0 the F8 command was used to step into code; in Visual Studio .NET the F11 command performs the same function. For more information, see Integrated Development Environment Changes in Visual Basic .NET.

In Visual Basic 6.0, expressions could be evaluated and values could be set in the **Immediate** window. In Visual Basic .NET, the **Command** window replaces the **Immediate** window; when in **Immediate** mode, the functionality is the same. Visual Basic .NET also supports several new options for expressions. For more information, see Expressions in Visual Basic.

See Also

Debugging | Debugging Managed Code | Introduction to Visual Basic .NET for Visual Basic Veterans

Setup and Deployment Changes in Visual Basic .NET

In Visual Basic 6.0, the Package and Deployment Wizard was used to create Setup programs to distribute your application. You could also package applications or components in CAB files for Internet download.

In Visual Basic .NET, Deployment projects are used to create Windows Installers for managed installation. Components can be packaged in redistributable merge modules, and the creation of CAB files is also supported.

> **Note** Later releases of Visual Studio 6.0 included the Visual Studio Installer add-in, which could be used to create Windows Installers. Installer projects created with Visual Studio Installer are not compatible with Visual Basic .NET.

See Also

Deploying Applications and Components

International Changes in Visual Basic .NET

In Visual Basic 6.0, international versions of an application were created by putting all localizable information such as strings into a separate resource file (.res) for each language. At run time, locale-specific resources were loaded from the resource file by calling the **LoadResString**, **LoadResPicture**, and **LoadResData** functions.

In Visual Basic .NET, international versions of an application are created by changing the **Language** property of a form at design time. A separate .NET resource file (.resx) is automatically created for each locale selected. You no longer need to explicitly load resources from code; resources are automatically loaded based on the user's locale. For more information, see Globalizing Windows Forms.

> **Note** Visual Basic 6.0 resource files could be edited using the Resource Editor add-in or the Visual C++ resource editors. Visual Basic .NET resource files cannot be edited directly; instead, you must edit the properties in the IDE.

In Visual Basic 6.0, strings were represented internally as Unicode characters but were displayed using Windows code pages. The **StrConv** function along with Binary and Unicode versions of string manipulation functions (for example, **ChrB** and **ChrW**) were necessary to convert between ANSI and DBCS code pages.

In Visual Basic .NET, forms are entirely Unicode-enabled; conversion between code pages is no longer required. For more information, see Encoding and Windows Forms Globalization.

In Visual Basic 6.0, the formatting of dates and currency in code required special consideration; values entered as text could be incorrectly interpreted when converted to dates or currency in localized applications.

In Visual Basic .NET, dates and currency are automatically formatted according to the user's culture. You can override the settings if necessary through functions in the **System.Globalization** namespace. For more information, see Culture-Specific Classes for Global Windows Forms and Web Forms.

See Also

Globalizing and Localizing Applications | Planning World-Ready Applications | Deployment and Localization | Globalization and Localization Namespaces in Visual Studio

Windows API Changes in Visual Basic .NET

In Visual Basic 6.0, the **Declare** statement was used to call the Windows API in order to access functionality not provided by Visual Basic. In Visual Basic .NET, the need to access the Windows API is greatly diminished because much of the functionality of the Windows API is wrapped up in the .NET Framework.

The **Declare** statement can still be used to call an API, but there are some differences that you need to be aware of. For example, some of the data types in Visual Basic .NET have no direct equivalent; they must be marshaled in order to be passed to an API call. For more information, see Walkthrough: Calling Windows APIs.

See Also

Data Type Changes in Visual Basic | Introduction to Visual Basic .NET for Visual Basic Veterans | Declare Statement

Registry Access Changes in Visual Basic .NET

In Visual Basic 6.0, you could access the registry using the **GetSetting** or **SaveSetting** functions, which provided access to a limited set of registry keys. To access other registry keys, calls to the Windows API were required.

In Visual Basic .NET, the **GetSetting** and **SaveSetting** functions are still supported for limited registry access. For more information, see Registry Summary.

In addition, complete access to the registry is provided via the **Registry** class in the .NET Framework.

In Visual Basic 6.0, files could be marked for registration during installation by manually editing the Setup.lst file. In Visual Basic .NET, the deployment tools provide full support for working with the registry during installation. For more information, see Deployment and the Registry.

See Also

Registry Class | Introduction to Visual Basic .NET for Visual Basic Veterans

Constant Changes in Visual Basic .NET

In Visual Basic 6.0, numerous constants were defined in the Visual Basic language. In Visual Basic .NET, most of the constants are replaced by enumerations in the .NET Framework.

The following links will take you to topics that show the mappings between Visual Basic 6.0 constants and their Visual Basic .NET equivalents.

> **Note** If a constant is not listed, there is no equivalent. For example, DDE constants are not listed because DDE is not supported in Visual Basic .NET.

In This Section

Align Constant Changes in Visual Basic .NET

Alignment Constant Changes in Visual Basic .NET

Arrange Constant Changes in Visual Basic .NET

BorderStyle Constant Changes in Visual Basic .NET

CallType Constant Changes in Visual Basic .NET

CheckBox Constant Changes in Visual Basic .NET

Color Constant Changes in Visual Basic .NET

Compare Method Constant Changes in Visual Basic .NET

Date Format Constant Changes in Visual Basic .NET

DayOfWeek Constant Changes in Visual Basic .NET

File Attribute Constant Changes in Visual Basic .NET

IMEMode Constant Changes in Visual Basic .NET

Key Code Constant Changes in Visual Basic .NET

Mouse Pointer Constant Changes in Visual Basic .NET

MsgBox Result Constant Changes in Visual Basic .NET

MsgBox Style Constant Changes in Visual Basic .NET

MultiSelect Constant Changes in Visual Basic .NET

OLEDropEffect Constant Changes in Visual Basic .NET

ScrollBar Constant Changes in Visual Basic .NET

StartUpPosition Constant Changes in Visual Basic .NET

StrConv Constant Changes in Visual Basic .NET

BorderStyle Constant Changes in Visual Basic .NET

System Color Constant Changes in Visual Basic .NET

TriState Constant Changes in Visual Basic .NET

Variant Type Constant Changes in Visual Basic .NET

VarType Constant Changes in Visual Basic .NET

VBA Constant Changes in Visual Basic .NET

WeekOfYear Constant Changes in Visual Basic .NET

WindowState Constant Changes in Visual Basic .NET

WindowStyle Constant Changes in Visual Basic .NET

Related Sections

Constants and Enumerations

Introduces Visual Basic .NET enumerations.

Control Changes in Visual Basic .NET

Provides links to topics describing the differences between Visual Basic 6.0 and Windows Forms controls and objects.

Language Changes in Visual Basic

Describes changes to the Visual Basic language.

Align Constant Changes in Visual Basic .NET

The following table shows the Visual Basic 6.0 **Align** constants and values and their Visual Basic .NET equivalents.

Visual Basic 6.0	Visual Basic .NET Equivalent
VbAlignNone (0)	System.Windows.Forms.DockStyle.None
VbAlignTop (1)	System.Windows.Forms.DockStyle.Top
VbAlignBottom (2)	System.Windows.Forms.DockStyle.Bottom
VbAlignLeft (3)	System.Windows.Forms.DockStyle.Left
VbAlignRight (4)	System.Windows.Forms.DockStyle.Right

See Also

Constant Changes in Visual Basic .NET | PictureBox Control Changes in Visual Basic .NET | Constants and Enumerations

Alignment Constant Changes in Visual Basic .NET

The following tables show the Visual Basic 6.0 **Alignment** constants and values and their Visual Basic .NET equivalents.

CheckBox and OptionButton Alignment Constants

Visual Basic 6.0	Visual Basic .NET Equivalent
VbLeftJustify (0)	System.Drawing.ContentAlignment.MiddleLeft
VbRightJustify (1)	System.Drawing.ContentAlignment.MiddleRight

TextBox Alignment Constants

Visual Basic 6.0	Visual Basic .NET Equivalent
VbLeftJustify (0)	System.Windows.Forms.HorizontalAlignment.Left
VbRightJustify (1)	System.Windows.Forms.HorizontalAlignment.Right
VbCenter (2)	System.Windows.Forms.HorizontalAlignment.Center

Label Alignment Constants

Visual Basic 6.0	Visual Basic .NET Equivalent
VbLeftJustify (0)	System.Drawing.ContentAlignment.TopLeft
VbRightJustify (1)	System.Drawing.ContentAlignment.TopRight
VbCenter (2)	System.Drawing.ContentAlignment.TopCenter

See Also

Constant Changes in Visual Basic .NET | CheckBox Control Changes in Visual Basic .NET | OptionButton Control Changes in Visual Basic .NET | TextBox Control Changes in Visual Basic .NET | Label Control Changes in Visual Basic .NET | Constants and Enumerations

Arrange Constant Changes in Visual Basic .NET

The following table shows the Visual Basic 6.0 **Arrange** constants and values and their Visual Basic .NET equivalents.

Visual Basic 6.0	Visual Basic .NET Equivalent
VbCascade (0)	System.Windows.Forms.MDILayout.Cascade
VbTileHorizontal (1)	System.Windows.Forms.MDILayout.TileHorizontal
VbTileVertical (2)	System.Windows.Forms.MDILayout.TileVertical
VbArrangeIcons (3)	System.Windows.Forms.MDILayout.ArrangeIcons

See Also

Constant Changes in Visual Basic .NET | MDIForm Object Changes in Visual Basic .NET | Constants and Enumerations

BorderStyle Constant Changes in Visual Basic .NET

The following table shows the Visual Basic 6.0 **BorderStyle** constants and values and their Visual Basic .NET equivalents.

Visual Basic 6.0	Visual Basic .NET Equivalent
VbBSNone (0)	**System.Windows.Forms.FormBorderStyle.None**
VbFixedSingle (1)	**System.Windows.Forms.FormBorderStyle.FixedSingle**
VbSizable (2)	**System.Windows.Forms.FormBorderStyle.Sizable**
VbFixedDialog (3)	**System.Windows.Forms.FormBorderStyle.FixedDialog**
VbFixedToolWindow (4)	**System.Windows.Forms.FormBorderStyle.FixedToolWindow**
VbSizableToolWindow (5)	**System.Windows.Forms.FormBorderStyle.SizableToolWindow**

See Also

Constant Changes in Visual Basic .NET | Form Object Changes in Visual Basic .NET | Constants and Enumerations

CallType Constant Changes in Visual Basic .NET

The following table shows the Visual Basic 6.0 **CallType** constants and their Visual Basic .NET equivalents.

Note For backward compatibility, the Visual Basic 6.0 **CallType** constants may also be used in Visual Basic .NET.

Visual Basic 6.0	Visual Basic .NET Equivalent
vbMethod	**CallType.Method**
vbGet	**CallType.Get**
vbSet	**CallType.Set**
vbLet	**CallType.Set**

See Also

Constant Changes in Visual Basic .NET | CallByName Function | Constants and Enumerations

CheckBox Constant Changes in Visual Basic .NET

The following table shows the Visual Basic 6.0 **CheckBox** constants and values and their Visual Basic .NET equivalents.

Visual Basic 6.0	Visual Basic .NET Equivalent
vbUnchecked (0)	System.Windows.Forms.CheckState.Unchecked
vbChecked (1)	System.Windows.Forms.CheckState.Checked
vbGrayed (2)	System.Windows.Forms.CheckState.Indeterminate

See Also

Constant Changes in Visual Basic .NET | CheckBox Control Changes in Visual Basic .NET | Constants and Enumerations

Color Constant Changes in Visual Basic .NET

The following table shows the Visual Basic 6.0 **Color** constants and their Visual Basic .NET equivalents.

Visual Basic 6.0	Visual Basic .NET Equivalent
vbBlack	**System.Drawing.Color.Black**
vbRed	**System.Drawing.Color.Red**
vbGreen	**System.Drawing.Color.Lime**
vbYellow	**System.Drawing.Color.Yellow**
vbBlue	**System.Drawing.Color.Blue**
vbMagenta	**System.Drawing.Color.Magenta**
vbCyan	**System.Drawing.Color.Cyan**
vbWhite	**System.Drawing.Color.White**

See Also

Constant Changes in Visual Basic .NET | Color Changes in Visual Basic .NET | Color Behavior Changes in Visual Basic .NET | System Color Constant Changes in Visual Basic .NET | Constants and Enumerations

Compare Method Constant Changes in Visual Basic .NET

The following table shows the Visual Basic 6.0 **Compare** constants and their Visual Basic .NET equivalents.

Note For backward compatibility, the Visual Basic 6.0 **Compare** constants may also be used in Visual Basic .NET.

Visual Basic 6.0	Visual Basic .NET Equivalent
vbBinaryCompare	**CompareMethod.Binary**
vbTextCompare	**CompareMethod.Text**
vbDatabaseCompare	No equivalent

See Also

Constant Changes in Visual Basic .NET | CompareMethod Enumeration | Constants and Enumerations

Date Format Constant Changes in Visual Basic .NET

The following table shows the Visual Basic 6.0 **Date Format** constants and their Visual Basic .NET equivalents.

Note For backward compatibility, the Visual Basic 6.0 **Date Format** constants may also be used in Visual Basic .NET.

Visual Basic 6.0	Visual Basic .NET Equivalent
vbGeneralDate	**DateFormat.GeneralDate**
vbLongDate	**DateFormat.LongDate**
vbShortDate	**DateFormat.ShortDate**
vbLongTime	**DateFormat.LongTime**
vbShortTime	**DateFormat.ShortTime**

See Also

Constant Changes in Visual Basic .NET | DateFormat Enumeration | Constants and Enumerations

DayOfWeek Constant Changes in Visual Basic .NET

The following table shows the Visual Basic 6.0 **DayOfWeek** constants and their Visual Basic .NET equivalents.

Note For backward compatibility, the Visual Basic 6.0 **DayOfWeek** constants may also be used in Visual Basic .NET.

Visual Basic 6.0	Visual Basic .NET Equivalent
VbUseSystemDayOfWeek	**FirstDayOfWeek.System**
vbSunday	**FirstDayOfWeek.Sunday**
vbMonday	**FirstDayOfWeek.Monday**
vbTuesday	**FirstDayOfWeek.Tuesday**
vbWednesday	**FirstDayOfWeek.Wednesday**
vbThursday	**FirstDayOfWeek.Thursday**
vbFriday	**FirstDayOfWeek.Friday**
vbSaturday	**FirstDayOfWeek.Saturday**

See Also

Constant Changes in Visual Basic .NET | FirstDayOfWeek Enumeration | Constants and Enumerations

File Attribute Constant Changes in Visual Basic .NET

The following table shows the Visual Basic 6.0 **File Attribute** constants and their Visual Basic .NET equivalents.

Note For backward compatibility, the Visual Basic 6.0 **File Attribute** constants may also be used in Visual Basic .NET.

Visual Basic 6.0	Visual Basic .NET Equivalent
vbNormal	**FileAttribute.Normal**
vbReadOnly	**FileAttribute.ReadOnly**
vbHidden	**FileAttribute.Hidden**
vbSystem	**FileAttribute.System**
vbVolume	**FileAttribute.Volume**
vbDirectory	**FileAttribute.Directory**
vbArchive	**FileAttribute.Archive**
vbAlias	No equivalent

See Also

Constant Changes in Visual Basic .NET | FileAttribute Enumeration | Constants and Enumerations

IMEMode Constant Changes in Visual Basic .NET

The following tables show the Visual Basic 6.0 **IMEMode** constants and values and their Visual Basic .NET equivalents.

Visual Basic 6.0	Visual Basic .NET Equivalent
vbIMEAlphaDbl (7)	System.Windows.Forms.ImeMode.AlphaFull
vbIMEAlphaSng (8)	System.Windows.Forms.ImeMode.Alpha
vbIMEDisable (3)	System.Windows.Forms.ImeMode.Disable
vbIMEHiragana (4)	System.Windows.Forms.ImeMode.Hiragana
vbIMEKatakanaDbl (5)	System.Windows.Forms.ImeMode.Katakana
vbIMEKatakanaSng (6)	System.Windows.Forms.ImeMode.KatakanaHalf
vbIMEModeAlpha (8)	System.Windows.Forms.ImeMode.Alpha
vbIMEModeAlphaFull (7)	System.Windows.Forms.ImeMode.AlphaFull
vbIMEModeDisable (3)	System.Windows.Forms.ImeMode.Disable
vbIMEModeHangul (10)	System.Windows.Forms.ImeMode.Hangul
vbIMEModeHangulFull (9)	System.Windows.Forms.ImeMode.HangulFull
vbIMEModeHiragana (4)	System.Windows.Forms.ImeMode.Hiragana
vbIMEModeKatakana (5)	System.Windows.Forms.ImeMode.Katakana
vbIMEModeKatakanaHalf (6)	System.Windows.Forms.ImeMode.KatakanaHalf
vbIMEModeNoControl (0)	System.Windows.Forms.ImeMode.NoControl
vbIMEModeOff (2)	System.Windows.Forms.ImeMode.Off
vbIMEModeOn (1)	System.Windows.Forms.ImeMode.On
vbIMENoOp (0)	System.Windows.Forms.ImeMode.NoControl
vbIMEOff (2)	System.Windows.Forms.ImeMode.Off
vbIMEOn (1)	System.Windows.Forms.ImeMode.On

See Also

Constant Changes in Visual Basic .NET | Form Object Changes in Visual Basic .NET | Constants and Enumerations

Key Code Constant Changes in Visual Basic .NET

The following tables show the Visual Basic 6.0 **Key Code** constants and values and their Visual Basic .NET equivalents.

Visual Basic 6.0	Visual Basic .NET Equivalent
vbKeyLButton (1)	System.Windows.Forms.Keys.LButton
vbKeyRButton (2)	System.Windows.Forms.Keys.RButton
vbKeyCancel (3)	System.Windows.Forms.Keys.Cancel
vbKeyMButton (4)	System.Windows.Forms.Keys.MButton
vbKeyBack (8)	System.Windows.Forms.Keys.Back
vbKeyTab (9)	System.Windows.Forms.Keys.Tab
vbKeyClear (12)	System.Windows.Forms.Keys.Clear
vbKeyReturn (13)	System.Windows.Forms.Keys.Return
vbKeyShift (16)	System.Windows.Forms.Keys.ShiftKey
vbKeyControl (17)	System.Windows.Forms.Keys.ControlKey
vbKeyMenu (18)	System.Windows.Forms.Keys.Menu
vbKeyPause (19)	System.Windows.Forms.Keys.Pause
vbKeyCapital (20)	System.Windows.Forms.Keys.Capital
vbKeyEscape (27)	System.Windows.Forms.Keys.Escape
vbKeySpace (32)	System.Windows.Forms.Keys.Space
vbKeyPageUp (33)	System.Windows.Forms.Keys.PageUp
vbKeyPageDown (34)	System.Windows.Forms.Keys.PageDown
vbKeyEnd (35)	System.Windows.Forms.Keys.End
vbKeyHome (36)	System.Windows.Forms.Keys.Home
vbKeyLeft (37)	System.Windows.Forms.Keys.Left
vbKeyUp (38)	System.Windows.Forms.Keys.Up
vbKeyRight (39)	System.Windows.Forms.Keys.Right
vbKeyDown (40)	System.Windows.Forms.Keys.Down
vbKeySelect (41)	System.Windows.Forms.Keys.Select
vbKeyPrint (42)	System.Windows.Forms.Keys.Print
vbKeyExecute (43)	System.Windows.Forms.Keys.Execute
vbKeySnapshot (44)	System.Windows.Forms.Keys.Snapshot
vbKeyInsert (45)	System.Windows.Forms.Keys.Insert
vbKeyDelete (46)	System.Windows.Forms.Keys.Delete
vbKeyHelp (47)	System.Windows.Forms.Keys.Help
vbKeyNumlock (144)	System.Windows.Forms.Keys.Numlock
vbKeyScrollLock (145)	System.Windows.Forms.Keys.Scroll

Alphabetic Key Code Constants

Visual Basic 6.0	Visual Basic .NET Equivalent
VbKeyA (65)	System.Windows.Forms.Keys.A
VbKeyB (66)	System.Windows.Forms.Keys.B
VbKeyC (67)	System.Windows.Forms.Keys.C
VbKeyD (68)	System.Windows.Forms.Keys.D
VbKeyE (69)	System.Windows.Forms.Keys.E
VbKeyF (70)	System.Windows.Forms.Keys.F
VbKeyG (71)	System.Windows.Forms.Keys.G
VbKeyH (72)	System.Windows.Forms.Keys.H
VbKeyI (73)	System.Windows.Forms.Keys.I
VbKeyJ (74)	System.Windows.Forms.Keys.J
VbKeyK (75)	System.Windows.Forms.Keys.K
VbKeyL(76)	System.Windows.Forms.Keys.L
vbKeyM (77)	System.Windows.Forms.Keys.M
VbKeyN (78)	System.Windows.Forms.Keys.N

VbKeyO (79)	System.Windows.Forms.Keys.O
VbKeyP(80)	System.Windows.Forms.Keys.P
VbKeyQ (81)	System.Windows.Forms.Keys.Q
VbKeyR (82)	System.Windows.Forms.Keys.R
VbKeyS (83)	System.Windows.Forms.Keys.S
VbKeyT (84)	System.Windows.Forms.Keys.T
VbKeyU (85)	System.Windows.Forms.Keys.U
VbKeyV (86)	System.Windows.Forms.Keys.V
VbKeyW (87)	System.Windows.Forms.Keys.W
VbKeyX (88)	System.Windows.Forms.Keys.X
VbKeyY (89)	System.Windows.Forms.Keys.Y
VbKeyZ (90)	System.Windows.Forms.Keys.Z

Numeric Key Constants

Visual Basic 6.0	Visual Basic .NET Equivalent
vbKey0 (48)	System.Windows.Forms.Keys.D0
vbKey1 (49)	System.Windows.Forms.Keys.D1
vbKey2 (50)	System.Windows.Forms.Keys.D2
vbKey3 (51)	System.Windows.Forms.Keys.D3
vbKey4 (52)	System.Windows.Forms.Keys.D4
vbKey5 (53)	System.Windows.Forms.Keys.D5
vbKey6 (54)	System.Windows.Forms.Keys.D6
vbKey7 (55)	System.Windows.Forms.Keys.D7
vbKey8 (56)	System.Windows.Forms.Keys.D8
vbKey9 (57)	System.Windows.Forms.Keys.D9

Numeric Keypad Key Constants

Visual Basic 6.0	Visual Basic .NET Equivalent
vbKeyNumpad0 (96)	System.Windows.Forms.Keys.NumPad0
vbKeyNumpad1 (97)	System.Windows.Forms.Keys.NumPad1
vbKeyNumpad2 (98)	System.Windows.Forms.Keys.NumPad2
vbKeyNumpad3 (99)	System.Windows.Forms.Keys.NumPad3
vbKeyNumpad4 (100)	System.Windows.Forms.Keys.NumPad4
vbKeyNumpad5 (101)	System.Windows.Forms.Keys.NumPad5
vbKeyNumpad6 (102)	System.Windows.Forms.Keys.NumPad6
vbKeyNumpad7 (103)	System.Windows.Forms.Keys.NumPad7
vbKeyNumpad8 (104)	System.Windows.Forms.Keys.NumPad8
vbKeyNumpad9 (105)	System.Windows.Forms.Keys.NumPad9
VbKeyMultiply (106)	System.Windows.Forms.Keys.Multiply
VbKeyAdd (107)	System.Windows.Forms.Keys.Add
VbKeySeparator (108)	System.Windows.Forms.Keys.Separator
VbKeySubtract (109)	System.Windows.Forms.Keys.Subtract
VbKeyDecimal (110)	System.Windows.Forms.Keys.Decimal
VbKeyDivide (111)	System.Windows.Forms.Keys.Divide

Function Key Constants

Visual Basic 6.0	Visual Basic .NET Equivalent
vbKeyF1 (112)	System.Windows.Forms.Keys.F1
vbKeyF2 (113)	System.Windows.Forms.Keys.F2
vbKeyF3 (114)	System.Windows.Forms.Keys.F3
vbKeyF4 (115)	System.Windows.Forms.Keys.F4
vbKeyF5 (116)	System.Windows.Forms.Keys.F5
vbKeyF6 (117)	System.Windows.Forms.Keys.F6
vbKeyF7 (118)	System.Windows.Forms.Keys.F7

vbKeyF8 (119)	**System.Windows.Forms.Keys.F8**
vbKeyF9 (120)	**System.Windows.Forms.Keys.F9**
vbKeyF10 (121)	**System.Windows.Forms.Keys.F10**
vbKeyF11 (122)	**System.Windows.Forms.Keys.F11**
vbKeyF12 (123)	**System.Windows.Forms.Keys.F12**
vbKeyF13 (124)	**System.Windows.Forms.Keys.F13**
vbKeyF14 (125)	**System.Windows.Forms.Keys.F14**
vbKeyF15 (126)	**System.Windows.Forms.Keys.F15**
vbKeyF16 (127)	**System.Windows.Forms.Keys.F16**

See Also

Constant Changes in Visual Basic .NET | Keys Enumeration | Constants and Enumerations

Mouse Pointer Constant Changes in Visual Basic .NET

The following table shows the Visual Basic 6.0 **Mouse Pointer** constants and values and their Visual Basic .NET equivalents.

Visual Basic 6.0	Visual Basic .NET Equivalent
VbDefault (0)	**Windows.Forms.Cursors.Default**
VbArrow (1)	**Windows.Forms.Cursors.Arrow**
VbCrosshair (2)	**Windows.Forms.Cursors.Cross**
VbIbeam (3)	**Windows.Forms.Cursors.IBeam**
VbIconPointer (4)	Obsolete — replaced by **System.Windows.Forms.Cursors.Default**
VbSizePointer (5)	**Windows.Forms.Cursors.SizeAll**
VbSizeNESW (6)	**Windows.Forms.Cursors.SizeNESW**
VbSizeNS (7)	**Windows.Forms.Cursors.SizeNS**
VbSizeNWSE (8)	**Windows.Forms.Cursors.SizeNWSE**
VbSizeWE (9)	**Windows.Forms.Cursors.SizeWE**
VbUpArrow (10)	**Windows.Forms.Cursors.UpArrow**
VbHourglass (11)	**Windows.Forms.Cursors.WaitCursor**
VbNoDrop (12)	**System.Drawing.Cursurs.NoDrop**
VbArrowHourglass (13)	**Windows.Forms.Cursors.AppStarting**
VbArrowQuestion (14)	**Windows.Forms.Cursors.Help**
VbSizeAll (15)	**Windows.Forms.Cursors.SizeAll**
VbCustom (99)	No equivalent — for details, see Cannot set a custom MousePointer

See Also

Constant Changes in Visual Basic .NET | MousePointer Changes in Visual Basic .NET | Constants and Enumerations

MsgBox Result Constant Changes in Visual Basic .NET

The following table shows the Visual Basic 6.0 **MsgBox Result** constants and their Visual Basic .NET equivalents.

> **Note** For backward compatibility, the Visual Basic 6.0 **MsgBox Result** constants may also be used in Visual Basic .NET.

Visual Basic 6.0	Visual Basic .NET Equivalent
vbOK	**MsgBoxResult.OK**
vbCancel	**MsgBoxResult.Cancel**
vbAbort	**MsgBoxResult.Abort**
vbRetry	**MsgBoxResult.Retry**
vbIgnore	**MsgBoxResult.Ignore**
vbYes	**MsgBoxResult.Yes**
vbNo	**MsgBoxResult.No**

See Also

Constant Changes in Visual Basic .NET | MsgBoxResult Enumeration | Constants and Enumerations

MsgBox Style Constant Changes in Visual Basic .NET

The following table shows the Visual Basic 6.0 **MsgBox Style** constants and their Visual Basic .NET equivalents.

> **Note** For backward compatibility, the Visual Basic 6.0 **MsgBox Style** constants may also be used in Visual Basic .NET.

Visual Basic 6.0	Visual Basic .NET Equivalent
vbOKOnly	**MsgBoxStyle.OKOnly**
vbOKCancel	**MsgBoxStyle.OKCancel**
vbAbortRetryIgnore	**MsgBoxStyle.AbortRetryIgnore**
vbYesNoCancel	**MsgBoxStyle.YesNoCancel**
vbYesNo	**MsgBoxStyle.YesNo**
vbRetryCancel	**MsgBoxStyle.RetryCancel**
vbCritical	**MsgBoxStyle.Critical**
vbQuestion	**MsgBoxStyle.Question**
vbExclamation	**MsgBoxStyle.Exclamation**
vbInformation	**MsgBoxStyle.Information**
vbDefaultButton1	**MsgBoxStyle.DefaultButton1**
vbDefaultButton2	**MsgBoxStyle.DefaultButton2**
vbDefaultButton3	**MsgBoxStyle.DefaultButton3**
vbDefaultButton4	No equivalent
vbApplicationModal	**MsgBoxStyle.ApplicationModal**
vbSystemModal	**MsgBoxStyle.SystemModal**
vbMsgBoxHelpButton	**MsgBoxStyle.MsgBoxHelp**
vbMsgBoxRight	**MsgBoxStyle.MsgBoxRight**
vbMsgBoxRtlReading	**MsgBoxStyle.MsgBoxRtlReading**
vbMsgBoxSetForeground	**MsgBoxStyle.MsgBoxSetForeground**

See Also

Constant Changes in Visual Basic .NET | MsgBoxStyle Enumeration | Constants and Enumerations

MultiSelect Constant Changes in Visual Basic .NET

The following table shows the Visual Basic 6.0 **MultiSelect** constants and values and their Visual Basic .NET equivalents.

Visual Basic 6.0	Visual Basic .NET Equivalent
vbMultiSelectNone (0)	System.Windows.Forms.SelectionMode.One
vbMultiSelectSimple (1)	System.Windows.Forms.SelectionMode.MultiSimple
vbvbMultiSelectExtended (2)	System.Windows.Forms.SelectionMode.MultiExtended

See Also

Constant Changes in Visual Basic .NET | ListBox Control Changes in Visual Basic .NET | Constants and Enumerations

OLEDropEffect Constant Changes in Visual Basic .NET

The following table shows the Visual Basic 6.0 **OLEDropEffect** constants and values and their Visual Basic .NET equivalents.

Visual Basic 6.0	Visual Basic .NET Equivalent
VbDropEffectNone (0)	**System.Windows.Forms.DragDropEffects.None**
VbDropEffectCopy (1)	**System.Windows.Forms.DragDropEffects.Copy**
VbDropEffectMove (2)	**System.Windows.Forms.DragDropEffects.Move**
VbDropEffectScroll (&H80000000)	**System.Windows.Forms.DragDropEffects.Scroll**

See Also

Constant Changes in Visual Basic .NET | Drag and Drop Changes in Visual Basic .NET | Constants and Enumerations

ScrollBar Constant Changes in Visual Basic .NET

The following table shows the Visual Basic 6.0 **ScrollBar** constants and values and their Visual Basic .NET equivalents.

Visual Basic 6.0	Visual Basic .NET Equivalent
vbSBNone (0)	System.Windows.Forms.ScrollBars.None
vbHorizontal (1)	System.Windows.Forms.ScrollBars.Horizontal
vbVertical (2)	System.Windows.Forms.ScrollBars.Vertical
vbBoth (3)	System.Windows.Forms.ScrollBars.Both

See Also

Constant Changes in Visual Basic .NET | Form Object Changes in Visual Basic .NET | PictureBox Control Changes in Visual Basic .NET | Constants and Enumerations

StartUpPosition Constant Changes in Visual Basic .NET

The following table shows the Visual Basic 6.0 **StartUpPosition** constants and values and their Visual Basic .NET equivalents.

Visual Basic 6.0	Visual Basic .NET Equivalent
vbStartUpManual (0)	**System.Windows.Forms.FormStartPosition.Manual**
vbStartUpOwner (1)	**System.Windows.Forms.FormStartPosition.CenterParent**
vbStartUpScreen (2)	**System.Windows.Forms.FormStartPosition.CenterScreen**
vbStartUpWindowsDefault (3)	**System.Windows.Forms.FormStartPosition.WindowsDefaultLocation**

See Also

Constant Changes in Visual Basic .NET | Form Object Changes in Visual Basic .NET | Constants and Enumerations

StrConv Constant Changes in Visual Basic .NET

The following table shows the Visual Basic 6.0 **StrConv** constants and their Visual Basic .NET equivalents.

Note For backward compatibility, the Visual Basic 6.0 **StrConv** constants may also be used in Visual Basic .NET.

Visual Basic 6.0	Visual Basic .NET Equivalent
vbUpperCase	**VbStrConv.UpperCase**
vbLowerCase	**VbStrConv.LowerCase**
vbProperCase	**VbStrConv.ProperCase**
vbWide	**VbStrConv.Wide**
vbNarrow	**VbStrConv.Narrow**
vbKatakana	**VbStrConv.Katakana**
vbHiragana	**VbStrConv.Hiragana**
vbUnicode	No equivalent constant. You can achieve the same functionality by using the **System.Text.Encoding.Convert** method. For more information, see Encoding.Convert Method (Encoding, Encoding, Byte[]) method.
vbFromUnicode	No equivalent

See Also

Constant Changes in Visual Basic .NET | VbStrConv Enumeration | Constants and Enumerations

Style Constant Changes in Visual Basic .NET

The following tables show the Visual Basic 6.0 **Style** constants and values and their Visual Basic .NET equivalents.

ComboBox Style Constants

Visual Basic 6.0	Visual Basic .NET Equivalent
vbComboDropdown (0)	System.Windows.Forms.ComboBoxStyle.DropDown
vbComboSimple (1)	System.Windows.Forms.ComboBoxStyle.Simple
vbComboDropdownList (2)	System.Windows.Forms.ComboBoxStyle.DropDownList

CheckBox and OptionButton Style Constants

Visual Basic 6.0	Visual Basic .NET Equivalent
VbButtonStandard (0)	System.Windows.Forms.Appearance.Normal
VbButtonGraphical (1)	System.Windows.Forms.Appearance.Button

> **Note** In Visual Basic 6.0, **Style** constants were also defined for **CommandButton** and **ListBox** controls. There are no equivalents in Visual Basic .NET. For details, see Style Property Changes in Visual Basic .NET or ListBox Control Changes in Visual Basic .NET.| Constants and Enumerations

See Also

Constant Changes in Visual Basic .NET | ComboBox Control Changes in Visual Basic .NET | Constants and Enumerations

System Color Constant Changes in Visual Basic .NET

The following table shows the Visual Basic 6.0 **System Color** constants and their Visual Basic .NET equivalents.

Visual Basic 6.0	Visual Basic .NET Equivalent
vb3DDKShadow	System.Drawing.SystemColors.ControlDarkDark
vb3DFace	System.Drawing.SystemColors.Control
vb3DHighlight	System.Drawing.SystemColors.ControlLightLight
vb3DLight	System.Drawing.SystemColors.ControlLight
vb3DShadow	System.Drawing.SystemColors.ControlDark
vbActiveBorder	System.Drawing.SystemColors.ActiveBorder
vbActiveTitleBar	System.Drawing.SystemColors.ActiveCaption
vbActiveTitleBarText	System.Drawing.SystemColors.ActiveCaptionText
vbApplicationWorkspace	System.Drawing.SystemColors.AppWorkspace
vbButtonFace	System.Drawing.SystemColors.Control
vbButtonShadow	System.Drawing.SystemColors.ControlDark
vbButtonText	System.Drawing.SystemColors.ControlText
vbDesktop	System.Drawing.SystemColors.Desktop
vbGrayText	System.Drawing.SystemColors.GrayText
vbHighlight	System.Drawing.SystemColors.Highlight
vbHighlightText	System.Drawing.SystemColors.HighlightText
vbInactiveBorder	System.Drawing.SystemColors.InactiveBorder
vbInactiveCaptionText	System.Drawing.SystemColors.InactiveCaptionText
vbInactiveTitleBar	System.Drawing.SystemColors.InactiveCaption
vbInactiveTitleBarText	System.Drawing.SystemColors.InactiveCaptionText
vbInfoBackground	System.Drawing.SystemColors.Info
vbInfoText	System.Drawing.SystemColors.InfoText
vbMenuBar	System.Drawing.SystemColors.Menu
vbMenuText	System.Drawing.SystemColors.MenuText
vbScrollBars	System.Drawing.SystemColors.Scrollbar
vbTitleBarText	System.Drawing.SystemColors.ActiveCaptionText
vbWindowBackground	System.Drawing.SystemColors.Window
vbWindowFrame	System.Drawing.SystemColors.WindowFrame
vbWindowText	System.Drawing.SystemColors.WindowText

See Also

Constant Changes in Visual Basic .NET | Color Changes in Visual Basic .NET | Color Behavior Changes in Visual Basic .NET | Color Constant Changes in Visual Basic .NET | Constants and Enumerations

TriState Constant Changes in Visual Basic .NET

The following table shows the Visual Basic 6.0 **TriState** constants and their Visual Basic .NET equivalents.

Note For backward compatibility, the Visual Basic 6.0 **TriState** constants may also be used in Visual Basic .NET.

Visual Basic 6.0	Visual Basic .NET Equivalent
vbTrue	**TriState.True**
vbFalse	**TriState.False**
vbUseDefault	**TriState.UseDefault**

See Also

Constant Changes in Visual Basic .NET | TriState Enumeration | Constants and Enumerations

Variant Type Constant Changes in Visual Basic .NET

The following table shows the Visual Basic 6.0 **VarType** constants and their Visual Basic .NET equivalents.

Visual Basic 6.0	Visual Basic .NET Equivalent
vbVEmpty	**VariantType.Nothing**
vbVNull	**VariantType.Null**
vbVInteger	**VariantType.Short**
vbVLong	**VariantType.Integer**
vbVSingle	**VariantType.Single**
vbVDouble	**VariantType.Double**
vbVCurrency	**VariantType.Decimal**
vbVDate	**VariantType.Date**
vbVString	**VariantType.String**

See Also

Constant Changes in Visual Basic .NET | VariantType (Object Browser) | Constants and Enumerations

VarType Constant Changes in Visual Basic .NET

The following table shows the Visual Basic 6.0 **VarType** constants and their Visual Basic .NET equivalents.

Visual Basic 6.0	Visual Basic .NET Equivalent
vbEmpty	VariantType.Empty
vbNull	VariantType.Null
vbInteger	VariantType.Short
vbLong	VariantType.Integer
vbSingle	VariantType.Single
vbDouble	VariantType.Double
vbCurrency	VariantType.Decimal
vbDate	VariantType.Date
vbString	VariantType.String
vbObject	VariantType.Object
vbError	VariantType.Error
vbBoolean	VariantType.Boolean
vbVariant	VariantType.Variant
vbDataObject	VariantType.DataObject
vbDecimal	VariantType.Decimal
vbByte	VariantType.Byte
vbUserDefinedType	VariantType.UserDefinedType
vbArray	VariantType.Array

See Also

Constant Changes in Visual Basic .NET | VariantType (Object Browser) | Constants and Enumerations

VBA Constant Changes in Visual Basic .NET

The following table shows the Visual Basic 6.0 **VBA** constants and their Visual Basic .NET equivalents.

Note For backward compatibility, the Visual Basic 6.0 **VBA** constants may also be used in Visual Basic .NET.

Visual Basic 6.0	Visual Basic .NET Equivalent
vbBack	**ControlChars.Back**
vbCr	**ControlChars.Cr**
vbCrLf	**ControlChars.CrLf**
vbFormFeed	**ControlChars.FormFeed**
vbLf	**ControlChars.Lf**
vbNewLine	**ControlChars.NewLine**
vbNullChar	**ControlChars.NullChar**
vbNullString	**Nothing** keyword
vbObjectError	**vbObjectError**
vbTab	**ControlChars.Tab**

See Also

Constant Changes in Visual Basic .NET | Constants and Enumerations

WeekOfYear Constant Changes in Visual Basic .NET

The following table shows the Visual Basic 6.0 **WeekOfYear** constants and their Visual Basic .NET equivalents.

Note For backward compatibility, the Visual Basic 6.0 **WeekOfYear** constants may also be used in Visual Basic .NET.

Visual Basic 6.0	Visual Basic .NET Equivalent
vbUseSystem	FirstWeekOfYear.System
VbFirstJan1	FirstWeekOfYear.Jan1
vbFirstFourDays	FirstWeekOfYear.FirstFourDays
vbFirstFullWeek	FirstWeekOfYear.FirstFullWeek

See Also

Constant Changes in Visual Basic .NET | FirstWeekOfYear Enumeration | Constants and Enumerations

WindowState Constant Changes in Visual Basic .NET

The following table shows the Visual Basic 6.0 **WindowState** constants and values and their Visual Basic .NET equivalents.

Visual Basic 6.0	Visual Basic .NET Equivalent
vbNormal (0)	System.Windows.Forms.FormWindowState.Normal
vbMinimized (1)	System.Windows.Forms.FormWindowState.Minimized
vbMaximized (2)	System.Windows.Forms.FormWindowState.Maximized

See Also

Constant Changes in Visual Basic .NET | Form Object Changes in Visual Basic .NET | Constants and Enumerations

WindowStyle Constant Changes in Visual Basic .NET

The following table shows the Visual Basic 6.0 **WindowStyle** constants and their Visual Basic .NET equivalents.

> **Note** For backward compatibility, the Visual Basic 6.0 **WindowStyle** constants may also be used in Visual Basic .NET.

Visual Basic 6.0	Visual Basic .NET Equivalent
vbHide	**AppWinStyle.Hide**
vbNormalFocus	**AppWinStyle.NormalFocus**
vbMinimizedFocus	**AppWinStyle.MinimizedFocus**
vbMaximizedFocus	**AppWinStyle.MaximizedFocus**
vbNormalNoFocus	**AppWinStyle.NormalNoFocus**
vbMinimizedNoFocus	**AppWinStyle.MinimizedNoFocus**

See Also

Constant Changes in Visual Basic .NET | AppWinStyle Enumeration | Constants and Enumerations

Namespace Changes in Visual Basic .NET

Namespaces may at first seem to be a new concept in Visual Basic .NET; conceptually they are similar to object libraries in Visual Basic 6.0.

In Visual Basic 6.0, various libraries contained objects that were used to build an application. For example, the Visual Basic library contained the form object and intrinsic control objects that were the foundation of a Windows application. You could see the objects contained in a given library by viewing it in the Object Browser.

In Visual Basic .NET, objects are contained in assemblies that are part of the .NET Framework Class Library. Each assembly represents a namespace. For example, form and control objects are contained in the **System.Windows.Forms** namespace. As in Visual Basic 6.0, namespaces can be inspected in the Object Browser.

See Also

Introduction to the .NET Framework Class Library in Visual Studio

Run-Time Changes in Visual Basic .NET

For applications created with Visual Basic 6.0, the Visual Basic run-time file (Msvbvm60.dll) had to be installed on a client computer in order for the application to run. This file was often included in Setup packages so that it could be installed along with the application if necessary.

In Visual Basic .NET, applications require the .NET Framework common language runtime, a run-time environment that manages the execution of code on a client computer. The common language runtime files are packaged in a merge module that is automatically included in installers built using Visual Studio deployment projects.

See Also

Deploying Applications and Components | Distributing the Common Language Runtime

Upgrading Applications Created in Previous Versions of Visual Basic

Visual Studio .NET provides the ability to upgrade applications created in Visual Basic 6.0 so that you can continue development, taking advantage of the benefits of the .NET Platform. When you first open a Visual Basic 6.0 project file (.vbp), the Upgrade Wizard will appear. A command-line tool is also provided for upgrading projects outside of the Visual Studio environment.

The upgrading tools modify the code within your project to comply with Visual Basic .NET syntax and replace any forms and controls with Visual Basic .NET equivalents. Due to the differences between Visual Basic 6.0 and Visual Basic. NET, some things in your project may not map directly — in these cases, an upgrade report is provided to guide you through the process of modifying your application.

Some Visual Basic 6.0 applications may not benefit from being upgraded to Visual Basic .NET. In addition, there are some things that you can do to your Visual Basic 6.0 applications that will help to make the upgrade process go smoothly.

The following topics will help you better understand the upgrade process.

In This Section

Why Upgrading is Necessary
Discusses the differences between Visual Basic 6.0 and Visual Basic .NET.
Things to Consider Before Upgrading
Discusses the limitations of and considerations for upgrading.
Preparing a Visual Basic 6.0 Application for Upgrading
Describes how to make sure that your application is ready to be upgraded.
Language Recommendations for Upgrading
Provides recommendations you can follow when preparing a Visual Basic 6.0 application for upgrade, or when developing Visual Basic 6.0 applications that might later be upgraded.
Visual Basic 6.0 Compatibility Library
Describes a mechanism provided for backward compatibility.
Visual Basic Upgrade Wizard
Provides an overview of upgrading a Visual Basic 6.0 project to Visual Basic .NET with the Upgrade Wizard.
Upgrading a Project with the Upgrade Wizard
Describes how to use the Upgrade Wizard.
Upgrading Visual Basic 6.0 Code with the Upgrade Visual Basic 6 Code Dialog Box
Explains how to upgrade Visual Basic 6.0 code to Visual Basic .NET code.
Visual Basic 6.0 Code Upgrade Dialog Box
Describes the user interface for upgrading Visual Basic 6.0 code to Visual Basic .NET code.
Upgrading WebClass Projects to Visual Basic .NET
Explains how to upgrade WebClass projects (also known as IIS Applications) to ASP.NET Web Application projects.
Completing the Upgrading Process
Describes what to do next after the Upgrade Wizard has finished running.
Visual Basic Upgrade Report
Describes the format and purpose of the Upgrade Report.
Viewing the Upgrade Report
Explains how to open the upgrade report.
Upgrading a Project from the Command Line
Describes how to use the command-line upgrading tool.
Troubleshooting Applications Upgraded from Visual Basic 6.0
Lists some known issues that are not detected by the upgrading tools and explains how to address them.

Related Sections

What's New in Visual Basic and Visual C#
Describes the new features in Visual Basic .NET.
Introduction to Visual Basic .NET for Visual Basic Veterans
Lists the differences between Visual Basic 6.0 and Visual Basic .NET.
Working with Both Visual Basic 6.0 and Visual Basic .NET
Describes compatibility between the two versions of Visual Basic.

Why Upgrading Is Necessary

There were two options to consider when designing Visual Basic .NET — retrofit the existing code base to run on top of the .NET Framework, or build from the ground up, taking full advantage of the platform. To deliver the features most requested by customers (for example, inheritance, threading), to provide full and unconstrained access to the platform, and to ensure that Visual Basic moves forward into the next generation of Web applications, the right decision was to build from the ground up on the new platform.

For example, many of the new features found in Windows Forms (the new forms package) could have been added to the existing code base as new controls or more properties. However, this would have been at the cost of all the other great features inherent to Windows Application Objects, such as security and visual inheritance.

One of the major goals for Visual Basic was to ensure that code could fully interoperate with code written in other languages, such as C# or C++, and enable the Visual Basic developer to harness the power of the .NET Framework simply, without resorting to the programming workarounds traditionally required to make Windows APIs work. Visual Basic now has the same variable types, arrays, user-defined types, classes, and interfaces as Visual C++ and any other language that targets the common language runtime; however, some features such as fixed-length strings had to be removed from the language.

Visual Basic is now a true object-oriented language; some unintuitive and inconsistent features like GoSub/Return and DefInt have been removed from the language.

The result is a re-energized Visual Basic, which will continue to be the most productive tool for creating Windows-based applications and is now positioned to be the best tool for creating the next-generation Web sites. The tradeoff is that Visual Basic 6.0 applications must be upgraded to conform to the new language and forms features, breaking compatibility with earlier versions.

See Also

Preparing a Visual Basic 6.0 Application for Upgrading | Things to Consider Before Upgrading |
Upgrading Applications Created in Previous Versions of Visual Basic

Things to Consider Before Upgrading

Visual Basic .NET represents a major shift from previous versions of Visual Basic. Because it was designed from the ground up to take advantage of the .NET Platform, there are many areas where compatibility with previous versions is broken.

> **Note** Visual Basic .NET and Visual Basic 6.0 can be installed on the same computer and run at the same time. Applications written in Visual Basic .NET and Visual Basic 6.0 can be installed and executed on the same computer. Components written in Visual Basic .NET can interoperate with COM components written in earlier versions of Visual Basic and other languages.

In most cases you will want to upgrade your Visual Basic 6.0 applications to Visual Basic .NET so that you can take advantage of the benefits of the .NET Platform as you continue development. There are, however, situations where you may wish to continue development in Visual Basic 6.0 rather than upgrading. Some of the factors in deciding to upgrade an application include features that aren't supported in Visual Basic .NET, and the amount of rework needed due to incompatibility and architectural considerations.

- Unsupported Features
- Determining the Amount of Rework Required

Unsupported Features

The following features from Visual Basic 6.0 are no longer supported in Visual Basic .NET. For more information, see Preparing a Visual Basic 6.0 Application for Upgrading.

- **OLE Container Control** There is no equivalent in Visual Basic .NET. Applications that depend on this control should be left in Visual Basic 6.0.
- **Dynamic Data Exchange (DDE)** DDE methods are no longer supported. Applications that depend on DDE should either be revised to use another method of inter-application communication or should be left in Visual Basic 6.0.
- **DAO or RDO Data Binding** Data binding to a DAO or RDO data source isn't supported in Visual Basic .NET; the **Data** control and the **RemoteData** control have no equivalent. DAO and RDO can only be accessed via code. Applications that rely on DAO or RDO data binding should either be updated to use ADO or should be left in Visual Basic 6.0.
- **Visual Basic 5.0 Controls** Visual Basic 6.0 included the Visual Basic 5.0 versions of the Windows Common controls and the **Data-Bound Grid** control. These controls are incompatible with Visual Basic .NET. Applications that rely on these controls should either be updated to use the 6.0 versions or should be left in Visual Basic 6.0.
- **DHTML Applications** There is no equivalent in Visual Basic .NET. However, DHTML applications can interoperate with Visual Basic .NET technologies.
- **ActiveX Documents** There is no equivalent in Visual Basic .NET. However, DHTML applications can interoperate with Visual Basic .NET technologies. ActiveX documents should either be rewritten as User Controls, or left in Visual Basic 6.0.
- **Property Pages** There is no equivalent in Visual Basic .NET. If your application depends heavily on property pages, it should be left in Visual Basic 6.0.

Determining the Amount of Rework Required

Due to the many differences between Visual Basic 6.0 and Visual Basic .NET, most applications will require some rework for items that can't be mapped directly or are otherwise incompatible. The amount of rework required depends on a number of factors, including the type of application, features, and the language constructs used.

The easiest way to determine the amount of rework required is to run the Upgrade Wizard and view the list of issues in the Upgrade Report. When an application is upgraded a new copy is created; the original application is left intact. If, after viewing the report, you decide not to upgrade, you can simply delete the new project and continue development in Visual Basic 6.0.

The following are additional considerations that may factor into your upgrade decision.

- **Single-tier Database Applications** Because data binding to DAO isn't supported, simple applications using controls bound directly to a local database (such as one created with Microsoft Access) may require a significant amount of rework.
- **Visual Basic Add-ins** Because Visual Basic .NET uses the Visual Studio integrated development environment, the object model for extensibility is significantly different than that of Visual Basic 6.0. Add-ins will require a considerable amount of rework.
- **Games** Applications relying on specific performance characteristics of Visual Basic 6.0, such as arcade games, will require rework because Visual Basic .NET has different performance characteristics.

- **Graphics** There is no support for the graphics methods of forms or for the shape and line controls. Applications that use these features extensively to draw on forms will require considerable rework.
- **Drag and Drop Functionality** The models for drag and drop functionality are significantly different; any code performing drag and drop actions will need to be rewritten.
- **Variants** When an application is upgraded, the **Variant** data type is converted to **Object**. If an application is heavily dependent on variants, this conversion can introduce some subtle differences in the application's behavior. For example, in Visual Basic 6.0 the expression `IsObject(Variant)` evaluates to **false**; in Visual Basic .NET it changes to `IsObject(Object)`, which evaluates to **true**.
- **Windows APIs** Due to language changes, many calls to Windows APIs will need to be revised or replaced with .NET Framework functions.

See Also

Preparing a Visual Basic 6.0 Application for Upgrading | Why Upgrading is Necessary |
Upgrading Applications Created in Previous Versions of Visual Basic

Preparing a Visual Basic 6.0 Application for Upgrading

Although the process of upgrading a Visual Basic 6.0 application to Visual Basic .NET is for the most part automatic, there are a number of things you can do to optimize your upgrade with a little bit of preparation.

By following the recommendations in this topic, you can minimize and, in some cases, eliminate the changes needed after upgrading your project to Visual Basic .NET. In most cases, the recommendations simply represent good programming practices; however, some also identify the objects and methods which have no equivalents, and which should be used sparingly if you intend to upgrade your project to Visual Basic .NET.

- General Recommendations
- Considerations for Forms and Controls
- Data Recommendations
- Web Architecture Recommendations

General Recommendations

This may seem obvious, but if your project doesn't compile and run under Visual Basic 6.0, it will not be upgraded successfully. It is recommended that you install Visual Basic 6.0 on the computer that you will be using for upgrading. In addition to allowing you to test the application, this will ensure that all necessary controls, components, and type libraries are available during upgrade.

Visual Basic .NET allows you to upgrade Visual Basic 6.0 projects; if you have a project written in Visual Basic version 1 to 5, you should first load it into Visual Basic 6.0 (choosing to upgrade Microsoft ActiveX controls), compile, and save the project before upgrading it to Visual Basic .NET.

Considerations for Forms and Controls

Visual Basic .NET has a new forms package, Windows Forms. Windows Forms is largely compatible with the forms package found in Visual Basic 6.0; however, there are some key differences that are outlined below:

- Visual Basic .NET does not support the OLE container control; you should avoid using this control in your Visual Basic 6.0 applications.
- There is no shape control in Visual Basic .NET. Square and rectangular shapes will be upgraded to labels, while ovals and circles cannot be upgraded. You should avoid using these in your applications.
- There is no line control in Visual Basic .NET. Horizontal and vertical lines are upgraded to labels. Diagonal lines are not upgraded, and you should avoid using them.
- Visual Basic .NET has a new set of graphics commands that replace the Form methods **Circle**, **CLS**, **PSet**, **Line**, and **Point**. Because the new object model is quite different from Visual Basic 6.0, these methods cannot be upgraded.
- For the Timer control, setting the **Interval** property to 0 does not disable the timer; instead the interval is reset to 1. In your Visual Basic 6.0 projects, you should set **Enabled** to **False** instead of setting the **Interval** to 0.

Visual Basic .NET has two menu controls, **MainMenu** and **ContextMenu**, whereas Visual Basic 6.0 has one menu control that can be opened as either a **MainMenu** or a **ContextMenu** controls. All menu controls are upgraded to a single **MainMenu** component containing **MenuItems** for each menu control; you will have to recreate your **ContextMenus** and delete the extra **MainMenu** controls.

- Visual Basic .NET has no support for Dynamic Data Exchange (DDE).
- Visual Basic .NET does not support the **Form.PrintForm** method.
- Although Visual Basic .NET has support for drag-and-drop functionality, the object model is different from the one in Visual Basic 6.0. Therefore, the Visual Basic 6.0 drag-and-drop properties and methods cannot be upgraded.
- The .NET Framework has an improved **Clipboard** object (**System.Windows.Forms.Clipboard**) that offers more functionality and supports more clipboard formats than the Visual Basic 6.0 **Clipboard** object. However, because of differences between object models, clipboard statements cannot be automatically upgraded.
- Visual Basic .NET does not support the **Name** property for forms and controls at run time; therefore you should not write code that iterates the **Controls** collection looking for a control with a certain name (this functionality is now available using the .NET **System.Reflection** classes.)

Data Recommendations

Visual Basic .NET introduces an enhanced version of ADO called ADO.NET that is optimized to work with data in distributed

applications and that provides performance improvements over ADO when used in distributed applications.

DAO, RDO, and ADO can still be used in code from Visual Basic .NET, with some trivial modifications (covered in the language section of this document). However, Visual Basic .NET does not support DAO and RDO data binding to controls, data controls, or RDO User connection. We recommend that if your applications contain DAO or RDO data binding you either leave them in Visual Basic 6.0 or upgrade the DAO or RDO data binding to ADO before upgrading your project to Visual Basic .NET, as ADO data binding is supported in Windows Forms. Information on how to upgrade DAO or RDO to ADO in Visual Basic 6.0 is available in the Visual Basic 6.0 Help.

Web Architecture Recommendations

Visual Basic .NET introduces ASP.NET, an enhanced version of ASP, and adds to the architecture with Web Forms, a technology for producing Web pages with a Visual Basic-like event model. The architecture is server-based.

For Web-based applications, applications that conform to the Microsoft multitier architecture guidelines, use Active Server Pages (ASP), and use Visual Basic 6.0 or Visual C++ 6.0 COM objects for business logic will be the easiest to upgrade. ASP is fully supported in Visual Basic .NET, and you can continue to extend your application using ASP, ASP.NET, and Web Forms. The Visual Basic 6.0 and Visual C++ 6.0 business objects can either be used without modification or upgraded to Visual Studio .NET.

Webclasses no longer exist in Visual Basic .NET. Existing Webclass applications can interoperate with Visual Basic .NET Web Forms and ASP applications; or they can be upgraded to Web Forms applications.

See Also

Things to Consider Before Upgrading | Why Upgrading is Necessary |
Upgrading Applications Created in Previous Versions of Visual Basic | Language Recommendations for Upgrading

Language Recommendations for Upgrading

Differences in language between Visual Basic 6.0 and Visual Basic .NET necessitate changes to code when an application is upgraded. The following are some recommendations you can follow when preparing a Visual Basic 6.0 application for upgrade, or when developing Visual Basic 6.0 applications that might later be upgraded.

In This Section

Upgrade Recommendation: Use Early Binding and Explicit Conversions

Upgrade Recommendation: Use the Date Data Type for Storing Dates

Upgrade Recommendation: Resolve Parameterless Default Properties

Upgrade Recommendation: Avoid Null Propagation

Upgrade Recommendation: Use Zero-Bound Arrays

Upgrade Recommendation: Use Constants Instead of Underlying Values

Upgrade Recommendation: Avoid Arrays and Fixed-Length Strings in User-Defined Types

Upgrade Recommendation: Avoid Legacy Keywords

Upgrade Recommendation: Adjust Data Types for Win32 APIs

Related Sections

Preparing a Visual Basic 6.0 Application for Upgrading
Describes how to make sure that your application is ready to be upgraded.

Things to Consider Before Upgrading
Discusses the limitations of and considerations for upgrading.

Upgrading Applications Created in Previous Versions of Visual Basic
Discusses upgrading your Visual Basic 6.0 applications.

Upgrade Recommendation: Use Early Binding and Explicit Conversions

Both Visual Basic 6.0 and Visual Basic .NET support late-bound objects, which is the practice of declaring a variable as the **Object** data type and assigning it to an instance of a class at run time. However, during the upgrade process, late-bound objects can introduce problems when resolving default properties, or in cases where the underlying object model has changed and properties, methods, and events need to be converted. For example, suppose you have a form called Form1 with a label called Label1; the following Visual Basic 6.0 code would set the caption of the label to "SomeText":

```
Dim o As Object
Set o = Me.Label1
o.Caption = "SomeText"
```

In Visual Basic .NET, the **Caption** property of a label control is replaced by the **Text** property. When your code is upgraded, all instances of the **Caption** property are changed to **Text**, but because a late-bound object is type-less, Visual Basic cannot detect what type of object it is, or if any properties should be translated. In such cases, you will need to change the code yourself after upgrading.

If you rewrite the code using early-bound objects, it will be upgraded automatically:

```
Dim o As Label
Set o = Me.Label1
o.Text = "SomeText"
```

Where possible you should declare variables of the appropriate object type rather than simply declaring them as the **Object** data type.

In the cases where you do use Object and Variant variables in your Visual Basic 6.0 code, we recommend you use explicit conversions when you assign the variables, perform operations on the variables, or pass the variables to a function. For example, the intention of the '+' operation in the following code is unclear:

```
Dim Var1 As Variant
Dim Var2 As Variant
Dim Var3 As Variant
Var1 = "3"
Var2 = 4
'BAD: Should Var1 and Var2 be added as strings or integers?
Var3 = Var1 + Var2
```

The above example may result in a run-time error in Visual Basic .NET. Rewriting the final line to use explicit conversions ensures the code will work:

```
'GOOD: explicit conversion
Var3 = CInt(Var1) + CInt(Var2)
```

Visual Basic .NET supports overloading functions based on parameter type. For example, the **Environ** function now has two forms:

```
Environ( Expression As Integer) As String
Environ( Expression As String ) As String
```

Visual Basic .NET determines which function to call based on the parameter type. If you pass an integer to **Environ()**, the integer version is called; if you pass a string, then the string version is called. Code that passes a **Variant** or **Object** data type to an overloaded function may cause a compile or run-time error. Using an explicit conversion, as in the following example, will mean your code will work as intended after it is upgraded to Visual Basic .NET:

```
Dim a As String
Dim v As Variant
v = "Path"
```

```
'GOOD: explicit conversion
a = Environ(CStr(v))
```

Using explicit conversions of late-bound objects is good coding practice. It makes the intention of the code easy to determine, and makes it easier for you to upgrade your project to Visual Basic .NET.

See Also

Language Recommendations for Upgrading

Upgrade Recommendation: Use the Date Data Type for Storing Dates

Earlier versions of Visual Basic supported using the **Double** data type to store and manipulate dates. You should not do this in Visual Basic .NET, because dates are not internally stored as doubles. For example, the following is valid in Visual Basic 6.0, but may cause a compile error in Visual Basic .NET:

```
Dim dbl As Double
Dim dat As Date
dat = Now
'BAD: Date can't be assigned to a double
dbl = dat
'BAD: Double can't be used in date functions
dbl = DateAdd("d", 1, dbl)
'BAD: CDate can't convert a double to a date
dat = CDate(dbl)
```

The .NET Framework provides the **ToOADate** and **FromOADate** functions to convert between doubles and dates. However, when your project is upgraded to Visual Basic .NET, it is difficult to determine the intention of code that uses doubles to store dates. To avoid unnecessary modifications to your code in Visual Basic .NET, always use the **Date** data type to store dates.

See Also

Language Recommendations for Upgrading

Upgrade Recommendation: Resolve Parameterless Default Properties

In Visual Basic 6.0, many objects expose default properties, which can be omitted as a programming shortcut. For example, **TextBox** has a default property of **Text**, so instead of writing:

```
MsgBox Form1.Text1.Text
```

you use the shortcut:

```
MsgBox Form1.Text1
```

The default property is resolved when the code is compiled. In addition, you could also use default properties with late-bound objects, as in the following example:

```
Dim obj As Object
Set obj = Form1.Text1
MsgBox obj
```

In the late-bound example, the default property is resolved at run time, and the **MsgBox** displays the value of the default property of the **TextBox** as **Text1**.

Visual Basic .NET does not support parameterless default properties, and consequently does not allow this programming shortcut. When your project is upgraded, Visual Basic .NET resolves the parameterless default properties, but late-bound usages that rely on run-time resolution cannot be automatically resolved. In these cases, you will have to change the code yourself. An additional complication is that many libraries implement default properties using a property called **_Default**. **_Default** acts as a proxy, passing calls to the real default property. So, when your project is upgraded, some default properties will be resolved to **_Default**. The code will still work as usual, but it will be less understandable than code written explicitly using the actual property. For these reasons, try to avoid using parameterless default properties in your Visual Basic 6.0 code. Instead of writing:

```
Dim obj As Object
Set obj = Me.Text1
'BAD: Relying on default property
MsgBox obj
'BAD: Relying on default property
MsgBox Me.Text1
```

use:

```
Dim obj As TextBox
Set obj = Me.Text1
'GOOD: Default property is resolved
MsgBox obj.Text
'GOOD: Default property is resolved
MsgBox Me.Text1.Text
```

While *parameterless* default properties are not supported in Visual Basic .NET, default properties *with parameters* are supported. To understand the difference between the two types, consider that parameterized default properties always have an index. An example is the default property of ADO **recordset**: the **Fields** collection. The code:

```
Dim rs As ADODB.Recordset
rs("CompanyName") = "SomeCompany"
rs!CompanyName = "SomeCompany"
```

is actually a shortcut for:

```
Dim rs As ADODB.Recordset
rs.Fields("CompanyName").Value = "SomeCompany"
```

```
rs.Fields!CompanyName.Value = "SomeCompany"
```

In this case, the **Fields** property is parameterized, and so the usage is valid in Visual Basic .NET; however, the default property of the **Fields** property, **Value,** is parameterless, so the correct usage in Visual Basic .NET is:

```
Dim rs As ADODB.Recordset
rs("CompanyName").Value = "SomeCompany"
rs!CompanyName.Value = "SomeCompany"
```

This example and most other default properties are resolved for you when the project is upgraded, so resolving them in Visual Basic 6.0 is simply a good programming practice. However, you should avoid using default properties with the **Object** and **Variant** data types, as these cannot be resolved and you will have to fix the code yourself in the upgraded project.

See Also

Language Recommendations for Upgrading

Upgrade Recommendation: Avoid Null Propagation

Previous versions of Visual Basic supported Null propagation. Null propagation supports the premise that when Null is used in an expression, the result of the expression will itself be Null. In each case in the following example, the result of `V` is always Null.

```
Dim V
V = 1 + Null
V = Null + Right$("SomeText", 1)
V = Right("SomeText", 0)
```

Null propagation is not supported in Visual Basic .NET. The statement `1+Null` will generate a type mismatch in Visual Basic .NET. Additionally, where Visual Basic 6.0 had two versions of the Left function — **Left$** returning a string, **Left** returning a variant which could be Null — Visual Basic .NET only has one version, **Left**, which always returns a string.

In order to be compatible with both Visual Basic 6.0 and Visual Basic .NET, you should always write code to test for Null instead of relying on Null propagation. Furthermore, in Visual Basic .NET, the following functions will no longer return Null:

Chr, **Command**, **CurDir**, **Date**, **Environ**, **Error**, **Hex**, **LCase**, **LTrim**, **Oct**, **Right**, **RTrim**, **Space**, **Str**, **Time**, **Trim**, **UCase**.

Null propagation is commonly used in database applications, where you need to check if a database field contains Null. In these cases you should check results using the function **IsNull()** and perform the appropriate action.

See Also

Language Recommendations for Upgrading

Upgrade Recommendation: Use Zero-Bound Arrays

Visual Basic 6.0 allowed you to define arrays with lower and upper bounds of any whole number. You could also use **ReDim** to reassign a variant as an array. To enable interoperability with other languages, arrays in Visual Basic .NET must have a lower bound of zero, and **ReDim** cannot be used unless the array variable was previously declared with the **Dim** keyword. Although this restricts the way arrays can be defined, it does allow you to pass arrays between Visual Basic .NET and any other .NET language. The following example shows the restriction:

```
'BAD: LBound must be 0
Dim a(1 To 10) As Integer
'BAD: Can't use ReDim without Dim
ReDim v(10)
'OK: Creates an array of 11 integers
Dim b(10) As Integer
'OK: Can ReDim previously Dimed variable
ReDim b(5) As Integer
```

A side effect is that the **Option Base** statement is removed from the language.

When your project is upgraded to Visual Basic .NET, any option base statements are removed from your code. If the array is zero-bound, it is left unchanged. However, if an array is non-zero–bound, then the lower bound is removed and a warning is inserted into the code, as in the following example:

```
Dim a(1 To 10) As Integer
```

changes to:

```
' UPGRADE_WARNING: Lower bound of an array was changed from 1 to 0.
Dim a(10) As Integer
```

In many cases, the upgraded code will work as it did before. However, if your application logic relies on the lower bound being 1, then you will need to make some modifications. **Dim**, **ReDim**, and **LBound** statements are marked with warnings to help you review the changes.

For this reason, you should use zero-bound arrays in your Visual Basic 6.0 code, avoid using **ReDim** as an array declaration, and avoid using **Option Base** 1.

See Also

Language Recommendations for Upgrading

Upgrade Recommendation: Use Constants Instead of Underlying Values

When writing code, try to use constants rather than relying on their underlying values. For example, if you are maximizing a form at run time, use:

```
' GOOD: Constant name
Me.WindowState = vbMaximized
```

rather than:

```
'BAD: Underlying value
Me.WindowStyle = 2
'BAD: Variable
Me.WindowStyle = X
```

Likewise, use **True** and **False** instead of -1 and 0.

In Visual Basic .NET, the values and in some cases the names of some properties and constants have changed; for example, the value of **True** changes from -1 to 1. When your project is upgraded to Visual Basic .NET, most constants are changed automatically for you; however, if you use underlying values or variables instead of the constant names, many cases cannot be upgraded automatically. Using constant names minimizes the number of modifications you have to do.

See Also

Language Recommendations for Upgrading

Upgrade Recommendation: Avoid Arrays and Fixed-Length Strings in User-Defined Types

Due to changes made which allow Visual Basic .NET arrays and structures (formerly known as user-defined types) to be fully compatible with other Visual Studio .NET languages, fixed-length strings are no longer supported in the language. In most cases this is not a problem, because there is a compatibility class which provides fixed-length string behavior, so the code:

```
Dim MyFixedLengthString As String * 100
```

upgrades to the following:

```
Dim MyFixedLengthString As New VB6.FixedLengthString(100)
```

However, fixed-length strings do cause a problem when used in structures. The problem arises because the fixed-length string class is not automatically created when the structure is created. Likewise, fixed-size arrays are not created when the structure is created.

When your code is upgraded, user-defined types with fixed-length strings or arrays will be converted to structures and marked with a comment telling you to initialize the fixed-length string or array before referencing the structure in code. However, you can shield yourself from this modification by changing your Visual Basic 6.0 user-defined types to use strings instead of fixed-length strings, and uninitialized arrays instead of fixed-size arrays. For example:

```
Private Type MyType
    MyArray(5) As Integer
    MyFixedString As String * 100
End Type
Sub Bar()
    Dim MyVariable As MyType
End Sub
```

can be changed to:

```
Private Type MyType
    MyArray() As Integer
    MyFixedString As String
End Type
Sub Bar()
    Dim MyVariable As MyType
    ReDim MyVariable.MyArray(5) As Integer
    MyVariable.MyFixedString = String$(100, " ")
End Sub
```

See Also

Language Recommendations for Upgrading

Upgrade Recommendation: Avoid Legacy Keywords

Because they have been removed from the language, you should avoid using the following keywords.

Keyword	Explanation
Def<*type*>	In previous versions of Visual Basic, **DefBool**, **DefByte**, **DefInt**, **DefLng**, **DefCur**, **DefSng**, **DefDbl**, **DefDec**, **DefDate**, **DefStr**, **DefObj** and **DefVar** were used in the declarations section of a module to define a range of variables as a certain type. For example `DefInt A-C` defined all variables beginning with the letter A, B, or C as an integer. Instead of using **Def**<type> statements, you should explicitly declare variables.
Computed **GoTo/GoSub**	Computed **GoTo**/**GoSub** statements take this form: `On x GoTo 100, 200, 300` These are not supported in Visual Basic .NET. Instead, you should use **If** statements, or **Select Case** constructs.
GoSub/Return	**GoSub** and **Return** statements are not supported in Visual Basic .NET. In most cases you can replace these with functions and procedures.
Option Base 0|1	The **Option Base** statement was used to specify the default lower bound of an array as either 0 or 1. This statement has been removed from the language since Visual Basic .NET natively only supports arrays with a zero lower bound. Non-zero lower bound arrays are supported through the VB6.Compatibility class.
VarPtr, **ObjPtr**, **StrPtr**	**VarPtr**, **VarPtrArray**, **VarPtrStringArray**, **ObjPtr** and **StrPtr** were undocumented functions used to get the underlying memory address of variables. These functions are not supported in Visual Basic .NET.
LSet	In Visual Basic 6.0, the **LSet** statement could be used to assign a variable of one user-defined type to another variable of a different user-defined type. This functionality is not supported in Visual Basic .NET.

See Also

Language Recommendations for Upgrading

Upgrade Recommendation: Adjust Data Types for Win32 APIs

Many APIs can be used exactly as they were in Visual Basic 6.0, with the caveat that you have to adjust your data types accordingly. The Visual Basic 6.0 **Long** data type is now the Visual Basic .NET **Integer** data type, and the Visual Basic 6.0 **Integer** data type is now the Visual Basic .NET **Short** data type. During the upgrade, these changes are made for you, and simple APIs work exactly the same as they did in Visual Basic 6.0. For example:

```
Private Declare Function GetVersion Lib "kernel32" () As Long
Function GetVer()
    Dim Ver As Long
    Ver = GetVersion()
    MsgBox ("System Version is " & Ver)
End Function
```

changes to:

```
Private Declare Function GetVersion Lib "kernel32" () As Integer
Function GetVer()
    Dim Ver As Integer
    Ver = GetVersion()
    MsgBox("System Version is " & Ver)
End Function
```

In addition to numeric data type upgrades, Visual Basic 6.0 had a fixed-length string data type which is not supported in Visual Basic .NET, and which is upgraded to a fixed-length string wrapper class. In many cases in Visual Basic 6.0 you can perform the same action using a normal string. For example:

```
Private Declare Function GetUserName Lib "advapi32.dll" Alias _
"GetUserNameA" (ByVal lpBuffer As String, ByRef nSize As Long) As Long
Function GetUser()
    Dim Ret As Long
    Dim UserName As String
    Dim Buffer As String * 25
    Ret = GetUserName(Buffer, 25)
    UserName = Left$(Buffer, InStr(Buffer, Chr(0)) - 1)
    MsgBox (UserName)
End Function
```

can be better written using a normal string explicitly set to length 25 instead of a fixed-length string:

```
Dim Buffer As String
Buffer = String$(25, " ")
```

This is upgraded to Visual Basic .NET as follows:

```
Declare Function GetUserName Lib "advapi32.dll" Alias _
"GetUserNameA" (ByVal lpBuffer As String, ByRef nSize As Integer) As Integer
Function GetUser()
    Dim Ret As Integer
    Dim UserName As String
    Dim Buffer As String
    Buffer = New String(CChar(" "), 25)
    Ret = GetUserName(Buffer, 25)
    UserName = Left(Buffer, InStr(Buffer, Chr(0)) - 1)
    MsgBox(UserName)
End Function
```

In some cases, Visual Basic .NET better handles passing strings to APIs, since you can optionally declare how you want strings to be passed using the **ANSI** and **UNICODE** keywords.

There are three cases where you may need to make some changes. The first is passing user-defined types that contain fixed-

length strings or byte arrays to APIs. In Visual Basic .NET you may need to change your code, adding the **MarshallAs** attribute (from **System.Runtime.InteropServices**) to each fixed-length string or byte array in the user-defined type. The second case is using the **As Any** variable type in a **Declare** statement. This is not supported in Visual Basic .NET. Variables of type **As Any** were often used to pass a variable that was either a string or Null; you can replace this Visual Basic 6.0 usage by declaring two forms of the API, one with longs, one with strings. For example, the GetPrivateProfileString API has a parameter lpKeyName of type **As Any**:

```
Private Declare Function GetPrivateProfileString Lib "kernel32" Alias _
"GetPrivateProfileStringA" (ByVal lpApplicationName As String, ByVal _
lpKeyName As Any, ByVal lpDefault As String, ByVal lpReturnedString _
As String, ByVal nSize As Long, ByVal lpFileName As String) As Long
```

You can remove the "**As Any**" by replacing the **Declare** with two versions, one that accepts a long, and one that accepts a string:

```
Private Declare Function GetPrivateProfileStringKey Lib "kernel32" Alias _
"GetPrivateProfileStringA" (ByVal lpApplicationName As String, ByVal _
lpKeyName As String, ByVal lpDefault As String, ByVal lpReturnedString _
As String, ByVal nSize As Long, ByVal lpFileName As String) As Long

Private Declare Function GetPrivateProfileStringNullKey Lib "kernel32" _
Alias "GetPrivateProfileStringA" (ByVal lpApplicationName As String, _
ByVal lpKeyName As Long, ByVal lpDefault As String, ByVal _
lpReturnedString As String, ByVal nSize As Long, ByVal lpFileName _
As String) As Long
```

When you wish to pass the value Null to the API, you use the GetPrivateProfileStringNullKey version. Doing it this way means that the function upgrades to Visual Basic .NET.

The final area where you may need to make some changes is if you are using APIs that perform thread creation, Windows subclassing, message queue hooking, and so on. Some of these functions will cause a run-time error in Visual Basic .NET. Many of these APIs have equivalents in Visual Basic .NET or the .NET Framework. You will have to fix these on a case-by-case basis.

See Also

Language Recommendations for Upgrading

Visual Basic 6.0 Compatibility Library

If you have used previous versions of Visual Basic, you will notice that Visual Basic .NET introduces several changes to the Visual Basic language and that some familiar functions have seemingly disappeared. These changes were necessary in order to make Visual Basic compliant with the .NET Platform and the common language specification (CLS).

When a Visual Basic 6.0 application is upgraded to Visual Basic .NET, the conversion of some code is impossible due to syntactical or architectural differences. For this reason, functions in the Visual Basic 6.0 Compatibility library (Microsoft.VisualBasic.Compatibility) are used to allow the code to run in Visual Basic .NET without the need for extensive modification. These functions mimic Visual Basic 6.0 behavior while remaining compliant with the common language specification.

> **Caution** Functions in the Visual Basic 6.0 Compatibility library are provided only for use by the upgrading tools. Although it is possible to use this library when writing new code, there is no guarantee that it will be supported in future versions of Visual Basic.

See Also

Language Changes in Visual Basic | Working with Both Visual Basic 6.0 and Visual Basic .NET | Upgrading from Visual Basic 6.0

Visual Basic Upgrade Wizard

Visual Basic .NET enables a fundamental shift from traditional Windows development to building next-generation Web and n-tier applications. For this reason, your code will need to be upgraded to take advantage of Visual Basic .NET.

This happens automatically when you open a Visual Basic 6.0 project in Visual Basic .NET: the Upgrade Wizard steps you through the upgrade process and creates a new Visual Basic .NET project (your existing project is left unchanged). This is a one-way process; the new Visual Basic .NET project cannot be opened in Visual Basic 6.0.

When your project is upgraded, the language is modified for any syntax changes and your Visual Basic 6.0 Forms are converted to Windows Forms. In most cases, you will have to make some changes to your code after it is upgraded. This is required because certain objects and language features either have no equivalent in Visual Basic .NET, or have an equivalent too dissimilar for an automatic upgrade. After the upgrade, you may also want to change your application to take advantage of some of the new features in Visual Basic .NET.

To help you make the changes, after your project is upgraded, Visual Basic .NET adds an upgrade report to your project itemizing any problems, and inserts comments into your upgraded code alerting you to statements that will need to be changed. Because these comments are displayed as 'TO DO' tasks in the new Task List window, you can easily see what changes are required, and navigate to the code statement simply by double-clicking the task. Each task and item in the upgrade report is associated with an online Help topic giving further guidance as to why the code needs to be changed, and what you need to do

See Also

Completing the Upgrading Process | Why Upgrading is Necessary | Preparing a Visual Basic 6.0 Application for Upgrading | Upgrading Applications Created in Previous Versions of Visual Basic

Upgrading a Project with the Upgrade Wizard

The Visual Basic Upgrade Wizard can be used to upgrade Visual Basic 6.0 projects for further development in Visual Basic .NET. The Upgrade Wizard does not modify your original Visual Basic 6.0 project, but rather creates a new Visual Basic .NET project based on the original project.

> **Note** Although the wizard makes the upgrade process easier, you may be required to modify some code in the upgraded application. You should take some time to familiarize yourself with Visual Basic .NET before attempting this process.

To upgrade a Visual Basic 6.0 project to Visual Basic .NET

1. Open your project in Visual Basic 6.0. On the **File** menu, choose **Make *projectname***. Run the application and make sure that there are no errors.
2. Start Visual Studio .NET. On the **File** menu, choose **Open**, **Project**.
3. In the **Open Project** dialog box, browse to the location of the Visual Basic 6.0 project and select the project (.vbp) file.

   This will launch the Visual Basic Upgrade Wizard.

4. On the second page of the wizard, choose from the available upgrade options. The options will vary depending on the type of project being upgraded.

   > **Tip** For an explanation of available options, press **F1** while in the wizard.

5. On the third page of the wizard, enter the location where the new project will be created.

   > **Note** Each Visual Basic .NET project must be in its own folder. If you specify a folder that contains files, you will be prompted to choose a different folder.

6. On the fourth page of the wizard, click **Next** to begin the upgrade process. When the process is complete, the new project will appear in Solution Explorer.
7. In Solution Explorer, double-click the **_UpgradeReport.htm** node to open the upgrade report.
8. Review the upgrade report. Any errors must be fixed before you can run the application. Any warnings should be examined to make sure that they do not affect application behavior.

   > **Tip** For an explanation of any error or warning, click on the description to display a Help topic.

9. On the **Debug** menu, choose **Start.** Run the application and make sure that there are no errors and that the behavior is the same as it was in Visual Basic 6.0.

   > **Note** The Upgrade Wizard cannot catch all possible incompatibilities; it is possible that you will encounter additional errors when you run the application. For a list of known issues, see Troubleshooting Applications Upgraded from Visual Basic 6.0.

See Also

Upgrading Applications Created in Previous Versions of Visual Basic | Viewing the Upgrade Report | Preparing a Visual Basic 6.0 Application for Upgrading | WebClass Changes in Visual Basic .NET

Visual Basic 6.0 Code Upgrade Dialog Box

The **Visual Basic 6.0 Code Upgrade** dialog box is used to upgrade Visual Basic 6.0 code to Visual Basic .NET. Code can be typed into the **Code** window or pasted from the Visual Basic 6.0 Code Editor. When you click the **Upgrade** button, the upgraded code is inserted into the Visual Basic .NET Code Editor at the current cursor location.

In addition, COM references required by the Visual Basic 6.0 code can be added to the **References** tab; these are added to the References in your Visual Basic .NET project.

To access the **Visual Basic 6.0 Code Upgrade** dialog box, choose **Upgrade Visual Basic 6 Code** from the **Tools** menu.

> **Note** The **Upgrade Visual Basic 6 Code** command is only available when working in the Code Editor.

Code
Visual Basic 6.0 code can be typed into the code window or pasted from the Visual Basic 6.0 Code Editor. Code is retained between uses so that you can further modify the code if necessary.

References
Displays COM references required by Visual Basic 6.0 code that you have added.

Reference Name
Displays the friendly name of the COM component. Only references that are checked will be added to the Visual Basic .NET project.

Version
Displays the version number (major and minor) of the COM component.

Path
Displays the file path and filename for the COM component.

Add References
Displays the **Add Reference** dialog box that contains a list of available COM components.

Upgrade
Upgrades any code entered into the Code window and inserts it in the Visual Basic .NET Code Editor; also adds any checked references in the References list to the Visual Basic .NET project.

See Also

Upgrading Visual Basic 6.0 Code with the Upgrade Visual Basic 6 Code Dialog Box |
Upgrading Applications Created in Previous Versions of Visual Basic

Upgrading Visual Basic 6.0 Code with the Upgrade Visual Basic 6 Code Dialog Box

The Upgrade Visual Basic 6 Code dialog box is a tool that converts Visual Basic 6.0 code to equivalent Visual Basic .NET code. Unlike the Visual Basic Upgrade Wizard, which upgrades an entire application, the Upgrade Visual Basic 6 Code dialog box is available at design time to convert small amounts of code as you are developing your Visual Basic .NET application.

The Upgrade Visual Basic 6 Code dialog box is useful in two ways:

- You can make use of existing Visual Basic 6.0 code by pasting it into the tool.
- You can type in code using Visual Basic 6.0 syntax in order to learn the Visual Basic .NET equivalent.

After running the tool, Visual Basic .NET code is inserted into the Code Editor at the current cursor location. If there are any problems with the conversion, comments are inserted with clickable links to help topics that explain the problem and any possible workarounds.

To upgrade Visual Basic 6.0 code

1. In the Code Editor, move your cursor where you want to insert code.
2. On the **Tools** menu, choose **Upgrade Visual Basic 6 Code**.

   The **Visual Basic 6.0 Code Upgrade** dialog box will be displayed.

3. In the **Visual Basic 6.0 Code Upgrade** dialog box, paste in existing Visual Basic 6.0 code, or enter code using Visual Basic 6.0 syntax.
4. If your code references a COM component, select the **References** tab and click **Add Reference**, then select the reference in the **Add Reference** dialog box.
5. Click **Upgrade** to convert the code and insert it into the Visual Basic .NET Code Editor.

   Any COM references will be added to the References node for your project in Solution Explorer, and a project-level Imports statement will be added for the Microsoft.VisualBasic.Compatibility namespace.

Tips for Using the Upgrade Visual Basic 6 Code Dialog Box

The Upgrade Visual Basic 6 Code dialog box has some limitations; not all code can be successfully converted. The following tips can help you to get the most out of the tool.

- Not all Visual Basic 6.0 code is compatible with Visual Basic .NET. For more information about the limitations, see Things to Consider Before Upgrading.
- How your Visual Basic 6.0 code is written can have an effect on how it is upgraded. There are a number of recommendations that you can follow that will make your code easier to upgrade. For more information, see Language Recommendations for Upgrading.
- If the code does not run successfully in Visual Basic 6.0, it will not run in Visual Basic .NET. When pasting code from a Visual Basic 6.0 project, you should first test the code before attempting to upgrade.
- Event signatures in Visual Basic .NET are different than their Visual Basic 6.0 counterparts. When entering code for form or control event procedures, it is best to first add the procedure declaration in Visual Basic .NET and then upgrade only the body of the procedure.
- The Upgrade Visual Basic 6 Code dialog box is designed to upgrade small sections of code. It relies on the code typed into the dialog box to determine context for the upgrade.

  For example, the statement `L.Caption = "MyCaption"` would not be translated because the tool does not know what type "L" is, so it cannot resolve the property "Caption". Inserting a declaration for "L" (Dim L As Label) prior to the statement would result in a proper upgrade.

- Entering invalid Visual Basic 6.0 syntax can result in an "Untranslated statement in WrapperSub" error. If you see this error, inspect your code to make sure the syntax is correct.
- For best performance, upgrade code in modules rather than in Form modules. If you need to upgrade a form, use the Upgrade Wizard.

See Also

Upgrading WebClass Projects to Visual Basic .NET

Visual Basic 6.0 WebClass projects (also known as IIS Application projects) can be upgraded to ASP.NET Web Application projects using the Upgrade wizard. The process for upgrading WebClass projects is essentially the same as for any other project type; however, there are some issues that you will want to be aware of.

In order to upgrade a WebClass project, there are two prerequisites:

- Internet Information Services (IIS) must be installed and running on the upgrade computer
- You must have Administrative privileges that allow you to install applications.

When a WebClass project is upgraded, by default the project will be created with a new project name of *projectname*.NET (where *projectname* is the name of the Visual Basic 6.0 project). This name is then used when naming the virtual directory. This virtual directory will also be configured as an application in IIS; the Application Friendly name will be the *projectname*.

If the virtual directory name already exists on the http://localhost server (typical in a repeat upgrade scenario), a number will be appended to the virtual directory and project names in order to ensure uniqueness.

When a Visual Basic 6.0 WebClass project is upgraded to Visual Basic .NET, the .asp file for the project is upgraded to an .aspx file. Any references to the .asp file within an HTML template file are not automatically changed to .aspx references. These are not upgraded because a template file might contain references to other .asp files that were not part of the WebClass project.

In addition, when files are copied to the new project directory during upgrade, only HTML template files are copied. Any other .html files or image files are not copied to the new directory.

When adding HTML files to a Visual Basic .NET project, they are added as Content files by default. When a WebClass project is upgraded, HTML files are added as Embedded Resources. If you add HTML Files to the project after upgrading, you must set their **Build Action** property to **Embedded Resource** in order to make them visible to the application.

> **Tip** You can still reference an HTML file that is not embedded by providing a full path to the file when calling **WriteTemplate**.

If you have never created an ASP.NET Web Application before upgrading your first Visual Basic 6.0 WebClass project, you will need to set the Visual Basic .NET project's **StartUp** object by right-clicking the .aspx file and selecting **Set As Start Page** from the shortcut menu before running the upgraded project.

See Also

WebClass Changes in Visual Basic .NET | Upgrading Applications Created in Previous Versions of Visual Basic

Completing the Upgrading Process

After running the Upgrade Wizard, your upgraded project will be opened in Visual Studio .NET. The project will appear in Solution Explorer; it will include upgraded copies of all project files plus an upgrade report.

In most cases, there is additional work that will need to be done to ensure that your project will run successfully in Visual Basic .NET. There are three categories of work items that must be completed:

- UPGRADE_ISSUE errors are items that will prevent the application from compiling. You must correct these errors before running the project. These are reported in the upgrade report as and also appear as items in the Task List.
- UPGRADE_TODO errors are items that will not keep the project from compiling but that will cause run-time errors. You must correct these errors before running the project. These are reported in the upgrade report, as items in the Task List, and as comments in the upgraded code.
- UPGRADE_WARNING errors are items that will not keep the project from compiling but that may cause run-time errors. You should investigate these and correct them as necessary. These are reported in the upgrade report, as items in the Task List, and as comments in the upgraded code.
- UPGRADE_NOTE items are items that represent major changes to code or behavioral differences, which could have potential side effects. You should read these notes and decide if you need to make changes. These are reported as comments in the code.

To complete the upgrade process, you will need to check the Task List for any errors or warnings and search your code for any comments that begin with "UPGRADE_NOTE".

> **Tip** Every comment line inserted into your code during upgrade contains a hyperlink. Clicking the link (CTRL+click) takes you to a Help topic that explains the issue and what you need to do to address it. The link appears at the end of the comment line; you may need to scroll in order to see it.

See Also

Visual Basic Upgrade Report | Task List Window | Searching and Replacing | Visual Basic Upgrade Wizard | Upgrading Applications Created in Previous Versions of Visual Basic

Visual Basic Upgrade Report

When a Visual Basic 6.0 project is upgraded to Visual Basic .NET, an upgrade report is added to the project in Solution Explorer. The upgrade report contains information about the upgrade process as well as a list of issues uncovered during the upgrade that will need to be addressed before your project can be run.

The upgrade report is in HTML format; you can view it within the Visual Studio .NET development environment by double-clicking the file in Solution Explorer, or you can view it in an external browser by choosing **Browse With** from the **File** menu.

The first portion of the report contains general information about the upgrade, including the settings that were used during the upgrade and the location of the new project files.

The remainder of the report contains a listing of global issues plus a listing for each file that was upgraded. By expanding the section for each file, you can view a detailed list of issues that will need to be addressed. Each issue includes information about the severity of the issue, the location of the code that must be fixed, and a description of the issue.

See Also

Completing the Upgrading Process | Visual Basic Upgrade Wizard |
Upgrading Applications Created in Previous Versions of Visual Basic

Viewing the Upgrade Report

When a Visual Basic 6.0 project is upgraded to Visual Basic .NET, an upgrade report is created and placed in the new project folder. The upgrade report contains summary information about the upgrade process plus information on any errors or issues encountered during upgrade.

> **Note** The upgrade report is only generated for projects upgraded using the Upgrade Wizard; no reported is generated for projects upgraded from the command line.

The upgrade report can be viewed in the Visual Studio .NET integrated development environment (IDE) or in any Web browser.

To view the upgrade report in Visual Studio .NET

- In Solution Explorer, double-click the **_UpgradeReport.htm** node to open the upgrade report.

To view the upgrade report in a Web browser

1. In Solution Explorer, select the **_UpgradeReport.htm** node.
2. On the **File** menu, choose **Browse With**.
3. In the **Browse With** dialog box, select the Web browser.

See Also

Upgrading Applications Created in Previous Versions of Visual Basic | Preparing a Visual Basic 6.0 Application for Upgrading

Upgrading a Project from the Command Line

In addition to the Visual Basic Upgrade Wizard, Visual Basic 6.0 projects can be upgraded to Visual Basic .NET using a command-line tool, Vbupgrade.exe.

> **Note** Vbupgrade.exe is not on the path by default. You will need to either add it to the path or switch to the directory where it is installed before running it from the command line. The default installation location is C:\Program Files\Microsoft Visual Studio .NET 2003\Vb7\VBUpgrade.

To upgrade a project from the command line

1. Open your project in Visual Basic 6.0. On the **File** menu, choose **Make *projectname***. Run the application and make sure that there are no errors.
2. At a command prompt, type **Vbupgrade *projectname*** (where *projectname* is the full path for the Visual Basic 6.0 application) plus any additional command-line arguments.

   > **Note** For information on Vbupgrade command-line syntax and options, see Vbupgrade Command-Line Syntax.

3. Open the resulting project in Visual Studio .NET. Review the code — notes are inserted into the code during upgrade to alert you to any potential problems. Any errors must be fixed before you can run the application. Any warnings should be examined to make sure that they do not affect application behavior.

   > **Tip** For an explanation of any error or warning, click on the description to display a Help topic.

4. On the **Debug** menu, choose **Start.** Run the application and make sure that there are no errors and that the behavior is the same as it was in Visual Basic 6.0.

   > **Note** The Vbupgrade tool cannot catch all possible incompatibilities; it is possible that you will encounter additional errors when you run the application. For a list of known issues, see Troubleshooting Applications Upgraded from Visual Basic 6.0.

See Also

Vbupgrade Command-Line Syntax | Upgrading Applications Created in Previous Versions of Visual Basic | Viewing the Upgrade Report | Preparing a Visual Basic 6.0 Application for Upgrading

Vbupgrade Command-Line Syntax

The Vbupgrade.exe command-line tool can be used to upgrade a Visual Basic 6.0 application to Visual Basic .NET.

The command line syntax for Vbupgrade.exe is:

```
Vbupgrade[.exe] inputname [/Out outputname] [/Verbose [/NoLog [/LogFile logfilename]]]
Vbupgrade[.exe] /?
```

inputname
Required.

Specifies the path and file name of the Visual Basic 6.0 project to be upgraded.

/Out *outputname*
Specifies the path for the folder where the Visual Basic .NET project will be created. The default path is \OutDir.

/Verbose
Displays all output to the DOS window during upgrade.

/NoLog
Does not create a log file during upgrade.

/LogFile *logfilename*
Specifies the path and file name for the log file created during upgrade. If the path and file name are not specified, a log file will be created in the same folder as the Visual Basic .NET project. The default file name is *ProjectFileName*.log, where *ProjectFileName* is the name of the project file.

/?
Displays a list of command-line options.

Remarks

Paths or file names that contain spaces must be surrounded by quotation marks.

See Also

Upgrading a Project from the Command Line | Upgrade Wizard

Troubleshooting Applications Upgraded from Visual Basic 6.0

Although the upgrading tools in Visual Basic .NET were carefully designed to detect and report any issues with upgraded applications, there are some cases that cannot possibly be detected. This topic lists some known issues that are not detected by the upgrading tools and explains how to address them.

> **Note** Additional troubleshooting issues are addressed in a Knowledge Base article that will be updated as new issues are discovered. The article number is Q295626, available at Microsoft Product Support Services (http://support.microsoft.com/directory/content.asp?ID=FH;EN-US;vbdnmig).

Cannot Access Help Links After Upgrading Visual Studio .NET

When working with an application upgraded with an earlier version of Visual Studio .NET, the Help links inserted by the upgrade tool may cause a "Page not found" error. This occurs because the links reference the Help collection for the earlier version, and that Help collection was uninstalled.

To fix this problem, use the Find and Replace command on the Edit menu to search for the string "MS.VSCC" and replace it with "MS.VSCC.<VersionNumber>" (for example, "MS.VSCC.2003" for Visual Studio .NET 2003) in all of the Help links.

Behavioral Difference for Fixed-length Strings in User-defined Types

In Visual Basic 6.0, strings assigned to a fixed-length string in a user-defined type were automatically truncated if they were longer than the fixed length. After upgrading to Visual Basic .NET, strings will no longer be truncated, possibly resulting in incorrect results.

> **Note** During upgrade, the **VBFixedString** attribute is added to fixed-length strings in user-defined types. This attribute allows the file functions in the Visual Basic Compatibility library to treat them as fixed-length strings.

To fix this problem, find any code that assigns a string to the fixed-length string and add code to check the length of the string and truncate it if necessary:

```
' Before
MyString = "1234567"
MyStruct.FixedString5 = MyString

' After
MyString = "1234567"
If Len(MyString) > 5 Then
   MyString = Microsoft.VisualBasic.Left(MyString, 5)
End If
MyStruct.FixedString5 = MyString
```

Closing a Form Calls Dispose

In Visual Basic 6.0, you could unload a form and later reload it by calling the **Show** method. In Visual Basic .NET, the **Close** method for a form calls the **Dispose** method so that it is automatically garbage-collected. This can cause subtle behavioral differences that may be hard to detect.

- In Visual Basic .NET, if you call the **Show** method for a form that has been unloaded, you will get a new instance of the form; any property settings from the base class will be lost.
- For forms that are shown modally, **Dispose** is not called automatically. In some cases, you may want to call **Dispose** in order to clean up resources.

Late-bound Calls to COM Objects May Cause Type Mismatch Errors

In Visual Basic 6.0, when a late-bound COM object was passed as a parameter to a late-bound call, the object was coerced to a **Variant** of type **Nothing**. When upgraded to Visual Basic .NET, COM objects declared as type **Object** are treated the same as **Variants** (which are always converted to type **Object** during upgrade); these objects are marshaled to the variant type **Empty**. This causes a type mismatch error in Visual Basic .NET.

To fix this problem, make sure that all objects are early bound.

Values Returned by Err.Number May Be Different

In some cases, errors returned by Visual Basic .NET may be different than those returned by Visual Basic 6.0. For error handling code that relied on the values returned by **Err.Number**, this might cause different behavior in your application.

The following code shows an example of this:

```
' Visual Basic 6.0
On Local Error GoTo Result
Dim x() As Boolean
Dim y As Variant

y = x(10)

Result:
If Err.Number = 9 Then
   ' Do something.
Else
   ' Do something else.
End If
```

Before upgrading, **Err.Number** will always return 9 (Subscript out of range) and execute the first part of the **If** statement. After upgrading, it will return 91 (Object variable or With block not set) and execute the **Else** clause. This is because in Visual Basic .NET, an array must be initialized before it can be referenced; in Visual Basic 6.0, arrays were initialized when declared.

If you depend on the return values from **Err.Number** in your code, you should carefully test the results and modify your code as necessary.

Dispose Must Be Explicitly Called for a DataEnvironment

In Visual Basic 6.0, any open recordsets and connections were closed when a **DataEnvironment** was closed. When upgraded to Visual Basic .NET, the DataEnvironment becomes a Public member of the **DataEnvironment** class; recordsets and connections are not automatically closed.

You will need to add code to explicitly close the recordsets and connections by calling the **DataEnvironment.Dispose** method. For Windows Forms applications you can call the **Dispose** method from the startup form's **Dispose** method. For class libraries, you will need to implement a **Dispose** method on your class library then have the client application call **Dispose** before it releases it's reference to the class.

See Also

Upgrading Applications Created in Previous Versions of Visual Basic | Working with Both Visual Basic 6.0 and Visual Basic .NET

Working with Both Visual Basic 6.0 and Visual Basic .NET

Both Visual Basic .NET and Visual Basic 6.0 can be installed on the same computer and run at the same time. Likewise, applications written in Visual Basic .NET and Visual Basic 6.0 can be installed and run on the same computer. Components written in Visual Basic .NET can interoperate with COM components written in earlier versions of Visual Basic and other languages. For example, you can drag an ActiveX control written in Visual Basic 6.0 onto a Visual Basic .NET Windows Form, use a Visual Basic 6.0 COM object from a Visual Basic .NET class library, or add a reference to a Visual Basic .NET library to a Visual Basic 6.0 executable.

Components compiled with Visual Basic .NET have subtle run-time differences from components compiled with Visual Basic 6.0. Because Visual Basic .NET objects are released through garbage collection, when objects are explicitly destroyed there may be a lag before they are actually removed from memory. There are additional differences such as data type and other language changes. The combined result of these differences is that Visual Basic .NET applications will have similar but not identical run-time behavior to Visual Basic 6.0 applications.

In addition, Visual Basic .NET makes binary compatibility between Visual Basic .NET components and those in Visual Basic 6.0 unnecessary. Components now have a more robust versioning and deployment system, files can be deployed by simply copying to a directory, and upgrading to a new version of a component is as simple as replacing the old file with a new file. All you have to do is ensure classes and methods are compatible with the previous version.

See Also

Why Upgrading is Necessary | Preparing a Visual Basic 6.0 Application for Upgrading |
Upgrading Applications Created in Previous Versions of Visual Basic

Walkthrough: Upgrading a Visual Basic 6.0 Application to Visual Basic .NET

In this walkthrough, you will upgrade a Visual Basic graphics sample application to Visual Basic .NET. In addition to illustrating the upgrading process, this walkthrough highlights the differences in graphics architectures between Visual Basic 6.0 and Visual Basic .NET. This should also serve as an introduction to dealing with architectural issues in your applications.

The sample application in this case is a simple form that contains two buttons, a picture box, a timer, and a hidden image control. Although not typical of most applications that you will want to upgrade, it does present several challenging issues that you may face when upgrading your own applications.

> **Note** This walkthrough requires that both Visual Basic 6.0 and the Drawing.vbp sample application be installed on your development computer. The sample application can be installed from the Visual Studio .NET CD; the default location is Program Files\Microsoft Visual Studio .NET 2003\Vb7\VB Samples\MigrationWalkthrough.

To familiarize yourself with the application

1. In Visual Basic 6.0, open the **Drawing.vbp** project file (on the **File** menu, choose **Open Project**).
2. On the **Run** menu, choose **Start**. Click the buttons, and observe the run-time behavior and appearance of the application.
3. On the **Run** menu, choose **End** to close the application.
4. In the Project Explorer, select **Form1(PicForm.frm)**. On the **View** menu, choose **Object**. Observe the design-time layout of the form.
5. On the **View** menu, choose **Code**. Familiarize yourself with the application's code.

   > **Tip** If you do not understand how the code works, try stepping through the code by choosing **Step Into** on the **Debug** menu.

6. On the **File** menu, choose **Exit** to close the application. If prompted to save changes, choose **No**.

To run the Upgrade wizard

1. Open Visual Studio .NET. On the **File** menu, choose **Open**, **Project**.
2. In the **Open Project** dialog box, locate the **Drawing.vbp** file and open it.

   This will launch the Visual Basic Upgrade Wizard. The first page of the wizard contains an explanation of what the wizard will do.

3. Click **Next** to go to the second page of the wizard. This page presents options for upgrading; in this case, there are no applicable options available.
4. Click **Next** to go to the third page of the wizard. Click **Next** to accept the default location for the new project, or enter a different location. The default location is a new folder immediately beneath the Visual Basic 6.0 project's folder.

   If prompted to create a new folder, choose **Yes**.

5. On the fourth page of the Wizard, click **Next** to begin the upgrade.

   When the upgrade is finished, the wizard will close and the new project will appear in Solution Explorer.

To view the upgrade results

1. In Solution Explorer, select **_UpgradeReport.htm** and double-click to open the Upgrade Report.

   Note that the report shows no Global Issues that need to be addressed; for PicForm.vb it shows six errors and no warnings.

   > **Note** If there were warnings, they would also appear in the Task List window as UPGRADE_WARNING items. Warnings are for code that might cause subtle differences in the run-time behavior of your application. Double-clicking an UPGRADE_WARNING in the Task List window will take you directly to the code that may need to be modified.

2. Expand the **PicForm.vb** section by clicking on the plus sign in the **New Filename** column. This displays a detailed list of upgrade issues for the form.

   Note that the issues apply to three different locations: the **ClearPictureBox_Click** procedure, the **Timer1_Timer** procedure, and the form layout. You can click on the **Description** link to display a Help topic explaining each issue. The layout issues in

this case do not require any action; checking the Help reveals that they are for two properties of the Visual Basic 6.0 **PictureBox** control that do not map to equivalent Visual Basic .NET properties

To fix the ClearPictureBox_Click error

1. In Solution Explorer, select **PicForm.vb**. On the **View** menu, choose **Code**.
2. In the Code Editor, select the **ClearPictureBox_Click** procedure.

   Tip There is an UPGRADE_ISSUE comment added to the procedure. If you scroll to the end of the comment, you can click on the link that displays the associated Help topic. All issues (except for layout issues) are added as code comments during upgrade, including UPGRADE_NOTE comments explaining what happened during upgrade; these do not appear in the Upgrade Report.

3. Add the following code to the **ClearPictureBox_Click** procedure:

```
Dim g as Graphics = Picture1.CreateGraphics()
g.Clear(Picture1.BackColor)
g.Dispose()
```

   The only error in the original procedure is the **Picture1.Cls()** method call. In Visual Basic .NET the intrinsic controls do not provide direct access to their drawing surfaces with methods (as in Visual Basic 6.0). All graphics operations are handled with a special object of type **Graphics**. You can access a control's drawing surface by calling the **CreateGraphics()** method to get an instance of a **Graphics** object.

4. Delete or comment out the original line `Picture1.Cls()`.

To fix the Timer1_Timer errors

1. In the Code Editor, select the **Timer1_Tick** procedure.

   Note The **Timer** event from the Visual Basic 6.0 **Timer** control is upgraded to the **Tick** event of the Visual Basic .NET **Timer** component. The Upgrade Report shows the old event name; the Code Editor uses the new event name.

2. At the start of the function add the following code:

```
Dim g as Graphics = Picture1.CreateGraphics()
```

   As with the previous procedure, the errors here begin with the lack of a **Graphics** object.

3. Find the first UPGRADE_ISSUE, the **Picture1.Print** method. Add the following code:

```
g.DrawString("VB .NET ROCKS!",  Me.Font, new SolidBrush(Color.Yellow), 0, 0 )
```

   The **DrawString** method of the **Graphics** object replaces the Visual Basic 6.0 **Print** method. Where the **Print** method took a single **Text** argument, **DrawString** also takes arguments specifying the **Font**, a **Brush** object that specifies a **Color** object, and the starting coordinates for drawing the text.

4. Delete or comment out the original line `Picture1.Print("Visual Basic ROCKS!")`.
5. Find the next UPGRADE_ISSUE, **Picture1.Cls()**. Replace the code with the following:

```
g.Clear(Picture1.BackColor)
```

6. Find the last UPGRADE_ISSUE, **Picture1.PaintPicture**. Add the following code:

```
g.DrawImage(sourceImage.Image, CSng(LeftPos * Picture1.Size.Width), 0)
```

   In this case, the **DrawImage** method of the **Graphics** class replaces the **PaintPicture** method.

7. Delete or comment out the original line `Picture1.PaintPicture(sourceImage, LeftPos * VB6.PixelsToTwipsX(Picture1.Width), 0)`.
8. At the end of the procedure, add the following code:

```
g.Dispose()
```

The **Dispose** method is necessary to free up the memory resources associated with the **Graphics** object.

To test the application

1. On the **Debug** menu, choose **Start**.

   Note The first time you run the application, you will be prompted to save the Solution file. Click **Save** to save the file and run the application.

2. Click the **ShowImage** button.

   You will notice that the image flickers when drawing. This is due to the entire area being cleared before each updated image is drawn on the screen.

The solution here is to only draw to the necessary parts of the screen, instead of the entire area represented by the **Graphics** object. You will want to make sure that when the image is drawn on the screen, any pixels remaining on the screen from the last call to **DrawImage()** are obscured with the background color. Since the image is moving from right to left, you only have to obscure the additional pixels to the right of the picture.

To fix the screen flicker problem

1. In the **Timer1_Tick** procedure, replace the `g.Clear(Picture1.BackColor)` method with the following code:

```
Dim imageWidth As Integer = sourceImage.Image.Width
Dim imageHeight As Integer = sourceImage.Image.Height
Dim left As Integer = imageWidth + (Picture1.Size.Width * LeftPos)

g.FillRectangle(New SolidBrush(Me.BackColor), New Rectangle(left, 0, 6, imageHeight))
```

   In place of the **Clear** method, you use the **FillRectangle** method, using the background color to obscure any remaining pixels to the right of the current image being drawn. To do this, you first need to calculate the left-most coordinate of the rectangle, which is accomplished by the first three lines of code.

2. On the **Debug** menu, choose **Start**.
3. Click the **ShowImage** button.

   You will notice that the image no longer flickers as it moves.

There is still one additional problem that should be fixed. You may have noticed that the message displayed in the **PictureBox** control is in a smaller font than the original version. This is because the **Font** property of the **PictureBox** control in the Visual Basic 6.0 application was set to Arial Bold 16 at design time. There is no **Font** property for the Visual Basic .NET **PictureBox** control, so instead the form's default font (`Me.Font`) is used. You can remedy this by declaring a new **Font** object.

To fix the font

1. In the **Timer1_Tick** procedure, add the following declaration:

```
Dim f As System.Drawing.Font = New System.Drawing.Font("Arial", _
16, FontStyle.Bold)
```

2. In the call to `g.DrawString`, change the font parameter from `Me.Font` to `f`.
3. On the **Debug** menu, choose **Start**.
4. Click the **ShowImage** button.

   The text is now displayed in the correct font.

Now that the application is back to parity with the original Visual Basic 6.0 application, you will add some code to enhance the application. After all, there is no point in upgrading the application if you do not want to take advantage of Visual Basic .NET.

In the next step, you will add functionality to make the form fade out slowly when it is closed.

Note The fade effect is not visible on 256-color displays; you will need to set your color depth to High Color or True

Color to view the results. The **Opacity** property only works on Windows XP.

To enhance the application

1. In the Code Editor, select the **Class Name** drop-down list and choose **(Base Class Events)**.
2. In the **Method Name** drop-down box, select the **Closing** event.
3. In the **Form1_Closing** procedure, add the following code:

```
Dim i as Single
For i = 1 to 0 Step -0.1
   Me.Opacity = i
   Application.DoEvents()
   System.Threading.Thread.Sleep(100)
Next
```

4. On the **Debug** menu, choose **Start**.
5. Close the form and notice how it fades away.

   You can control how fast it fades by increasing the size of the step (or by reducing the Sleep delay).

See Also

Upgrading from Visual Basic 6.0 | Introduction to Windows Forms | Graphics Changes in Visual Basic .NET

Upgrading Visual InterDev 6.0 Applications to Visual Studio .NET

If you have existing Visual InterDev projects, you might want to upgrade them to Visual Studio .NET. Although you cannot open a Visual InterDev project in Visual Studio .NET, there are features in Visual Studio .NET that will allow you to make a smooth transition from Visual InterDev.

> **Note** Upgrading Visual Interdev 6.0 applications to Visual Studio .NET 2003 is not supported.

There are two possible ways to work with an existing Visual InterDev application in Visual Studio .NET:

- **Creating a new Web project** to interact with an existing Visual InterDev application. This will create all of the extra files needed by your new Web application, in addition to creating the new project. You can then move files into your new Web application as you upgrade them. This approach is recommended when you want to have all the associated files created for you at the time you create the new Web project.

  > **Note** This is the recommended path for upgrading a project that is under source control with Visual SourceSafe. This will effectively fork your history, and you will be unable to access previous versions of your Visual InterDev project as part of your Visual Studio .NET project.

- **Creating a new project in your existing folder**. One way to begin your project transition is by using the New Project in Existing Folder template. That template allows you to create a new Visual Studio .NET project in an existing folder. From there you can add your existing Visual InterDev files to your new project and upgrade them as needed. This approach creates only a new project, and does not create any of the associated files you will need to run your new Web application, such as the Web.config file. Therefore, it is recommended that you use this approach when you want to manually upgrade any configuration data to your new Web application. Do not use this method to upgrade a project that is under source control with Visual SourceSafe.

  > **Note** The New Project in Existing Folder template is not available in the Standard Edition of Visual Basic and Visual C# .NET. For more information, see Visual Basic Standard Edition Features or Visual C# Standard Edition Features.

HTML files can be transferred from Visual InterDev projects to Visual Studio .NET projects without any loss of functionality. ASP pages will also continue to function, although it is recommended that you upgrade your ASP pages to ASP.NET in order to take advantage of the rich new array of features that the .NET Framework provides. Design-time controls (DTCs) are no longer supported, and new solutions will have to be developed to accommodate any loss of functionality. There is currently no tool for migrating ASP to ASP.NET, so you will have to upgrade your Web projects manually. For more information on migrating ASP pages to ASP.NET, see Migrating ASP Pages to Web Forms Pages. For more information on creating Web Applications, see Creating Web Applications and Services.

To upgrade your project to Visual Studio .NET

1. Open Visual Studio .NET.
2. From the **File** menu, choose **New** and then choose **Project**.

   The **New Project** dialog box appears.

3. Select either **Visual Basic Projects** or **Visual C# projects**, depending upon which language you want to use. For more information, see Programming Languages.
4. Enter the name for the new project in the **Name** box.
5. Decide whether you need to manually create upgrade-configuration data.

   - If you want to automatically create new configuration data, select **ASP.NET Web Application.**

     - or -

   - If you want to manually upgrade configuration data, double-click the **New Projects In Existing Folder** project template.

     > **Note** You must specify the HTTP path to the existing project. If you specify the file path, a Windows project will be created instead of a Web project.Al

     > **Caution** Do not use this method to upgrade a project that is under source control with Visual

SourceSafe.

6. Browse to the folder that contains your **Visual InterDev** project and select it.
7. In Solution Explorer, right-click the new project and select **Add** from the shortcut menu, then **Add Existing Item**.

   The **Add Existing Item** window opens.

8. Click the drop-down list on the **Files Of Type** box and select **All Files**.

   All files in the folder will be shown.

9. Select the files from your Visual InterDev project that you want to add to your Visual Studio .NET project.

See Also

Introduction to ASP.NET Web Applications in Visual Studio | Creating ASP.NET Web Applications | Migrating to Web Forms | Migrating ASP Pages to Web Forms Pages | Creating Web Applications and Services

Managing Projects

The project is the starting point for authoring applications, components, and services in Visual Studio .NET. It acts as a container that manages your source code, data connections, and references. A project is organized as part of a solution, which can contain multiple projects that are interdependent of each other. The topics in this section describe the various tasks involved in creating and managing Visual Basic and Visual C# projects.

In This Section

Local vs. Web Projects
Explains the essential differences between these two project models.
File Properties
Describes property settings that indicate what actions the project system should perform on the files.
Visual Basic and Visual C# Projects
Provides a list of the types of projects that can be created with project templates.
Referencing Namespaces and Components
Provides links to topics about adding, removing, fixing, and managing project references.
Working with Web Projects
Provides links to topics about tasks involved in working with Web projects.
Resource File Naming Conventions
Provides links to information about XML and non-XML assembly-manifest resource files.
Configuring Applications Using Dynamic Properties
Describes how to configure your projects so that some project properties are stored to a configuration file instead of inside the compiled code.

Related Sections

File Types and File Extensions in Visual Basic and Visual C#
Lists and describes the file extensions for all files available in Visual Basic and Visual C# projects.
Project Changes in Visual Basic .NET
Summarizes how projects are different than they were in Visual Basic 6.0.
Visual Basic Code Example Topics
Lists quickstart-like code examples that help you perform common tasks in Visual Basic .NET.
Decision Chart
Provides a graphical, linked guide to decisions about designing your application.
Debugging
Describes how to resolve logic and semantic errors by using the Visual Studio Debugger.
Tracing and Instrumenting Applications in Visual Basic and Visual C#
Describes how to monitor the execution of your application while it is running.
Build Configurations
Describes options for building and deploying solutions and projects.
Default and Custom Builds
Gives an overview of creating standard and custom builds from within the IDE.
Devenv Command Line Switches
Provides the standard command line switches for Visual Studio.
Visual Basic Compiler Options
Provides compiler command line options for Visual Basic.
C# Compiler Options
Provides compiler options for Visual C#.
Refreshing Items in Solution Explorer
Describes how to incorporate changes made by other users into your project.
What's New in Projects
Lists and briefly describes the major new Visual Basic project features, including deployment projects.
Adding New Project Items
Provides step-by-step instructions on adding new project items.
Introduction to Solutions, Projects, and Items
Gives an overview of solutions and projects.
Deployment Projects
Recommends which kind of deployment project to use for your application or component.
Removing, Deleting, and Excluding Items
Provides instructions for how to remove, delete, or exclude items in Visual Studio.

Setting Visual Basic Project Properties

Provides step-by-step instructions on how to access project properties.

Setting Visual C# Project Properties

Provides descriptions of Visual C# project property settings.

Programming with Office

Provides information on using Microsoft Office and Visual Studio .NET as part of your business application.

Working with Multiple Versions of the .NET Framework

By default, managed applications created with Visual Studio .NET 2003 or later require version 1.1 of the common language runtime in order to run. There may be some cases where you want your application to target an earlier version of the runtime. For example, if all of your customers have already installed runtime version 1.0 and deploying version 1.1 is not feasible, you may want to target version 1.0.

> **Note** Although it is possible to target version 1.0 of the common language runtime from Visual Studio .NET 2003, it is highly recommended that you use Visual Studio .NET 2002 for that instead. You can install and run both versions side-by-side on the same computer. For more information, see Installation and Setup.

> **Caution** Targeting a previous version of the runtime could cause the application to fail if it references .NET Framework classes or members that did not exist in the previous version. You should thoroughly test your application against all specified versions to make sure that it runs properly. For more information on the differences between version 1.0 and 1.1, see What's New in the .NET Framework Version 1.1.

Visual Studio .NET 2003 projects use the **Supported Runtimes** property to specify the required versions of the common language runtime. This property is only available for Windows Application, Windows Service, Console Application, and ASP.NET Web Application projects.

The **Supported Runtimes** property can be set in the **Project Property Pages** dialog box. For more information, see Specifying the Runtime Version for An Application.

> **Note** When you specify version 1.0, the app.config file will be modified with a section specifying that it can run on version 1.0 of the runtime. The application will actually be built using the version 1.1 compiler, and will be a version 1.1 application in all other respects.

In the event that you specify support for than one runtime version, when the application runs on a computer with more than one runtime version installed, the first available version listed in the .config file will be loaded.

Windows Application, Windows Service, or Console Application Projects

For these project types, selecting a runtime causes the project system to update the app.config file in your project, or create an app.config file if one does not already exist. At run time, the name of the app.config file must be *filename.ext*.config and the file must be in the same directory as the executable. The project system will create the *filename.ext*.config file from the app.config file and place it in the bin*target* directory.

ASP.NET Web Application Projects

For ASP.NET Web Application projects, modifying this property causes the project system to update the web.config file in your project by adding assembly binding (<bindingRedirect> and <assemblyIdentity>) tags in a <dependentAssembly> section. Web pages in Visual Studio-built ASP.NET web applications are precompiled to a .dll with a dependency on the runtime associated with the compiler.

The web.config file remains in the project directory and is not renamed.

For more information, see ASP.NET Side-by-Side Overview

See Also

Specifying the Runtime Version for An Application | Side-by-Side Execution | ASP.NET Side-by-Side Overview

Projects and Backward Compatibility

When a project created in Visual Studio .NET 2002 is opened in Visual Studio .NET 2003 or later, you are prompted with a warning that the project file will be updated. After you open the project, you will no longer be able to open it in the earlier version.

The versions of Visual Studio .NET rely on different versions of the common language runtime — for example, 1.0 for Visual Studio .NET 2002 and 1.1 for Visual Studio .NET 2003. After an application has been built against version 1.1, it requires that version 1.1 be present on a target computer in order to run.

Applications built using version 1.0 will continue to run with either runtime version.

> **Note** Using Visual Studio .NET 2003 or later, it is still possible to create applications that target runtime version 1.0 using the **Supported Runtimes** property. For more information, see Working With Multiple Versions of the .NET Framework.

See Also

Working With Multiple Versions of the .NET Framework | Side-by-Side Execution

Specifying the Runtime Version for an Application

Applications created in Visual Studio .NET 2003 or later can be set to target one or more versions of the common language runtime by setting the **Supported Runtimes** property. By default, the latest version of the common language runtime is used; specify a different runtime version if you want your application to run on a previous version of the common language runtime.

> **Caution** Selecting a previous version of the runtime could cause the application to fail if it references .NET Framework classes that did not exist in the previous version. You should thoroughly test your application against all specified versions. For more information, see Working With Multiple Versions of the .NET Framework.

To specify the runtime version in Visual Basic .NET

1. Select the project in Solution Explorer.
2. On the **Project** menu, choose **Properties**.
3. In the **Property Pages** dialog box, select the **Build** node and click the **Change** button.
4. In the **.NET Framework Version** dialog box, select a version option and click **Update**.

To specify the runtime version in Visual C# .NET

1. Select the project in the Solution Explorer. On the **Project** menu, choose **Properties**.
2. In the **Property Pages** dialog box, select the **Supported Runtimes** property and click the Ellipsis (...) button.
3. In the **.NET Framework Version** dialog box, select a version option and click **Update**.

See Also

Working With Multiple Versions of the .NET Framework | Build, Common Properties, <Projectname> Property Pages Dialog Box | Side-by-Side Execution

Local vs. Web Projects

There are several factors to consider when you decide whether to use a local or Web project model for your application.

First, installation strategy should play a part in your overall decision. If you need a quick-deploying application on a server with minimal client configuration, the Web project model is best. You can install and run this project on the server without installing support files on each client, and make upgrades only on the server. However, for projects that have little server involvement or for stand-alone applications, a local project model is the better choice. In this scenario, upgrades can be completed on each individual client computer.

Another factor is the division of workload. When analyzing your application's framework, you need to consider where the bulk of the work is going to be done. Web projects offer a server-based strategy where the majority of operations are conducted on the server. The client computer uses a Web browser to send information to and display information from the server-side application. If your server resources are limited, then consider using a local project model for your application. The work can be done on the client computer without consuming precious server time.

The following table gives a side-by-side comparison of Web projects and local projects in both the design and working environments:

Consideration	**Web project**	**Local project**
File storage during development	Files are stored at a specified URL on a computer with Internet Information Services (IIS) installed. The project also keeps a temporary cache of files on client computers for working offline.	Project files are stored on the local computer.
Build model when developing	Application code is compiled as a DLL and run on the server. Results are seen on the client computer.	Code is compiled and run on the client computer.
Deployment method	The project is deployed to a Web server.	Project files are installed on the client computer with required support files.
Workload	Client computers run the application remotely with a Web browser. Processing is done primarily on the server.	The application is run locally on the client computer.
Upgrading	Application changes are made at the server level without installation of support files on client computers.	The application upgrades are installed on client computers.
Moving project files	The Copy Project command is used to move files to the specified URL.	Project files are physically copied from one location to another.

See Also

Creating Web Projects | Opening Web Projects | Managing Project Web References | Changing Web Project Access Methods | Visual Basic and Visual C# Projects | Taking Web Projects Offline

File Properties

You can use file properties to indicate what actions the project system should perform on the files. For example, you can set file properties to indicate whether a file should be compiled or embedded into the build output as a resource.

You can select any file in Solution Explorer and then examine its properties in the Properties window. Visual Basic and Visual C# files have four properties: **FileName**, **BuildAction**, **CustomTool**, and **CustomToolNamespace**.

> **Note** The **BuildAction**, **CustomTool**, and **CustomToolNamespace** properties are provided for advanced scenarios. The default values are typically sufficient and do not need to be changed.

FileName Property

You can rename a file by clicking the FileName property in the Properties window and typing in the new name. Note that if you change the file's name, Visual Studio .NET will automatically rename any .vb or .resx files that are associated with it.

BuildAction Property

The BuildAction property indicates what Visual Studio .NET does with a file when a build is executed. **BuildAction** can have one of several values:

- **None** - The file is not included in the project output group and is not compiled in the build process. An example is a text file that contains documentation, such as a Readme file.
- **Compile** - The file is compiled into the build output. This setting is used for code files.
- **Content** - The file is not compiled, but is included in the Content output group. For example, this setting is the default value for an .htm or other kind of Web file.
- **Embedded Resource** - This file is embedded in the main project build output as a DLL or executable. It is typically used for resource files.

The default value for **BuildAction** depends on the extension of the file you add to the solution. For example, if you add a Visual Basic project to Solution Explorer, the default value for **BuildAction** is **Compile**, because the extension .vb indicates a code file that can be compiled. File names and extensions appear in Solution Explorer.

Note that the name of the file in your project will not be the identifier for the managed resource in the assembly manifest (see Assembly Manifest for more information). The identifier will be *namespace.filename.extension*, where *namespace* is the value of the **DefaultNamespace** property in a Visual C# project or **RootNamespace** property in a Visual Basic .NET project. *Filename* and *extension* remain the same as their original designation. If the file is a .resx file, the project system will run resgen.exe on the file, creating a .resource file. The .resource file will be embedded in the assembly, so the assembly manifest will refer to the .resources file and not the .resx file.

For example, if you add the file MyFile.bmp to a project whose default namespace is MyProj, and set the build action to **Embedded Resource**, MyProj.MyFile.bmp will be the identifier in the assembly manifest. If you then add the file MyFile.resx to the project, the default build action will be **Embedded Resource** and, MyProj.MyFile.resources will be the identifier in the assembly manifest.

For more information on how to access resource files (compiled from .resx files) at run time, see ResourceManager Class. For more information on how to access all other embedded files and resources at run time, see Assembly.GetManifestResourceStream Method.

CustomTool Property

Custom tools are components that can be used to transform files from one type to another at design time. For example, a custom tool might be a dataset code generator that reads in an XML Schema (.xsd) file, and generates classes in a code file that programmatically exposes its tables and columns. There is a predefined list of custom tools available in the product; this property allows you to see which custom tool is applied to a file. In rare circumstances, you might need to change the value of this property. The value of this property must either be blank or one of the built-in custom tools.

To set or change the custom tool, click the CustomTool property in the Properties window and type in the name of a custom tool.

CustomToolNamespace Property

If you have a custom tool assigned to your project, the CustomToolNamespace property allows you to specify the namespace you want to assign to code generated by the custom tool. When you specify a value for the **CustomToolNamespace** property, code generated by the tool is placed in the specified namespace. If the property is empty, generated code is placed in the default

namespace for the folder in which the converted file lives; for Visual Basic, this is the project's root namespace, and for Visual C# this corresponds to the setting of the DefaultNamespace property for the folder.

See Also

Managing Projects | Visual Basic and Visual C# Projects | File Types and File Extensions in Visual Basic and Visual C#

Setting Visual Basic Project Properties

You specify global settings for your project by using project properties. These default settings control the appearance and behavior of the items in your project throughout its life cycle. Properties are saved in the project's .vbproj file, and can be accessed through the **Property Pages** dialog box.

To access project properties

1. With a project selected in Solution Explorer, click **Property Pages** on the **View** menu.

   Notice the two folders in the project's **Property Pages** dialog box: **Common Properties** and **Configuration Properties**. Only the properties in **Configuration Properties** can be set by project configuration.

2. Select the **Configuration Properties** folder. Using the **Configuration** drop-down list, select the configuration for which you want to set this project's properties.

   If you have the **Common Properties** folder selected, the **Configuration** drop-down list is unavailable.

3. In the left pane, select a folder and property page.
4. In the right pane, set the available properties.

   > **Note** Properties that are dimmed are defined when the project is created; they are displayed for information only.

Getting Help on Properties

Documentation is available for each property page by simply clicking the **Help** button.

For a list of property pages available in a Visual Basic project, see <Projectname> Property Pages Dialog Box.

See Also

<Projectname> Property Pages Dialog Box

Setting Visual C# Project Properties

A project's properties specify how a project is built and debugged.

Accessing Project Properties

Visual C# property pages are only available when the project is selected in Solution Explorer; property pages are not available for individual files for a Visual C# project. However, you can access a file's properties in the **Properties** window.

To access project properties

1. With a project selected in Solution Explorer, click **Property Pages** on the **View** menu.

   Notice the two folders in a Visual C# project's **Property Pages** dialog box: **Common Properties** and **Configuration Properties**. Only the properties in **Configuration Properties** can be set by project configuration.

2. Using the **Configuration** drop-down list box and with the **Configuration Properties** folder selected, select the configuration for which you want to set this project's properties. If you have the **Common Properties** folder selected, the **Configuration** drop-down list box is unavailable.
3. In the left pane, select a folder and property page.
4. In the right pane, set the available properties.
5. Click **OK** to save the settings. If you want a change to take effect immediately, click **Apply**.

> **Note** Properties that are grayed are defined when the project is created; they are displayed for information only.

Properties are saved in the project's .csproj file.

For information on the **Build** menu in the Visual Studio environment, see Default and Custom Builds.

Getting Help on Properties

Documentation is available for each property; simply click the **Help** button.

- General, Common Properties, <Projectname> Property Pages Dialog Box
- Web Settings, Common Properties, <Projectname> Property Pages Dialog Box
- Designer Defaults, Common Properties, <Projectname> Property Pages Dialog Box
- References Path, Common Properties, <Projectname> Property Pages Dialog Box
- Build Events, Common Properties, <Projectname> Property Pages Dialog Box
- Build, Configuration Properties, <Projectname> Property Pages Dialog Box
- Debugging, Configuration Properties, <Projectname> Property Pages Dialog Box
- Advanced, Configuration Properties, <Projectname> Property Pages Dialog Box
- Device, Common Properties, <Projectname> Property Pages Dialog Box

See Also

Project Properties | Properties Window | Visual Basic and Visual C# Projects | C# Compiler Options

File Types and File Extensions in Visual Basic and Visual C#

Project items refer to the types of files you add to a project. The project items available to you differ depending on whether you are working in a local project or a Web project. For more information, see Local vs. Web Projects and Creating Web Projects.

Adding Project Items

The Add New Item dialog box is accessed through the Project menu. For more information about adding items to your project see Adding New Project Items.

The following tables list the available project items for local and Web projects. Note that for some project items, the file extension differs depending on whether the item occurs in a Visual Basic project (indicated by .vb) or in a Visual C# project (indicated by .cs). For other project items, the file extension does not distinguish between a Visual Basic project and a Visual C# project; that is, there is only one possible file extension for the project item.

Local Project Items

Project Item	File Extension	Purpose of the Project Item
Windows Form	.vb or .cs	A form for creating Windows applications. For more information, see Creating Windows Forms.
Class	.vb or .cs	A code file that contains a class declaration. For more information, see Classes.
Module (Visual Basic only)	.vb	A file for storing groups of functions.
Component Class	.vb or .cs	A class for creating components using the visual designer. For more information, see Component Classes.
User Control	.vb or .cs	A class for creating a Windows Form control using the visual designer.
Data Form Wizard	.vb or .cs (.aspx for Web projects.)	A data form for Windows applications. For more information, see Data Form Wizard.
DataSet	.xsd	A file for creating an XML schema with **DataSet** classes. For more information, see Visual Studio Tools for Creating Datasets.
XML File	.xml	A blank XML file. For more information, see XML Designer.
XML Schema	.xsd	A file for creating a schema for XML documents. For more information, see Introduction to XML Schemas.
Code File	.vb or .cs	A blank code file.
Custom Control	.vb or .cs	A class for creating a user-drawn Windows Form control. For more information, see Authoring Controls for Windows Forms.
HTML Page	.htm	An HTML page that can include client-side code. For more information, see Design View, HTML Designer.
Inherited Form	.vb or .cs	A new form based on an existing Windows Form. For more information, see Windows Forms Inheritance.
Web Custom Control	.vb or .cs	A class for creating an ASP.NET server control. For more information, see ASP.NET Server Controls.
Inherited User Control	.vb or .cs	A new control based on an existing Windows Form control. For more information, see Inheriting from Existing Windows Forms Controls.
Windows Service	.vb or .cs	A class for creating Windows services. For more information, see Creating Windows Services.
COM Class	.vb or .cs	A class that can be exposed to COM. For more information, see COM Interoperability in Visual Basic and Visual C#.
Transactional Component	.vb or .cs	A class for use with transactional components.
Text File	.txt	An empty text file.
Frameset	.htm	An HTML file that hosts multiple HTML pages. For more information, see Introduction to Framesets.
XSLT File	.xslt	A file used to transform XML documents. For more information, see XSLT Developer's Guide.
StyleSheet	.css	A cascading style sheet used for rich HTML style definitions. For more information, see Creating an External CSS Style Sheet.
Installer Class	.vb or .cs	A class to be invoked at setup time. For more information, see Creating Installation Components.

Crystal Report	.rpt	A Crystal Report file that publishes data to a Windows form. For more information, see Crystal Report Designer Overview.
Bitmap File	.bmp	A blank bitmap image file that can be used for creating simple images.
Cursor File	.cur	A file for creating custom cursors. For more information, see Cursor Creation.
Icon File	.ico	An image file for creating a custom icon. For more information, see Icons.
Assembly Resource File	.resx	A file used to edit and define application resources. For more information, see Resources in Applications.
Assembly Information File	.vb or .cs	A file containing general assembly information. For more information, see Assemblies.
Application Configuration File	.config	A file used to configure application settings. For more information, see Configuration Files.
JScript File	.js	A script file containing JScript code. For more information, see JScript Fundamentals.
VBScript File	.vbs	A script file containing VBScript code. For more information, see VBScript User's Guide.
Windows Script Host	.wsf	A file containing script that is run as a Windows program. For more information, see Using Windows Script Files.

Web Project Items

Project Item	File Extension	Purpose of the Project Item
Web Form	Two files: .aspx and either .vb for Visual Basic or .cs for Visual C#	A form for creating Web applications. For more information, see Web Forms Pages.
Web Service	Two files: .asmx and either .vb for Visual Basic or .cs for Visual C#	A component that provides the ability to exchange interoperable messages using standard protocols such as HTTP, XML, XSD, SOAP and WSDL. For more information, see Creating XML Web Services in Managed Code.
Mobile Web Form	.aspx	A form for creating mobile Web applications. For more information, see Creating ASP.NET Mobile Web Applications.
Class	.vb or .cs	A code file that contains a class declaration.
Module (Visual Basic only)	.vb	A file for storing groups of functions.
Component Class	.vb or .cs	A class for creating components using the visual designer.
Data Form Wizard	.aspx (.vb or .cs for local projects)	A data form for Web applications. For more information, see Data Form Wizard.
DataSet	.xsd	A file for creating an XML schema with **DataSet** classes. For more information, see Visual Studio Tools for Creating Datasets.
Web User Control	.ascx	An ASP.NET server control created using the visual designer. For more information, see Developing ASP.NET Server Controls.
Mobile Web User Control	.ascx	An ASP.NET server control created using the visual designer and used in a mobile Web application. For more information, see Creating ASP.NET Mobile Web Applications.
HTML Page	.htm	An HTML page that can include client-side code. For more information, see Design View, HTML Designer.
Frameset	.htm	An HTML file that hosts multiple HTML pages. For more information, see Introduction to Framesets.
Style Sheet	.css	A cascading style sheet used for rich HTML style definitions. For more information, see Creating an External CSS Style Sheet.
XML File	.xml	A blank XML file. For more information, see XML Designer.
XML Schema	.xsd	A file for creating a schema for XML documents. For more information, see Introduction to XML Schemas.
XSLT File	.xslt	A file used to transform XML documents. For more information, see XSLT Developer's Guide.
Web Custom Control	.vb or .cs	A class for creating an ASP.NET server control. For more information, see ASP.NET Server Controls.

Code File	.vb or .cs	A blank code file.
Static Discovery File	.disco	An optional file that acts as a discovery mechanism for the XML Web service. The .disco file is not automatically created for an XML Web service. For information on creating a discovery file for your XML Web service, see Enabling Discovery for an XML Web Service.
Global Application Class	.asax	Sometimes called an asax file, this file allows you to write code to handle global ASP.NET application-level events such as **Session_OnStart** and **Application_OnStart**. The file has a name of global.asax, which you cannot change. For more information, see The Global.asax File.
Web Configuration File	.config	ASP.NET uses this file to configure the Web settings for a Web project. The file has a name of Web.config, which you cannot change. For more information, see Application Root Directory Configuration File (Web.config).
Text File	.txt	An empty text file.
Installer Class	.vb or .cs	A class to be invoked at setup time. For more information, see Creating Installation Components.
Crystal Report	.rpt	A Crystal Report file that publishes data to a Web Form. For more information, see Crystal Report Designer Overview.
Bitmap File	.bmp	A blank bitmap image file that can be used for creating simple images.
Cursor File	.cur	A file for creating custom cursors. For more information, see Cursor Creation.
Icon File	.ico	An image file for creating a custom icon. For more information, see Icons.
Assembly Resource File	.resx	A file used to edit and define application resources. For more information, see Resources in Applications.
Assembly Information File	.vb or .cs	A file containing general assembly information. For more information, see Assemblies.
JScript File	.js	A script file containing JScript code. For more information, see JScript Fundamentals.
VBScript File	.vbs	A script file containing VBScript code. For more information, see VBScript User's Guide.
Windows Script Host	.wsf	An empty code file used for Windows scripting. For more information, see Introduction to Windows Service Applications.

See Also

What's New in Projects | Visual Basic and Visual C# Projects

Visual Basic and Visual C# Projects

When you create a new project, icons in the New Project dialog box and Add Project dialog box represent the available project types and their templates. The project template that is associated with the project icon you choose determines the output type and other options available for that project.

> **Note** Some project types listed in this topic are not available in the Standard Edition of Visual Basic and Visual C# .NET. For more information, see Visual Basic Standard Edition Features or Visual C# Standard Edition Features.

There are two basic project models in Visual Basic and Visual C#: local and Web. For more information, see Local vs. Web Projects.

The following table lists the types of projects available for Visual Basic and Visual C#.

Project Template	Used To Create
Windows Application Template	Traditional standalone Windows application or a rich front-end to a distributed Web application. For more information, see Creating Windows Applications. **Visual Basic Note** This replaces the Standard EXE project template in Visual Basic 6.0.
Class Library Template	Reusable class or component that can be shared with other projects. This project type is considered windowless and will not contain a Windows Form class. For more information, see Component Classes.
Windows Control Library Template	Custom control to use on Windows Forms. For more information, see Control Authoring for Windows Forms.
ASP.NET Web Application Template	Programmable ASP.NET Web application. For more information, see Introduction to ASP.NET Web Applications in Visual Studio.
ASP.NET Mobile Web Application Template	ASP.NET Web Application viewable on PDAs (personal digital assistants), mobile phones, and other mobile devices. For more information, see ASP.NET Mobile Designer.
ASP.NET Web Service Template	XML Web services authored with ASP.NET, functionality that can be published and called by external applications. For more information, see Creating XML Web Services in Managed Code.
Web Control Library Template	Custom control that can be used on Web Forms pages. For more information, see ASP.NET Server Controls. **Visual Basic Note** This project type is analogous to the ActiveX Control project in Visual Basic 6.0.
Console Application Template	Command-line application. For more information, see Console Application Template.
Windows Service Template	Long running applications that do not have a user interface. Windows Service Applications (formerly called "NT services") can monitor items such as system performance. For more information, see Introduction to Windows Service Applications.
New Project in Existing Folder Template	Blank project within an existing application folder for using files from a pre-existing project. This template can be used with both local and Web projects.
Empty Project Template	An empty project. The template creates the necessary file structure needed to store application information; any references, files, or components must be added manually.
Empty Web Project Template	For advanced users who want to start programming in an empty environment and add their own server-based functionality. For more information, see Web Forms Pages and Projects in Visual Studio.

See Also

What's New in Projects | File Types and File Extensions in Visual Basic and Visual C# | Creating Windows Applications | XML Web Services in Managed Code | Web Forms Pages and Projects in Visual Studio | Projects and Solutions, Environment, Options Dialog Box | Programming with Office

Class Library Template

You can use the Class Library template to quickly create reusable classes and components that can be shared with other projects. For more information about creating component classes, see Component Classes.

> **Note** The Class Library template is not available in the Standard Edition of Visual Basic .NET. For more information, see Visual Basic Standard Edition Features.
>
> **Visual Basic Note** The Class Library template replaces the ActiveX DLL project in Visual Basic 6.0. For more information, see Project Changes in Visual Basic .NET.

The template automatically adds the essential project references and files:

- AssemblyInfo - Used to describe the assembly and specify versioning information. For more information, see Programming with Assemblies.
- Class (.vb file for Visual Basic or .cs for Visual C#) - Contains the empty class module.
- References to these .NET Framework namespaces:
  - System - Contains fundamental classes and base classes that define commonly used values and reference data types, events and event handlers, interfaces, attributes, and processing exceptions.
  - System.Data - Contains classes that constitute the ADO.NET architecture, which is the primary data access method for managed applications.
  - System.Xml - Contains classes that provide standards-based support for processing XML.

See Also

Add New Project Dialog Box | Managing Solutions, Projects, and Files | Item Management in Projects | Adding New Project Items | Component Classes

Console Application Template

The Console Application project template adds the necessary items needed to create a console application. Console applications are typically designed without a graphical user interface and are compiled into a stand-alone executable file. A console application is run from the command line with input and output information being exchanged between the command prompt and the running application. Because information can be written to and read from the console window, this makes the console application a great way to learn new programming techniques without having to be concerned with the user interface.

The template automatically adds the essential project references and files to use as a starting point for your application:

- AssemblyInfo (.vb file for Visual Basic or .cs file for Visual C#) - Used to describe the assembly and specify versioning information. For more information, see Programming with Assemblies.
- Class (.vb file for Visual Basic or .cs for Visual C#) - Contains the empty class module.
- References to these .NET Framework namespaces:
    - System - Contains fundamental classes and base classes that define commonly used values and reference data types, events and event handlers, interfaces, attributes, and processing exceptions.
    - System.Data - Contains classes that constitute the ADO.NET architecture, which is the primary data access method for managed applications.
    - System.Xml - Contains classes that provide standards-based support for processing XML.

See Also

File Types and File Extensions in Visual Basic and Visual C# | Item Management in Projects |
Managing Solutions, Projects, and Files

Empty Project Template

The Empty Project template can be used when you want to create your own project type. The template creates the necessary file structure needed to store application information. Any references, files, or components must be added manually. For more information on adding references, see Adding and Removing References.

> **Note** This project type is not available in the Standard Edition of Visual Basic .NET. For more information, see Visual Basic Standard Edition Features.

See Also

Project Items | File Types and File Extensions in Visual Basic and Visual C# | Adding New Project Items

Empty Web Project Template

The Empty Web Project template is for advanced users who want to start with an empty project. The template creates the necessary file structure for a server-based project on your Internet Information Services (IIS) server. References and components (such as Web Forms pages) must be added manually. For information about Web projects, see Web Forms Pages and Projects in Visual Studio.

> **Note** This project type is not available in the Standard Edition of Visual Basic .NET. For more information, see Visual Basic Standard Edition Features.

Requirements

You must have:

- A computer with IIS installed.
- If you want to use FrontPage Server Extensions as your access method, the server extensions must also be installed on that computer. For more information, see Web Access Methods.
- You must have the correct Web permissions settings for your application. For more information, see Introduction to Web Application Security.

See Also

Web Walkthroughs | Project Items | File Types and File Extensions in Visual Basic and Visual C#

New Project in Existing Folder Template

The New Project in Existing Folder template creates a blank project within an existing application directory. You can then choose to add the files from the pre-existing application directory into this new project by right-clicking each of these items in Solution Explorer and selecting Include in Project on the shortcut menu.

> **Note** This project type is not available in the Standard Edition of Visual Basic and Visual C# .NET. For more information, see Visual Basic Standard Edition Features or Visual C# Standard Edition Features.

After you select this template and enter a project name, the Create a New Project in Existing Folder dialog box appears. You can type in a path or browse to an existing folder.

> **Tip** Click **Show All Files** on the Solution Explorer toolbar to view the pre-existing files.

If you create the new project for an existing Web application on a server, you can browse to an existing Web folder by selecting My Network Places in the browse window and locating the Web directory where you want to place your new project.

> **Note** To create a new Web project in an existing Web folder, you must ensure that the settings for your application are set to File Share. For more information, see Web Access Methods.

See Also

Adding New Project Items | Project Items | File Types and File Extensions in Visual Basic and Visual C#

ASP.NET Web Application Template

The ASP.NET Web Application project template is used to create an ASP.NET Web application on a computer that has Internet Information Services (IIS) version 5.0 or later installed. The template creates the basic files needed on the server to help you start designing your application. For an overview of Web applications, see Introduction to ASP.NET Web Applications in Visual Studio.

Requirements

You must have:

- A computer with IIS, version 5.0 or later, installed locally or on a remote server.
- If you want to use FrontPage Server Extensions as your access method, the server extensions must also be installed on that computer. For more information, see Web Access Methods.
- You must have the correct Web permissions settings for your application. For more information, see Web Application Security at Design Time in Visual Studio.

Files and References Added by the Template

The template automatically adds the essential project references and files to use as a starting point for your application:

- AssemblyInfo (.vb file for Visual Basic or .cs file for Visual C#) - Used to describe the assembly and specify versioning information. For more information, see Programming with Assemblies.
- Global.asax - Contains code for responding to application-level events raised by ASP.NET. For more information, see The Global.asax File.
- Styles.css - Contains default HTML style settings. For more information, see Working With CSS Styles.
- Web.config - Contains application settings for the ASP.NET application. For more information, see Application Configuration Files.
- WebForm1.aspx - Contains the portion of the default Web Forms page containing UI elements (controls), similar to an HTML page. For more information, see Web Forms Pages.
- Webform1 (.vb file for Visual Basic or .cs file for Visual C#) - Contains a class file for the default Web Forms page that contains the system-generated and user code for the page. For more information, see Web Forms Pages.
- References to these .NET Framework namespaces:
    - System - Contains fundamental classes and base classes that define commonly used values and reference data types, events and event handlers, interfaces, attributes, and processing exceptions.
    - System.Data - Contains classes that constitute the ADO.NET architecture, which is the primary data access method for managed applications.
    - System.Drawing - Contains classes that provide access to GDI+ basic graphics functionality.
    - System.Web - Contains classes and interfaces that enable browser/server communication.
    - System.Xml - Contains classes that provide standards-based support for processing XML.

Comparison of Web Application and Web Service Templates

Both the ASP.NET Web Application and ASP.NET Web Service templates include the core references needed to create an application that accesses both Web and data technologies. The main difference between these two templates is the use for which each project type is intended. A Web application is used to create an environment with which users can visually interact through a Web browser, while an XML Web service is used to publish a service that applications can call upon to perform a task and return a value.

The primary structural difference between a Web application and a Web service is the use of ASP.NET Web Forms technology. When the ASP.NET Web Application template creates a project, it includes a Web Forms page, a Styles.css file for storing default HTML settings, and references the System.Drawing namespace. These items are the building blocks for creating interactive Web documents that users can access from a remote computer with a Web browser. For more information, see Introduction to ASP.NET Web Applications in Visual Studio.

The ASP.NET Web Service template does not provide a Web Forms page, but instead includes a service (.asmx) file. This text file serves as the addressable entry point for the XML Web service. It references code in pre-compiled assemblies, a code-behind file, or code contained in the .asmx file itself.

See Also

ASP.NET Web Service Template | Web Forms Pages and Projects in Visual Studio | Working with Web Projects | Web Walkthroughs | File Types and File Extensions in Visual Basic and Visual C# | Web Namespaces in Visual Studio

Web Control Library Template

You can use the Web Control Library project template to author custom Web server controls. The template adds the necessary project items needed to start creating a control that can then be added to any Web project. For more information, see ASP.NET Server Controls.

> **Note** This project type is not available in the Standard Edition of Visual Basic and Visual C# .NET. For more information, see Visual Basic Standard Edition Features or Visual C# Standard Edition Features.
>
> **Visual Basic Note** This project type is analogous to the ActiveX Control project in Visual Basic 6.0. For more information, see Project Changes in Visual Basic .NET.

The template automatically adds the essential project references and files to use as a starting point for your application:

- AssemblyInfo (.vb file for Visual Basic or .cs file for Visual C#) - Used to describe the assembly and specify versioning information. For more information, see Programming with Assemblies.
- WebCustomControl file (.cs file for Visual C# or .vb file for Visual Basic) - Contains the code for your custom control, which is stored in the solution directory.
- References to these .NET Framework namespaces:
  - System - Contains fundamental classes and base classes that define commonly used values and reference data types, events and event handlers, interfaces, attributes, and processing exceptions.
  - System.Data - Contains classes that constitute the ADO.NET architecture, which is the primary data access method for managed applications (added by default to Visual Basic projects only).
  - System.Drawing - Contains classes that provide access to GDI+ basic graphics functionality.
  - System.Management - Contains classes for managing information and events about the system, devices, and applications instrumented to the Windows Management Instrumentation (WMI) infrastructure (added by default to Visual Basic projects only).
  - System.Web - Contains classes and interfaces that enable browser/server communication.
  - System.Xml - Contains classes that provide standards-based support for processing XML (added by default to Visual Basic projects only).

See Also

Component Classes | Developing ASP.NET Server Controls | Web Forms Pages |
File Types and File Extensions in Visual Basic and Visual C# | Developing ASP.NET Server Controls

ASP.NET Web Service Template

Use the ASP.NET Web Service project template to author an XML Web service that can be consumed by other Web services or applications on a network. For more information on XML Web services, see XML Web Services in Managed Code. To understand the difference between the Web service and Web application templates, see ASP.NET Web Application Template.

Requirements

You must have:

- A computer with Internet Information Services (IIS) installed.
- If you want to use FrontPage Server Extensions as your access method, the server extensions must also be installed on that computer. For more information, see Web Access Methods.
- You must have the correct Web permissions settings for your application. For more information, see Web Application Security at Run Time.

Files and References Added by the Template

The template automatically adds the essential project references and files to use as a starting point for your application:

- AssemblyInfo (.vb file for Visual Basic or .cs file for Visual C#) - Used to describe the assembly and specify versioning information. For more information, see Programming with Assemblies.
- Global.asax – Contains code for responding to application-level events raised by ASP.NET. For more information, see The Global.asax File.
- Service file (.vb file for Visual Basic or .cs file for Visual C#) – Contains the class that is used to add functionality to your XML Web service. For more information, see Creating and Accessing XML Web Services Walkthroughs.
- Web.config – Contains a file with application-specific settings. For more information, see Application Configuration Files.
- References to these .NET Framework namespaces:
    - System - Contains fundamental classes and base classes that define commonly used values and reference data types, events and event handlers, interfaces, attributes, and processing exceptions.
    - System.Data - Contains classes that constitute the ADO.NET architecture, which is the primary data access method for managed applications.
    - System.Web - Contains classes and interfaces that enable browser/server communication.
    - System.Web.Services - Contains classes that enable you to build and use XML Web services, programmable entities residing on a Web server and exposed via standard Internet protocols.
    - System.Xml - Contains classes that provide standards-based support for processing XML.

See Also

Introduction to Programming XML Web Services in Managed Code | File Types and File Extensions in Visual Basic and Visual C# | Creating ASP.NET Web Service Projects

Windows Application Template

The Windows Application project template in Visual Basic and Visual C# lays the groundwork for a standard Windows application. For more information on creating a Windows application, see Creating a Windows Application Project.

> **Visual Basic Note** This application replaces the Standard EXE application in Visual Basic 6.0. For more information, see Project Changes in Visual Basic .NET.

The template automatically adds the essential project references and files to use as a starting point for your application:

- AssemblyInfo - Used to describe the assembly and specify versioning information. For more information, see Programming with Assemblies.
- Form (.vb file for Visual Basic or .cs for Visual C#) - Contains the Windows Form.
- References to these .NET Framework namespaces:
  - System - Contains fundamental classes and base classes that define commonly used values and reference data types, events and event handlers, interfaces, attributes, and processing exceptions.
  - System.Data - Contains classes that constitute the ADO.NET architecture, which is the primary data access method for managed applications.
  - System.Drawing - Contains classes that provide access to GDI+ basic graphics functionality.
  - System.Windows.Forms - Contains classes for creating Windows-based applications that take full advantage of the rich user interface features available in the Microsoft Windows operating system.
  - System.Xml - Contains classes that provide standards-based support for processing XML.

See Also

Creating a Windows Application Project | Creating Windows Forms | Managing Solutions, Projects, and Files

Windows Control Library Template

The Windows Control Library project template is used to create custom controls to use on Windows Forms. For more information , see Windows Forms Controls.

> **Note** This project type is not available in the Standard Edition of Visual Basic and Visual C# .NET. For more information, see Visual Basic Standard Edition Features or Visual C# Standard Edition Features.
>
> **Visual Basic Note** This template is analogous to the User Control project in Visual Basic 6.0. For more information, see Project Changes in Visual Basic .NET.

The template automatically adds the essential project references and files to use as a starting point for your application:

- AssemblyInfo (.vb file for Visual Basic or .cs file for Visual C#) - Used to describe the assembly and specify versioning information. For more information, see Programming with Assemblies.
- UserControl file (.cs file for Visual C# or .vb file for Visual Basic) - Contains a class that derives from **System.Windows.Forms.UserControl** and implements a user control.
- References to these .NET Framework namespaces:
  - System - Contains fundamental classes and base classes that define commonly used values and reference data types, events and event handlers, interfaces, attributes, and processing exceptions.
  - System.Data - Contains classes that constitute the ADO.NET architecture, which is the primary data access method for managed applications.
  - System.Drawing - Contains classes that provide access to GDI+ basic graphics functionality.
  - System.Windows.Forms - Contains classes for creating Windows-based applications that take full advantage of the rich user interface features available in the Microsoft Windows operating system.
  - System.Xml - Contains classes that provide standards-based support for processing XML.

See Also

Authoring Components | Component Classes | File Types and File Extensions in Visual Basic and Visual C# | Component Model Namespaces in Visual Studio

Windows Service Template

The Windows Service template adds the necessary items needed to create a Windows Service application, a long-running executable application that runs in its own Windows session. For more information, see Introduction to Windows Service Applications.

> **Note** This project type is not available in the Standard Edition of Visual Basic and Visual C# .NET. For more information, see Visual Basic Standard Edition Features or Visual C# Standard Edition Features.

The template automatically adds the essential project references and files to use as a starting point for your application:

- AssemblyInfo (.vb file for Visual Basic or .cs file for Visual C#) - Used to describe the assembly and specify versioning information. For more information, see Programming with Assemblies.
- Service file (.cs file for Visual C# or .vb file for Visual Basic) - Contains the designer file, which is stored in the solution directory.
- References to these .NET Framework namespaces:
  - System - Contains fundamental classes and base classes that define commonly used values and reference data types, events and event handlers, interfaces, attributes, and processing exceptions.
  - System.Data - Contains classes that constitute the ADO.NET architecture, which is the primary data access method for managed applications.
  - System.ServiceProcess - Contains classes that allow you to install and run services, which are long-running executables with no user interface.
  - System.Xml - Contains classes that provide standards-based support for processing XML.

See Also

File Types and File Extensions in Visual Basic and Visual C# | Managing Solutions, Projects, and Files | Components and Assemblies | Introduction to Monitoring Performance Thresholds

Referencing Namespaces and Components

Project references are used to access the properties, methods, and events of a defined object and apply them to your programming. The following topics contain an overview of references in Visual Basic and Visual C# and give instructions on how to add, remove, and manage existing references.

In This Section

Project References
Describes the basic concepts of project references.
Adding and Removing References
Explains how to add and remove references to assemblies, components, and other projects in the same solution.
Troubleshooting Broken References
Describes the main causes of broken references and ways to fix them.
Managing Project Web References
Describes the procedure used to rename or update a project Web reference.

Related Sections

Adding and Removing Web References
Describes how to add a Web reference using the Project menu or Server Explorer.
Introduction to the .NET Framework Class Library in Visual Studio
Describes the .NET class library.

Project References

In order to write code against an external component, your project must first contain a reference to it. A reference can be made to the following types of components:

- .NET class libraries or assemblies
- COM components
- Other class libraries of projects in the same solution
- XML Web services

For more information about XML Web service references, see Web References.

> **Visual Basic Note** Project references are managed differently in Visual Studio .NET than they were in Visual Basic 6.0. For more information, see Project Changes in Visual Basic .NET.

References to Shared Components

At run time, components must be in one of two locations: in the output path of the project or the global assembly cache (see Global Assembly Cache). If the project contains a reference to an object that is not in one of these locations, the reference must be copied to the output path of the project when the project is built. The CopyLocal property indicates whether this copy needs to be made. If the value is **true**, the reference is copied. If **false**, the reference is not copied.

By default, the **CopyLocal** property is set to **false** if the assembly or component is in the global assembly cache or is a framework component. Otherwise, the value is set to **true**. Project-to-project references are always set to **true**.

Project-to-Project References

In a solution with multiple projects, you can create references to objects authored in another project that resides in the same solution. This creates an interdependency that requires special consideration when building and running the application. For more information, see Preparing and Managing Builds.

See Also

Reference Path, Common Properties, <Projectname> Property Pages Dialog Box | Adding and Removing References | Troubleshooting Broken References | Referencing Namespaces and Components | Programming with Assemblies

Adding and Removing References

In order to use a component in your application, you must first add a reference to it. Visual Studio .NET provides three options on the Add Reference dialog box:

- .NET - Lists all .NET Framework components available for referencing.
- COM - Lists all COM components available for referencing.
- Projects - Lists all reusable components created in local projects.

You can also add Web references in the Add Web Reference dialog box. For more information, see Adding and Removing Web References.

> **Note** You should avoid adding file references to outputs of another project in the same solution, because doing so may cause compilation errors. Instead, use the Projects tab of the Add References dialog box to create project-to-project references. This makes team development easier, by allowing for better management of the class libraries you create in your projects. For more information, see Troubleshooting Broken References.

To add a reference

1. In Solution Explorer, expand the project node you want to add a reference to.
2. Right-click the **References** node for the project and select **Add Reference** from the shortcut menu.
3. To add a reference to a component or components, do the following:
   a. In the **Add Reference** dialog box, select the tab indicating the type of component you want to reference. For more information, see Add Reference Dialog Box.
   b. In the top pane, select the component you want to reference, and then click the **Select** button.

      > **Tip** If the component you are looking for is not in the list, you may locate it using the **Browse** button.

      The component you referenced appears in the **SelectedComponents** pane of the dialog box. Repeat Step b for each additional component you wish to add.

   c. Click **OK** when you have added references to all the components you need.

      Selected references will appear under the **References** node of the project.

To remove a reference

- In Solution Explorer, right-click the reference in the **References** node, and select **Remove** from the shortcut menu.

By default, the Add References dialog box only lists assemblies that are in the Public Assemblies folder (Program Files\Microsoft Visual Studio .NET\Common7\IDE\Public Assemblies) or in the Global Assembly cache. You can add your own assemblies to the list by adding a registry key to specify their location.

To add assemblies to the Add References dialog box

1. In Windows, click the **Start** button, click **Run**, then type **regedit** to open the Registry Editor.
2. Select the registry key **HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\VisualStudio\7.0\AssemblyFolders**
3. On the **Edit** menu, point to **New**, then click **Key**.
4. Enter a name for the key (for example, **MyAssemblies**).
5. Select the **(Default)** value for the key that you just added.
6. On the **Edit** menu, choose **Modify**.
7. In the **Edit String** dialog box, select the **Value data** field and enter the full path to the folder where your assemblies are located.

   > **Note** Only the specified folder will be searched each time the Add References dialog box is opened; subfolders will not be searched. You must add a separate key for each subdirectory to be searched.

See Also

Troubleshooting Broken References | Referencing Namespaces and Components | Managing Project Web References | Adding New Project Items

Troubleshooting Broken References

If your application attempts to use a broken reference, an exception error is generated. The inability to find the referenced component is the primary trigger for the error, but there are several situations in which a reference can be considered broken. These instances are shown in the following list:

- The project's reference path is incorrect or incomplete.
- The file being referenced has been deleted.
- The file being referenced has been renamed.
- The network connection or authentication has failed.
- The reference is to a COM component that is not installed on the computer.

The following are remedies to these problems.

> **Note** Files in assemblies are referenced with absolute paths in the project file. Because of this, it is possible for users working in a multideveloper environment to be missing a referenced assembly in their local environment. To avoid these errors, it is better in these cases to add project-to-project references. For more information, see Adding and Removing References and Programming with Assemblies.

Reference Path Is Incorrect

If projects are shared on different computers, some references might not be found when a component is located in a different directory on each computer. References are stored under the name of the component file (for example, MyComponent). When a reference is added to a project, the folder location of the component file (for example, C:\MyComponents\) is appended to the Reference Path project property.

When the project is opened, it attempts to locate these referenced component files by looking in the directories on the reference path. If the project is opened on a computer that stores the component in a different directory, such as D:\MyComponents\, the reference cannot be found and an error appears in the Task List.

To fix this problem, you can delete the broken reference and then replace it using the Add Reference dialog box. Another solution is to use the Reference Path item in the project's property pages and modify the folders in the list to point to the correct locations. The Reference Path property is persisted for each user on each computer, so modifying your reference path does not affect other users of the project.

> **Tip** Project-to-project references do not have these problems. For this reason, use them instead of file references, if possible. For more information, see Components and Assemblies.

To fix a broken project reference by correcting the reference path

1. In Solution Explorer, right-click your project node and choose **Properties** from the shortcut menu.
2. Select the appropriate property page in the dialog box.

   - If you are using Visual Basic, select **Reference Path** under the **Common Properties** folder in the left pane. In the **Reference Path** pane, type the path in the **Folder** field, and then click the **Add Folder** button.

     -or-

   - If you are using Visual C#, select **References Path** under the **Common Properties** folder in the left pane. Click in the large edit box to get an edit point, and then type the path to the folder that contains the item you want to reference and click **OK**.

Referenced File Has Been Deleted

It is possible that the file being referenced has been deleted and no longer exists on the drive.

To fix a broken project reference for a file that no longer exists on your drive

- Delete the reference. For more information, see Adding and Removing References.
- If the reference exists in another location on your computer, read it from that location. For more information, see Adding and Removing References.

Referenced File Has Been Renamed

It is possible that the file being referenced has been renamed.

To fix a broken reference for a file that has been renamed

- Delete the reference, and then add a reference to the renamed file.

  > **Tip** If the reference exists in another location on your computer, you need to read it in from that location. For more information, see Adding and Removing References.

Network Connection or Authentication Has Failed

There can be a variety of causes for inaccessible files: a failed network connection or a failed authentication, for example. Each cause might have a unique means of recovery; for example, you might need to contact your local administrator for access to the required resources. However, deleting the reference and fixing the code which used it is always an option. For more information, see Adding and Removing References.

COM Component Is Not Installed on Computer

If a user has added a reference to a COM component and a second user attempts to run the code on a computer that does not have this component installed, the second user will receive an error that the reference is broken. Installing the component on the second computer will correct the error. For more information about using references to COM components in your projects see COM Interoperability in Visual Basic and Visual C#.

See Also

Referencing Namespaces and Components | Managing Projects | Adding and Removing References | Add Reference dialog box | Reference Path

Managing Project Web References

Web references differ from traditional references and components in that they refer to XML Web services published on either a local intranet or the Internet. The standard references that programmers are accustomed to adding to their applications are to components and class libraries installed on the local computer. You add a Web reference to your project when you want to incorporate functionality from a Web service into your application. For instructions on creating a Web reference, see Adding and Removing Web References.

You manage project Web references by renaming or modifying them to suit your needs or by updating them when changes have been made to the Web service the reference calls.

Renaming a Project Web Reference

The Web reference's name is also its programmatic namespace. The namespace can be changed to suit your needs. You can change the name of the Web reference to any string expression that is meaningful to its function. For example, you could change the name of a Web reference that handles account totals to "AccountTotal."

To rename a Web reference in a project

1. In Solution Explorer, expand your project's **Web References** folder, and then select the node for the reference you want to rename.
2. On the **File** menu, choose **Rename**.

   A box appears around the selected reference.

3. Type the new name for the reference, and press **ENTER**.

   Note If you have used the old name elsewhere in your project code, remember to change each occurrence to the new name.

Updating a Project Web Reference

If your application contains a Web reference to an XML Web service that has been recently modified on the server, you may need to update the reference in your project. For example, you may need to update a Web reference when new methods are added to an existing XML Web service on a server. In this case, you would need to generate a new proxy for the Web service so you can access the new method; this happens automatically when you update the Web reference.

To update a project Web reference

1. In Solution Explorer, access your project's **Web References** folder and select the node for the Web reference you want to update.
2. Right-click the reference and click **Update Web Reference**.

   The new files for the XML Web service are downloaded to your project. Information for the XML Web service is updated within your project.

URL Behavior Property

Each Web reference in your project interacts with a proxy class that is used to access the Web service's contents. You can use the UrlBehavior property to control the way a proxy class obtains the URL for a specific Web reference. The property has two settings:

Setting	Effect
Static	The proxy class sets the URL using a hard-coded URL when you created the instance of the class.
Dynamic	The application obtains the URL at run time from the <appSettings> Element element of your application's configuration file.

For more information on dynamic versus static Web references, see Locating XML Web Services.

See Also

Adding and Removing Web References | Add Web Reference Dialog Box | Web Access Methods | Creating Web Projects | Opening Web Projects | Changing Web Project Access Methods | Visual Basic and Visual C# Projects | Referencing Namespaces and Components

Working with Web Projects

ASP.NET Web applications and XML Web services allow you to create applications that leverage the power of the World Wide Web. These types of projects run on a Web server, sending and receiving information using standard protocols such as HTTP and XML. Using Web projects, you can create everything from a traditional Web site that serves HTML pages, to fully featured business applications that run on an intranet or the Internet, to sophisticated business-to-business applications.

In This Section

Introduction to Web Projects
Provides an introduction to the features of Web projects in Visual Studio .NET.
Web Access Methods
Explains the issues you face in choosing how your Web project files will be accessed and how to choose the correct method.
Creating Web Projects
Gives directions for how to create a new Web project.
Opening Web Projects
Provides instructions for opening existing Web projects and adding them to a solution.
Setting Web Permissions
Provides instructions for setting permissions that control who can access the files in your Web project in what way.
Taking Web Projects Offline
Describes how to take Web projects offline and synchronize or refresh them.
Troubleshooting Web Projects
Explains how to fix some common problems encountered when working with Web projects.

Related Sections

Local vs. Web Projects
Describes differences between directory-based Web applications and file-based local applications.
Managing Project Web References
Provides instructions for renaming and updating references to XML Web services.
Adding and Removing Web References
Provides instructions for exposing XML Web services in your application.
Visual Basic and Visual C# Projects
Describes the types of projects available in Visual Basic and Visual C#.
What's New in Projects
Describes the major new project management features of Visual Studio .NET.
Project Changes in Visual Basic .NET
Describes the major differences between projects in Visual Basic 6.0 and Visual Basic .NET.
Deployment of a Web Setup Project
Provides an introduction to deploying to a Web server.
Deploying XML Web Services in Managed Code
Gives directions for how to deploy XML Web services to a network.
Walkthrough: Deploying a Web Solution
Provides step-by-step instructions for deploying a simple ASP.NET Web application.
Upgrading Visual InterDev 6.0 Applications to Visual Studio .NET
Describes how Visual InterDev 6 applications relate to Visual Studio .NET.

Introduction to Web Projects

There are many types of projects available in Visual Studio .NET, each designed for a specific purpose. Each project contains specific types of files, references, and default items. In Visual Basic and Visual C#, two of the major kinds of projects you can create are Web-based projects and local projects. Web projects are stored and managed on a Web server with the purpose of being exposed to a network, whereas local projects are stored locally and not optimized for network access.

Visual Basic and Visual C# include templates to help you create three types of Web-based projects:

- ASP.NET Web Application template - a programmable Web application with designer support for creating Web Forms pages and incorporating features of ASP.NET.
- ASP.NET Web Service template - an application that provides the ability to exchange interoperable messages across the Web using standard protocols such as HTTP, XML, XSD, SOAP, and WSDL.
- Empty Web Project template - an empty Web application you can customize.

> **Note** There is also a Web Control Library template, but it is actually a local project.
>
> **Note** The Empty Web Project and Web Control Library templates are not available in the Standard Edition of Visual Basic .NET; the Web Control Library template is not available in the Standard Edition of Visual C# .NET. For more information, see Visual Basic Standard Edition Features or Visual C# Standard Edition Features.

For more information about Web projects, see Working with Web Projects. For more information about choosing between Web and local projects, see Windows Forms and Web Forms Recommendations and Local vs. Web Projects.

Web Project User Interface

When you create a Web application in Visual Basic and Visual C#, you create programmable Web pages that serve as the user interface for your Web application. A Web Forms page presents information to the user in any browser or client device and implements application logic using server-side code. Web Forms page output may contain almost any HTTP-capable language, including HTML, XML, WML, and ECMAScript (JScript, JavaScript). For more information, see Introduction to Web Forms Pages.

Access and Storage for Web Project Files

You can choose two ways for Visual Studio to manage the files in your ASP.NET Web application:

- **FrontPage Server Extensions access** - All files are managed using the HTTP protocol.
- **File-share access** - All files are managed using Windows-based file management commands. This option requires LAN access to the server, but it can be slightly faster because there is less management overhead. In addition, file-share access provides superior support for source-code control.

For more information on access methods and choosing the correct method, see Web Access Methods.

In addition, Visual Studio keeps files in a cache on the local computer. The local directory is created and used for two reasons:

- Certain designers require file-level write access to some of your project files. If you are using FrontPage Server Extensions, you access the server using HTTP, which would preclude these designers from accessing the files they need.
- Keeping files in the cache allows you to work on your Web Forms pages in Offline mode. For more information, see Taking Web Projects Offline.

The cache is always kept in sync with the actual project files on the server. (If you are in Offline mode, the files are synchronized when you reconnect to the server project.) When you change a file in your project, the update is written to both the server and the cache. In a few instances, files are kept in the cache but not written to the server, as with intermediate files created during a build process.

Essentially, the cache is a temporary directory used by Visual Studio. You will see references to the cache when you are creating a deployment project. However, you should not otherwise have any cause to see or work with the cache.

Web Project Security

On the file level, you can use the Web permissions feature of Visual Studio to control who can gain access to your Web project files and what level of access individuals have to the application. The permissions you set in Visual Studio are applied to Windows entities (folders, users, groups, and so on.) They are similar to equivalent permissions you can set in Internet Information Services (IIS). You can choose to set individual rights or simply inherit the rights of the Web folder that the application is stored in. For more information, see Setting Web Permissions.

On an application level, security concerns are a critical part of the decisions you must make in designing and creating your Web application. Web applications by definition allow users access to a central resource — the Web server — and through it, to others such as database servers. By understanding and implementing proper security measures, you guard your own resources as well as provide a secure environment in which your users are comfortable working with your application. For more information on the security considerations you must face when creating your application, see Introduction to Web Application Security.

Debugging and Deploying Web Projects

When you build a Web project, the application files are remotely compiled on the Web server. Information is sent to the Web application, processed, and the results are then returned to the client computer and viewed as a Web Form or an update of information on a Web Form. For more information, see Debugging Preparation: ASP.NET Web Applications.

You can use deployment projects to quickly and easily deploy your Web applications to the appropriate servers. For more information, see Walkthrough: Deploying a Web Solution.

See Also

Creating Web Projects | Working with Web Projects | Web Forms Pages | ASP.NET Applications | Creating Web Applications and Services | Introduction to ASP.NET Web Applications in Visual Studio | Visual Basic and Visual C# Projects | Debugging Preparation: ASP.NET Web Applications | Walkthrough: Deploying a Web Solution | Introduction to Web Application Security | Setting Web Permissions

Web Access Methods

You can choose two ways for Visual Studio to manage the files in your ASP.NET Web application:

- **FrontPage Server Extensions access** - All files are managed using the HTTP protocol.
- **File-share access** - All files are managed using Windows-based file management commands. This option requires LAN access to the server, but it can be slightly faster because there is less management overhead. In addition, file-share access provides superior support for source-code control.

> **Note** If you create your Web application using FrontPage access on a server with a FAT or FAT32 partition, you will not be able to use source control. This is because the FAT partition does not allow for full security functionality, so all operations are performed in the context of the same guest account. All source control operations are performed in the context of this anonymous user (IUSR_Machinename). This can incorrectly cause your files to appear to be exclusively checked out to another user when you attempt to perform check-in or check-out operations. To resolve this problem, convert your FAT partition to NTFS file system.

Choosing the Best Access Method

There are two primary items to consider when choosing the appropriate Web access method. First, you need to consider the location of the server you plan to use. File share requires that you have sufficient access rights on the server to create a file structure. If you do not have access to a shared folder on the Web server, you cannot use the file share access method. For this reason, file share is more appropriate for creating applications on the local intranet. FrontPage Server Extensions are a better choice for developing projects on the Internet, because they use reference pointers to the server.

Second, you need to consider the source control model you prefer to use. When you create a Web project with FrontPage as the access method, only one copy of the Web project exists and all developers work in this single project. Each developer checks out a file exclusively, meaning that only one developer can edit a file at a time. In addition, developers can only access the project using FrontPage Server Extensions.

> **Note** Using FrontPage Server Extensions as your access method is not recommended for a multiuser development environment.

A project using the file share access method behaves similarly to local projects under source control. Developers check out copies of the project files from source control and then check their changes back into the same source control system.

When to Specify the Web Access Method

The default access method for Web projects is file share. However, before you create a new Web project, you can set the preferred Web access method in the Web Settings page of the Project node in the Options dialog box. For more information on creating a new Web project, see Creating Web Projects.

After a Web project is created, the Web access method is stored and used the next time the project is opened. If there is a problem, you can switch access methods by using the Web Access Failed dialog box. For more details on the information in this dialog box see Web Access Failed Dialog Box.

You can change a Web project's access method in the Project Property Pages dialog box as long as the Web project is not under source control. Visual Studio will use the new access method the next time the Web project is opened. For more information about choosing a different access method, see Changing Web Project Access Methods.

See Also

Changing Web Project Access Methods | Web Access Failed Dialog Box | Compilation and Deployment of Web Projects | Creating Web Projects | Opening Web Projects | Managing Project Web References | Visual Basic and Visual C# Projects

Changing Web Project Access Methods

Web applications can access files on a network by using either file share or FrontPage Server Extensions access methods. For more information about which Web access method to use, see "Choosing the Best Access Method" in Web Access Methods. You can change the Web Access method for a Visual Basic project, but the change will not take effect until the next time the project is opened.

For more information about changing the default Web access method, see Creating Web Projects.

> **Note** If you change the Web access method to file share, you must specify a valid path to your Web project. A fully qualified path to your Web folder must be added in the Web Settings page of the Project Properties dialog box.

To change an existing Web project's access method

1. In Solution Explorer, right-click the project node, then choose **Properties** from the shortcut menu.
2. In the **Common Properties** folder in the left pane, click **Web Settings**.
   - Choose either **File share** or **FrontPage** from the **Web Access Method** section.
3. Close your project and reload it.

See Also

Web Access Methods | Web Settings, Common Properties, <Projectname> Property Pages Dialog Box (C#) | Web Settings, Common Properties, <Projectname> Property Pages Dialog Box (Visual Basic) | Visual Basic and Visual C# Projects | Project Type Changes in Visual Basic .NET | Web Projects | Creating Web Projects | Working with Web Projects | Opening Web Projects

Creating Web Projects

A Web project can be either a standard ASP.NET Web application or an XML Web service that is designed to be consumed by other applications. For more information, see Creating Web Applications and Services.

The Web server on which your project exists can be on either a local or remote computer. When you create a new Web project, you must specify the type of file access to use; Visual Studio .NET uses file-share Web access as the default for all your Web projects, unless you change the default setting in the Options dialog box. To determine which Web access method to use, see "Choosing the Best Access Method" in Web Access Methods. If you want to change the Web access method, see Changing Web Project Access Methods.

> **Note** Changing the default Web access method in the Options dialog box does not affect a project you have open; the setting affects only future Web projects. Therefore, you should usually specify a Web access method before you create a new Web project.

To specify the default Web access method

1. On the **Tools** menu, choose **Options**.
2. Expand the **Projects** node in the left pane, and then select **Web Settings**.
3. In the **Preferred Access Method** section, choose either the **File share** or **FrontPage** access method.

   All new Web projects you create will use that Web access method as the default. If you want to change the Web access method for an existing Web project, see Changing Web Project Access Methods.

To create a new Web project

1. On the **File** menu, point to **New** and then click **Project** to display the **New Project** dialog box.
2. In the **Project Types** pane, select either **Visual Basic** or **Visual C#**.
3. In the **Templates** pane, select the type of Web project you want to create. For an overview of Web project types, see Visual Basic and Visual C# Projects.

   - **ASP.NET Web application** - For more information, see Introduction to Web Forms Pages.

     - or -

   - **ASP.NET Web Service** - For more information, see Introduction to Programming XML Web Services in Managed Code.

     - or -

   - **Empty Web Project** - For more information, see Empty Web Project Template.

     > **Note** The Empty Web project type is not available in the Standard Edition of Visual Basic .NET. For more information, see Visual Basic Standard Edition Features.

4. In the **Location** text box, specify the URL where you want to place the Web project, including the name you want to give it.

   For example, to create a project titled myWebApp at the default localhost location, the URL string placed in the Location text box will look like this:

   ```
   http://localhost/myWebApp
   ```

   This allows for exact placement of your project files in a specific directory on a Web server.

   > **Note** If the local computer you are working on is also a Web server, you can create your Web project on that computer. The first time you start Visual Studio, the default location of a Web project is your local computer. In subsequent Visual Studio sessions, you can select the location of a previously used Web server from a drop-down list. If you want to choose a different server, you can click the **Browse** button to find other servers on your network, or you can type in the URL to a server (for example, http://domain.com) and press ENTER.

5. Click **OK** to create the Web project.

   Visual Studio connects to the server and adds the appropriate project items to Solution Explorer in the project node.

See Also

Web Access Methods | Opening Web Projects | Working with Web Projects | Changing Web Project Access Methods | Visual Basic and Visual C# Projects

Opening Web Projects

The method you use to open a Web project depends on the source-control options you have chosen for the project. If you are using source control to manage your project files, you must open and save project files according to the guidelines set in the Source Control node of the Options dialog box. For more information on using source control to manage your project files, see Source Control Services.

For Web projects that are not under source control, project files stored on a server can be opened simultaneously by multiple users. When the project is opened, copies of the project files are stored on your local computer. Before executing a build command, Visual Studio replaces the project files stored on the server with the edited files located in your local directory. If you execute the Build command either directly or when debugging your project, and there is another developer currently working on the project, you will overwrite any project files saved by the second developer.

When you open a Web project, the system opens it using the Web access method (file share or FrontPage Server Extensions) last used for the project. For more information about choosing a Web access method, see Web Access Methods.

To open an existing Web project that is not part of source control

1. On the **File** menu, point to **Open**, and then choose **Project From Web**.
2. In the **Open Project From Web** dialog box, enter the URL of the Web server where the project is located, and then click **OK**.
3. In the **Open Project** dialog box, double-click your Web project's folder.
4. Double-click the appropriate project file to open the project.

   The Web project opens and its files are listed in Solution Explorer. If the project could not be opened, an error message is shown that allows you to try another Web access method or to work offline. For more information, see Troubleshooting Web Projects.

See Also

Web Access Methods | Creating Web Projects | Changing Web Project Access Methods | Visual Basic and Visual C# Projects | Working with Web Projects | Source Control Services

Setting Web Permissions

Using the Web permissions facilities of Visual Studio, you can control who can gain access to your Web project files and what level of access individuals have to the application. The permissions you set in Visual Studio are applied to Windows entities (folders, users, groups, and so on.) They are similar to equivalent permissions you can set in Internet Information Services (IIS). You can choose to set individual rights or simply inherit the rights of the Web folder that the application is stored in. For more information, see the topics on permissions in the IIS documentation (http://www.microsoft.com/windows2000/en/server/iis/).

To inherit the rights for a Web project

1. From the **Project** menu, click **Web Projects**, and then select **Web Permissions** to open the **Web Permissions** dialog box.

   Note You can use the Web permissions facility of Visual Studio only if the Web access method of your project is FrontPage Server Extensions. For more information, see Web Access Methods.

2. In the **Settings** tab, select **Use same permissions as the root Web server**.

To set individual rights for a Web project

1. From the **Project** menu, click **Web Projects**, and then select **Web Permissions** to open the **Web Permissions** dialog box.

   Note You can use the Web permissions facility of Visual Studio only if the Web access method of your project is FrontPage Server Extensions. For more information, see Web Access Methods.

2. Add and set your user's access by doing the following:
   a. In the **Users** tab, click the **Add** button to add new users to the list.
   b. Select the user you added and click the **Edit** button to open the **Edit Users** dialog box.
   c. Select the appropriate user access rights, and then click **OK**.
   d. In the **Groups** tab, click **Add** to add a user group to the list.
   e. Repeat for each user you want to add.
3. Set the computer rights necessary as follows:
4. In the **Computers** tab, click **Add** to add a new computer.
   a. In the **Add Computer** dialog box, type in the IP address for the desired computer, select the required access level, and then click **OK**.
   b. Repeat for each computer you want to add to the list.

See Also

Working with Web Projects | Project Properties | Web Forms Pages and Projects in Visual Studio | Working with Web Projects | Setting Visual Basic Project Properties

Setting Unique Permissions for an ASP.NET Web Application

When you set up access rights to your ASP.NET Web application, you have a choice between inheriting the permissions set on the default Web root where your application exists, or setting individual rights for users, groups, and computers. For more information, see Setting Web Permissions.

> **Note** You can use the Web permissions facility of Visual Studio only if the Web access of your project is FrontPage Server Extensions. For more information, see Web Access Methods.

To set unique permissions for a Web application

1. In Solution Explorer, select the project you want to set permissions for.
2. On the **Projects** menu, click **Web Project** and then **Web Permissions**.

   > **Note** Your project must use Front Page Server Extensions in order for this item to be available.

3. On the **Settings** tab, select **Use unique permissions for this Web application**.

   This option specifies that the current Web application does not inherit its permissions setting from the root Web application.

4. Choose **Apply**.

   > **Note** You must apply the change to unique settings in this tab before the other tabs are available for changes.

5. Set the appropriate rights:

   - To set rights for individual users, click the **Users** tab. For more information, see Manipulating User Permissions for ASP.NET Web Applications.
   - To set rights for groups of users, click the **Groups** tab. For more information, see Manipulating Group Permissions for ASP.NET Web Applications.
   - To set rights for computers by IP address, click the **Computers** tab. For more information, see Manipulating Computer Permissions to ASP.NET Web Applications.

See Also

Settings Tab, Web Permissions Dialog Box | Manipulating User Permissions for ASP.NET Web Applications | Manipulating Group Permissions for ASP.NET Web Applications | Manipulating Computer Permissions to ASP.NET Web Applications

Manipulating Computer Permissions to ASP.NET Web Applications

You can use the Web Permissions dialog box to manipulate computer permissions to your ASP.NET Web applications in a variety of ways:

- You can add the names of computers you want to have access to your application and set their permission levels.
- You can modify permission levels for a computer.
- You can remove a computer's permissions to your applications.

> **Note** You can use the Web permissions facility of Visual Studio only if the Web access of your project is FrontPage Server Extensions. For more information, see Web Access Methods.

To add a computer to a Web application

1. On the **Projects** menu, point to **Web Project**, and then click **Web Permissions**.
2. On the **Settings** tab, select **Use unique permissions for this Web application** and then click **Apply**.
3. On the **Computers** tab, click **Add**.
4. In the **IP Address** boxes, type the address for the computer.

   > **Note** You can specify a group of computers that have the same initial address by using an asterisk (*).

5. In the **Computer permissions** box, select the permission level for the computer.
6. Click **Apply**.

Once a computer has been given permissions, you can change the permissions of the computers.

To change permissions for a computer

1. On the **Projects** menu, point to **Web Project**, and then click **Web Permissions**.
2. On the **Computers** tab, select the computer or computer mask to edit, and click **Edit**.
3. In the **Edit Computer** dialog box, select the new level of permission for the group.
4. Click **Apply**.

See Also

Computers Tab, Web Permissions Dialog Box

Manipulating User Permissions for ASP.NET Web Applications

You can use the Web Permissions dialog box to manipulate user permissions to your ASP.NET Web applications in a variety of ways:

- You can add the names of users you want to have access to your application and set their permission levels.
- You can modify permission levels for a user.
- You can remove a user's permissions to your applications.

> **Note** You can use the Web permissions facility of Visual Studio only if the Web access of your project is FrontPage Server Extensions. For more information, see Web Access Methods.

To add users to a Web application

1. On the **Projects** menu, point to **Web Project**, and then click **Web Permissions**.
2. On the **Settings** tab, select **Use unique permissions for this Web application** and then click **Apply**.
3. On the **Users** tab, click **Add** and specify the user to add:

   - In the **Add Names** box, type the domain and user name for the new user in this format: *domain**username*.

   –or–

   - In the **Obtain list from** box, select the domain and add users from the **Names** field.

4. In the **New users can** box, select the permission level of the new users. Note that the selected permission level applies to all new users in the **Add Names** field.
5. Choose **Apply**.

You can change the permissions for each user.

To change permissions for a user

1. On the **Projects** menu, point to **Web Project**, and then click **Web Permissions**.
2. On the **Users** tab, select the user to edit and click **Edit**.
3. In the **Edit Users** box, select the new level of permission for the user.
4. Choose **Apply**.

You can revoke the permissions that an individual has by deleting an item from the user list.

To delete user permissions on a Web application

1. On the **Projects** menu, point to **Web Project**, and then click **Web Permissions**.
2. Select the user to remove from the **Users** tab.

   > **Note** If your master Web server is on an operating system using the file allocation table (FAT) file system instead of the NTFS file system, you cannot change the permissions for groups.

3. Click **Remove**.
4. Click **Apply**.

See Also

Users Tab, Web Permissions Dialog Box

Manipulating Group Permissions for ASP.NET Web Applications

You can use the Web Permissions dialog box to manipulate user permissions to your ASP.NET Web applications in a variety of ways:

- You can add the names of groups you want to have access to your application and set their permission levels.
- You can modify permission levels for a group.
- You can remove a group's permissions to your applications.

> **Note** You can use the Web permissions facility of Visual Studio only if the Web access of your project is FrontPage Server Extensions. For more information, see Web Access Methods.

To add groups of users to a Web application

1. On the **Projects** menu, point to **Web Project**, and then click **Web Permissions**.
2. On the **Settings** tab, select **Use unique permissions for this Web application** and then click **Apply**.
3. On the **Groups** tab, click **Add**.
4. In the **Add Names** box, type the domain and group name for the new groups in this format: *domain\group name*.

   –or–

   In the **Obtain list from** box, select the domain and add groups from the **Names** field.
5. In the **New users can** box, select the permission level of the new groups. Note that the selected permission level applies to all new groups in the **Add Names** field.
6. Choose **Apply**.

Once a group has been given permissions, you can change the permissions of the group.

To change permissions for a group

1. On the **Projects** menu, point to **Web Project**, and then click **Web Permissions**.
2. On the **Groups** tab select the group to edit, and click **Edit**.
3. In the **Edit Groups** dialog box, select the new level of permission for the group.
4. Choose **Apply**.

You can revoke the permissions that a group has by deleting the group from the group list.

To delete group permissions on a Web application

1. On the **Projects** menu, point to **Web Project**, and then click **Web Permissions**.
2. Select the group to remove from the **Group** tab.

   > **Note** If your master Web server is on an operating system using the file allocation table (FAT) file system instead of the NTFS file system, you cannot change the permissions for groups.

3. Click **Remove**.
4. Click **Apply**.

See Also

Groups Tab, Web Permissions Dialog Box

Taking Web Projects Offline

Visual Studio gives you the ability to make changes to your Web application while working away from your network. This feature is useful when you need to make source code changes at times when you do not have access to your network. For example, you may need to make changes while traveling with a laptop. If you need to work on your project while offline, you can run and debug your Web project without having a network connection.

> **Note** Any Web references in your application that point to procedures on external computers will not be accessible offline. Procedures using these references will throw exception errors.

When your connection is restored, you can synchronize your folders. Visual Studio compares your local application files with the files stored on the server. If Visual Studio finds differences between the two sets of files, it will prompt you to correct the differences. The synchronization process is bidirectional, meaning that changes reflected on server files can be incorporated to your local directory.

Another consideration when working offline is the source-control options you have chosen for your project files. These options have a direct effect on the way you access and save changes to your source files. For more information, see Source Control Services.

To take a Web project offline

- From the **Project** menu, click **Web Project**, and then select **Work Offline**.

  Visual Studio updates the files stored in the local Web cache directory by replacing them with the current files located on the server.

  > **Note** The location of the local Web cache is specified in the **Web Options** dialog box. For more information about Web options, see Web Settings, Projects, Options Dialog Box.

To synchronize all Web project files

- From the **Project** menu, click **Web Project**, and then select **Synchronize All Folders**.

To examine possible differences between online and offline Web project files

- From the **Project** menu, click **Web Project**, and then select **Compare to Master Web**. ("Master Web" means the files on the server.)

Refreshing Web Project Files

Note the difference between refreshing and synchronizing. Refreshing files makes the Web project files located on your computer (the local files) identical to the files on the server by copying files to the local computer or removing files that do not exist on the server. Synchronizing compares local and server files and updates both sets of files with changes found during the comparison.

To refresh Web project files

- Click the **Refresh** button on the Solution Explorer toolbar.

See Also

Working with Web Projects | Solution Items | Solution Explorer | Source Control Services | Web Settings, Projects, Options Dialog Box

Troubleshooting Web Projects

These topics help solve common problems that may occur when creating or opening Web projects in Visual Studio .NET.

In the Visual Basic and Visual C# Documentation

Web Access Failed Dialog Box
Provides help for error messages that may arise when creating and opening Web projects.
Setting Web Permissions
Provides help for setting Web permissions for your Web applications.
Troubleshooting Broken References
Provides help for various problems that you might encounter when adding references to files and components.
Troubleshooting Problems Portal
Provides links to problem-solving topics for Visual Basic and Visual C#.
Web Access Methods
Provides information about managing files in applications, and the appropriate method when using source control.

Additional Information

Getting Assistance
Provides links to information about product support and Help.

Resource File Naming Conventions

This section describes the naming convention for assembly manifest resources in Visual Basic and Visual C# projects.

In This Section

XML Resources
Covers those files whose extension is .resx and whose **BuildAction** property is set to **Embedded Resource**.

Non-XML Resources
Covers any file whose **BuildAction** property is **EmbeddedResource** and whose extension is something other than .resx.

Related Sections

Managing Projects
Provides links to topics about Visual Basic and Visual C# projects.

XML Resources

An XML resource file is a file whose extension is .resx and whose **BuildAction** property is set to **Embedded Resource**. Such XML files must follow a specific schema understood by the ResXToResources conversion tool. With this tool, XML resource files are transformed into binary large objects (BLOBs) prior to inclusion in the assembly manifest.

The following is the file naming convention for such resources:

base_filename[**.***optional RFC 1766 culture info string*]**.resx**

Aside from file system limitations, there are no restrictions on *base_filename* or the optional *RFC 1766 culture info string*. Examples of valid XML resource file names include:

- MyResource.resx
- MyResource.de-de.resx
- My.Resource.File.resx

See Also

Non-dependent XML Resource Files | Dependent XML Resource Files | File Properties

Non-dependent XML Resource Files

A non-dependent XML resource file is defined as a .resx file whose parent object in the hierarchy is either a folder or the project root. The manifest resource name of this resource is computed as follows: the default namespace of the folder containing the XML resource file + the name of the XML resource file + **.resources**. For example, consider a project named Project1 with root namespace RootNamespace and the following hierarchy: Project1->Folder1->MyResource.de-de.resx.

For Visual Basic, where the default namespace for every folder is always the root namespace, the name `MyResource.resx` would be transformed to the manifest resource name `RootNamespace.MyResource.de-de.resources`.

For Visual C#, where the default namespace for every folder is always inferred from the folder's location in the hierarchy, the name MyResource.resx would be transformed to the manifest resource name `RootNamespace.Folder1.MyResource.de-de.resources`.

See Also

Dependent XML Resource Files

Dependent XML Resource Files

A dependent XML resource file is a file whose parent is another file.

In This Section

XML Resources Dependent Upon Invalid Code Files
Briefly describes the resource-naming scheme for this type of resource.
XML Resources Dependent Upon Valid Code Files
Briefly describes the resource-naming scheme for this type of resource.

Related Sections

XML Resources
Describes an XML resource file and the naming convention for such files.

XML Resources Dependent Upon Invalid Code Files

If an XML resource file is dependent upon a file for which a valid code model cannot be obtained, then the project system will follow the resource-naming scheme described in Non-dependent XML Resource Files.

See Also

XML Resources Dependent Upon Valid Code Files

XML Resources Dependent Upon Valid Code Files

If an XML file is dependent upon a valid language code file for the given project, and if a file code-model object can be obtained on the said file, then the manifest-resource name of the XML resource file shall be computed as follows: full name of the first class in the file that is the parent of the resource file + optional RFC 1766 culture info string (if supported by .NET Framework Resource Manager) + **.resources**.

If the parent of the XML resource file does not contain any class definitions, then the project system will fall back to the resource-naming scheme described in Non-dependent XML Resource Files.

For example:

- Consider the following project hierarchy where the full name of the first class in file Form1.vb is ACME.WidgetApp.MyClass: Project1->Form1.vb->Form1.resx. Then the manifest resource name for the XML resource file Form1.resx would be `ACME.WidgetApp.MyClass.resources`.
- Consider the same project, except now Form1.resx is named Form1.de.resx. Since **de** is an RFC 1766 culture string supported by the Resource Manager, it is automatically included in the name of the resource. Consequently, in this case, the manifest resource name would become `ACME.WidgetApp.MyClass.de.resources`.
- Consider the same project, except now Form1.resx is named Form1.de-ff.resx. Although **de-ff** is a valid culture info string as per RFC 1766, it is not a culture info string that is supported by the .NET Framework Resource Manager. Therefore, de-ff will not be included in the resulting manifest resource name, and the manifest resource name would be the same as in the first example.

See Also

XML Resources Dependent Upon Invalid Code Files

Non-XML Resources

A non-XML resource file is any file whose **BuildAction** property is **EmbeddedResource** and whose extension is something other than .resx. All such resource files follow the same naming convention regardless of whether they are dependent files.

The naming for such files is as follows:

base_filename+[**.***optional RFC 1766 culture info string*]**.***extension*

Aside from file-system limitations, there are no restrictions on *base_filename*, the *optional RFC 1766 culture info string*, or *extension*.

See Also

Files Without Embedded Culture Info Strings | Files With Embedded Culture Info Strings | File Properties

Files Without Embedded Culture Info Strings

The project system uses the following convention for computing manifest resource names for a non-XML resource file without an embedded culture info string: the default namespace of the folder containing the resource file + the base filename + extension of the resource file.

Consider a project Project1 with root namespace RootNamespace and the following hierarchy: Project1->Folder1->picture1.jpg.

For Visual Basic, where the default namespace for every folder is always the root namespace, the computed manifest resource name for picture1.jpg would be `RootNamespace.picture1.jpg`.

For Visual C#, where the default namespace for every folder is always inferred from the folder's location in the hierarchy, the computed manifest resource name for picture1.jpg would be `RootNamespace.Folder1.picture1.jpg`.

See Also

Files With Embedded Culture Info Strings | Non-XML Resources

Files With Embedded Culture Info Strings

The project system uses the following convention for computing manifest resource names for a non-XML resource file with an embedded culture info string.

If the culture info string is supported by the .NET Framework Resource Manager, the assembly manifest resource name is computed as follows: the default namespace of the folder containing the resource file + the base filename of the resource file + the extension of the resource file.

If the culture info string is not supported by the .NET Framework Resource Manager, the assembly manifest resource name is computed as described in Files Without Embedded Culture Info Strings.

Consider a project Project1 with root namespace RootNamespace and the following hierarchy: Project1->Folder1->picture1.de-de.jpg.

For Visual Basic, where the default namespace for every folder is always the root namespace, the computed manifest resource name for picture1.de-de.jpg would be `RootNamespace.picture1.jpg`.

For Visual C#, where the default namespace for every folder is always inferred from the folder's location in the hierarchy, the computed manifest resource name for picture1.de-de.jpg would be `RootNamespace.Folder1.picture1.jpg`.

Consider the situation in the previous example, except the file picture1.de-de.jpg is now named picture1.de-ff.jpg. Although **de-ff** is a valid culture info string as per RFC 1766, it is not a culture info string that is supported by the .NET Framework Resource Manager. Therefore, de-ff will be included in the computed assembly resource name. In this case, the assembly resource name would be `RootNamespace.picture1.de-ff.jpg` for Visual Basic and `RootNamespace.Folder1.picture1.de-ff.jpg` for Visual C#.

See Also

Files Without Embedded Culture Info Strings | Non-XML Resources

Configuring Applications Using Dynamic Properties

With dynamic properties you can store some of an application's property values to an external file and load them when the application runs. This allows you to make changes to property values without recompiling the application, after your project has been deployed.

In This Section

Introduction to Dynamic Properties
Provides an overview of storing and retrieving property values.
Setting Dynamic Properties
Provides instructions on how to set a property to be stored in an external file.
Changing the Value of Dynamic Properties
Describes how to alter the value of a dynamic property after deployment.
Removing Keys from Properties
Describes how to change a dynamic property to one that is stored within the compiled application.
Walkthrough: Storing and Retrieving Dynamic Properties
Describes the major steps in storing and retrieving property values.

Related Sections

Introduction to ADO.NET Connection Design Tools
Presents information on using dynamic properties with data features.
Deploying Applications and Components
Information about creating setup executables, packaging files, and publishing Web sites.
Saving the Properties of Components
Describes how to store and retrieve properties for custom controls.
Manageability
Discusses best practices for managing distributed applications.

Introduction to Dynamic Properties

Dynamic properties allow you to configure your application so that some or all of its property values are stored in an external configuration file rather than in the application's compiled code. By providing administrators with the means to update property values that may need to change over time, this can reduce the total cost of maintaining an application after the application has been deployed. For example, suppose you are building an application that uses a test database during the development process, and you need to switch it to a production database when you deploy it. If you store the property values inside the application, you have to manually change all of the database settings before you deploy, and then recompile the source code. If you store these values externally, you can make a single change in the external file and the application will pick up the new values the next time it runs.

> **Security Note** Property values stored in a configuration file are not secure. Sensitive data such as passwords and credit-card information should not be stored as dynamic properties.

You can use dynamic properties in any application that compiles to an .exe file. Projects that compile a DLL cannot use dynamic properties directly, however DLL properties can be set dynamically by the .exe referencing the .dll. Although the properties of any component, form, or control in these applications can be handled dynamically, some properties make better dynamic property candidates than others. Most often, you will store and retrieve properties that connect to external resources that might change, including databases, event logs, or performance counters. Many of these properties are identified as default candidates for dynamic properties. For more information, see "Default Dynamic Properties" below.

> **Note** The example code in this topic relies on the existence of a **SqlConnection** object and a configuration file; it will cause compilation errors without them. For more information on creating a **SqlConnection** object, see Walkthrough: Simple Data Access in a Windows Form. For more information on creating a configuration file, see Walkthrough: Storing and Retrieving Dynamic Properties.

Dynamic Properties and Configuration Files

When you set a property as configurable, its value is written to a configuration file and code is inserted in the class to indicate that the property value should be retrieved from this external resource. The configuration file varies based on the type of application; Web-based applications use the Web.config file, and Windows-based applications use a similar file with the extension .config. All of the forms and components in an application use a single configuration file. You cannot switch to a different configuration file or use multiple files within a single application.

Within the configuration file, properties are persisted via XML. For example, suppose you indicated that the **ConnectionString** property for a data connection should be stored in the configuration file. You would see this code in the Code Editor indicating that the value is stored externally:

```
' Visual Basic
Me.SqlConnection1.ConnectionString = CType(System.Configuration.ConfigurationSettings.AppSett
ings.GetValues("SqlConnection1.ConnectionString")

// C#
this.sqlConnection1.ConnectionString = ((string)( System.Configuration.ConfigurationSettings.
AppSettings.GetValues("SqlConnection1.ConnectionString"))));
```

> **Security Note** For more information about creating secure data connections, see Database Security.

In the configuration file, the value for this property would be stored via XML, using the key indicated in the form's code:

```
<configuration>
   <appSettings>
      <add key="sqlConnection1.ConnectionString" value="data source=myserver;initial catalog=
Apps;Integrated Security=SSPI;packet size=4096" />
   </appSettings>
</configuration>
```

Each value in the configuration file is assigned a key that is used to store and retrieve the value. The initial portion of the key indicates the component from which the value originated. For example, these two keys indicate a ConnectionString property on two different data connections within a single application:

```
<configuration>
```

```
  <appSettings>
    <add key="sqlConnection1.ConnectionString" value="data source=myserver;initial catalog=Ap
ps;Integrated Security=SSPI;packet size=4096" />
    <add key="sqlConnection2.ConnectionString" value="data source=myserver;initial catalog=Ap
ps;Integrated Security=SSPI;packet size=4096" />
  </appSettings>
</configuration>
```

You can modify the configuration file directly in order to dynamically update the property values in your application. The next time the application starts, it will update the values.

In a sense, configuration files are similar to resource files; both are used to store values outside the compiled application and both use XML to store their information. However, resource files serve a much different purpose than configuration files. Resource files are used to store strings and other localizable resources for translation purposes, while configuration files (in the case of dynamic properties) are used to update property values. The values in a resource file are meant to be translated but typically not changed, while the dynamic property values in a configuration file can be changed as needed. Also, you can associate multiple resource files with a project, but you can only have a single configuration file for your application. For more information on resource files, see Hierarchical Organization of Resources for Localization.

Setting Multiple Properties to the Same Key

Multiple properties can reference the same key-value pair in a configuration file. For example, if you have three components in your class that all access the same database, you can store the **ConnectionString** property for each in the same key-value pair. This means that if the database changes, you can update a single value in the settings file to apply the change to all three components. You would do this by setting each of the properties to the same key, like this:

```
' Visual Basic
Me.SqlConnection1.ConnectionString = CType(System.Configuration.ConfigurationSettings.AppSett
ings.GetValues("SqlConnection1.ConnectionString")
Me.SqlConnection2.ConnectionString = CType(System.Configuration.ConfigurationSettings.AppSett
ings.GetValues("SqlConnection1.ConnectionString")
Me.SqlConnection3.ConnectionString = CType(System.Configuration.ConfigurationSettings.AppSett
ings.GetValues("SqlConnection1.ConnectionString")

// C#
this.sqlConnection1.ConnectionString = ((string)(configurationAppSettings.GetValue System.Con
figuration.ConfigurationSettings.AppSettings.GetValues("SqlConnection1.ConnectionString"))));
this.sqlConnection2.ConnectionString = ((string)(configurationAppSettings.GetValue System.Con
figuration.ConfigurationSettings.AppSettings.GetValues("SqlConnection1.ConnectionString"))));
this.sqlConnection3.ConnectionString = ((string)(configurationAppSettings.GetValue System.Con
figuration.ConfigurationSettings.AppSettings.GetValues("SqlConnection1.ConnectionString"))));
```

Note that while the components in use differ, they all refer to the key for sqlConnection1.ConnectionString.

When you choose the same key for multiple properties, only one entry is made in the configuration file. The initial value stored is that from the first property that was given that key.

Data Types and Dynamic Properties

XML stores everything as a string. If the property value you are storing is not a string, you will see additional information in the Code Editor to indicate the data type for the property value. For example, suppose you are dynamically storing the value for a Timer component's **Interval** and **Enabled** properties. In the configuration file, this property would be stored like this:

```
<appSettings>
   <add key=timer1.Interval" value="100" />
   <add key=timer1.Enabled" value="True" />
</appSettings>
```

In the Code Editor, you would see this code to indicate that the retrieved value needs to be changed to a Double data type:

```
' Visual Basic
Me.Timer1.Enabled = System.Configuration.ConfigurationSettings.AppSettings.GetValues ("Timer1
.Enabled")
        Me.Timer1.Interval = System.Configuration.ConfigurationSettings.AppSettings.GetValues
```

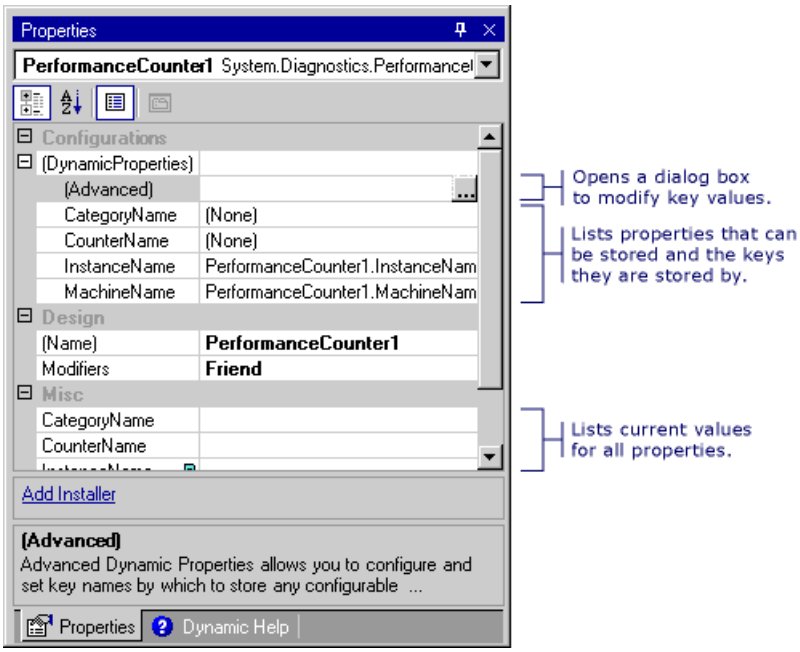
```
("Timer1.Interval")

// C#
this.timer1.Enabled = ((bool)(configurationAppSettings.GetValue("timer1.Enabled", typeof(bool
))));

this.timer1.Interval = ((System.Double)(configurationAppSettings.GetValue("timer1.Interval",
typeof(System.Double))));
```

Note There are two timers, one for Windows Forms and one for server-based applications, with slight differences in programming models. This example uses the server-based timer. For more information on the two timers available, see Introduction to Server-Based Timers.

Default Dynamic Properties

By default, the **DynamicProperties** node of the Properties window lists component properties that are likely to be stored in an external file. The following figure shows the way the Properties window arranges these items:

Dynamic Properties in the Properties Window



If you want to dynamically store a value for a property that is not listed in the Properties window, you can do so by selecting the ellipsis next to the Advanced listing in this section of the Properties window. This opens the **Dynamic Properties** dialog box, which lists all of the properties with an elementary data type (such as Boolean, string, float, double, integer, short, and so on) for the current component or form.

Note that if no properties are listed in the **DynamicProperties** node of the Properties window, this means that there are no properties that are configurable by default for that component. However, you can still add properties using the dialog box.

Performance and Security Concerns with Dynamic Properties

There are two items that may be of concern when you use dynamic properties. First, dynamic property storage and retrieval may affect your application's performance somewhat. Retrieving property values from within the compiled application is slightly faster than retrieving properties values from the configuration file. The first time a configuration file value is accessed by the application, there will be a slight performance impact as the application reads the file. This impact is very minor, however, and should not be noticeable. Any subsequent retrieval of that or other properties has a much smaller performance impact — similar to reading a value from a hash table.

Second, if you are storing property values such as user IDs and passwords to an external file, you will want to control who has access to the file and the values it contains. One way to accomplish this is to secure your file with a Windows *access control list* (ACL). An access control list specifies what operations different users can perform on specific files and directories. In addition, if you are working with a Web-based application, you can take advantage of the integrated security option provided by Windows, Internet Information Services (IIS), and SQL Server. In that model, a user's authentication credentials for the local network are also used to access database resources, and no explicit user name or password are used in the connection string.

For details about security, see your Windows documentation or the following topics:

- Best Practices for Securability
- .NET Framework Security
- Securing ASP.NET Web Applications
- Introduction to ADO.NET Connection Design Tools

See Also

Walkthrough: Storing and Retrieving Dynamic Properties | Configuring Applications Using Dynamic Properties | Setting Dynamic Properties | Removing Keys from Properties | Changing the Value of Dynamic Properties

Setting Dynamic Properties

You use the Dynamic Properties dialog box to indicate on a property-by-property basis those values you want to write to the configuration file. By default, this dialog box lists each property that Microsoft has identified as being a likely candidate for configuration; for most components, this will be a subset of the available properties.

For each property you want to store, you identify the key that the value should be stored under in the project's configuration file. For example, if you are storing the **ConnectionString** property for a data connection called oledbConnection1, you might set the key for this value to be oledbConnection1.ConnectionString. The settings file would then store the current value of the **ConnectionString** property under this uniquely identified key.

To set a property to be dynamically stored

1. Access the Properties window for the form, control, or component for which you want to set up dynamic properties.
2. Set the values of any properties you want to store in the configuration file.
3. Expand the **DynamicProperties** node.
4. If the property you want to store is listed beneath this node, click the **Ellipsis** ( ) button next to it and do the following:
   a. Click **Map property to a key in configuration file**.
   b. If you want to change the value under which the property is stored, modify the setting of the **Key** field.
5. If the property you want to store is not listed, click the **Ellipsis** ( ) button next to the **Advanced** property, and perform these steps in the dialog box that appears:
   a. In the **Properties** list box, select the check box next to the property whose value you want to store in an external file.
   b. In the **Key Mapping** drop-down box, either accept the default key or type a new key value by which you want the property's value to be stored.
   c. Click **OK**.
6. Repeat step 4 or 5 for each property for which you want to store a value.
7. Click **OK** to close the dialog box.

   All of the properties you chose to store dynamically are now listed in the **DynamicProperties** node of the Properties window.

See Also

Introduction to Dynamic Properties | Removing Keys from Properties | Changing the Value of Dynamic Properties | Walkthrough: Storing and Retrieving Dynamic Properties

Changing the Value of Dynamic Properties

You change the value of a dynamic property after deployment by modifying the configuration file in which the property value was stored. You can modify the configuration file by double-clicking it in Solution Explorer, or by opening it in any editor from the application's deployed location. Depending on whether you are working with a Windows application or a Web application, your configuration file will be named differently:

- If you are using a Windows application, the design-time configuration file will be named app.config. After your project is compiled and deployed, the file will be copied to the bin directory and renamed to *assembly.config*, where *assembly* is the name of your project's assembly.
- If you are using a Web application, the information about dynamic properties is stored in AppSettings section of the Web.config file.

After making the change to the configuration file, the new property value will be incorporated the next time you start your application.

For example, suppose you have an instance of the **EventLog** component that you have connected to a custom log called Orders, and now you want to switch it to a different log called ProcessedOrders. The component instance's **Log** property was dynamically stored in the .config file, which would currently indicate the following:

```
<add key="eventLog1.Log" value="Orders" />
```

To change this, you would pull up the configuration file and change the value as follows:

```
<add key="eventLog1.Log" value="ProcessedOrders" />
```

To make a change to the value of a dynamic property after deploying your application

1. In an editor, open the configuration file for your application.
2. Locate the line that contains the property and value you want to modify. The line should look like this:

   ```
   <add key="Component.Property" value="currentvalue" />
   ```

   where *Component* is the name of the control or other element on which the property occurs, and *Property* is the name of the property whose value is being stored. *currentvalue* is the stored value for the property.
3. Modify the *currentvalue* portion of the statement to the new value you want to use.
4. Save and close the file.
5. Restart your deployed application.

   The new value will be updated automatically.

   > **Note** If you are working in a Web application, the application will automatically restart as soon as you modify the configuration file.

See Also

Introduction to Dynamic Properties | Setting Dynamic Properties | Removing Keys from Properties | Walkthrough: Storing and Retrieving Dynamic Properties

Removing Keys from Properties

You remove a key from a property when you no longer want to store that property's value in an external file. Removing a key removes the code indicating that the property value is stored in a configuration file.

Note that the system does not automatically update the configuration file when you remove a key from a property — the entry still appears in the configuration file. This is because the configuration file does not keep track of references to a key; because multiple properties may use the same key, the configuration file has no way of knowing when a key is no longer in use. Therefore, there may be many more keys in your configuration file than are actually in use in the product. This should not cause problems for your application.

To remove a key from a property

1. Access the Properties window for the form, control, or component.
2. Expand the **DynamicProperties** node. All properties that currently have keys are listed beneath this section.
3. Locate the property whose key you want to remove, then click the **Ellipsis** ( ) button next to it.
4. In the dialog box that appears, uncheck the **Map property to a key in configuration file** check box.

   When you return to the Properties window, that property will now show a value of **None** in the **DynamicProperties** node, indicating that its value is no longer being stored.

   > **Note** If you want, you can edit the configuration file to remove the entry for this property; however, leaving the property key in the file will not cause any errors in your application.

See Also

Introduction to Dynamic Properties | Setting Dynamic Properties | Changing the Value of Dynamic Properties | Walkthrough: Storing and Retrieving Dynamic Properties

Walkthrough: Storing and Retrieving Dynamic Properties

The procedures in this topic demonstrate the process of creating a project, storing some of its property settings in an external file, and then modifying the stored values and dynamically updating them in the deployed application.

In this application, you will create an instance of the Event Log component and store its property values to a file.

To create your component instance

1. On the **File** menu, point to **New**, and then click **Project**.
2. In the **New Project** dialog box, choose Visual Basic or Visual C# in the left pane, then choose the **Windows Application** template. Name it **MyLog**.
3. From the **Components** tab of the Toolbox, drag an **EventLog** component to the form.

   > **Note** Dynamic properties are not associated in any way with the **EventLog** component; you are simply using this component in this walkthrough as an example of the type of components whose properties you can store externally.

4. In the Properties window for the **EventLog** component, set the following values:
   - Set the **Log** property to **Application** so that your component can connect to a valid Windows event log.
   - Set the **MachineName** property to the name of the server on which the event log resides. Use a period (**.**) for the local machine.
   - Set the **Source** property to the string of your choice. In this case, you can use the name of your project.
5. On the **View** menu, click **Code** to open the Code Editor for Form1.
6. Expand the node titled **Windows Forms Designer generated code**.

   You should see a line showing the property value for the **Log** property, as follows:

   ```
   ' Visual Basic
   Me.EventLog1.Log = "Application"

   // C#
   this.eventLog1.Log = "Application";
   ```

7. Save your files.

   > **Note** For more information on creating **EventLog** component instances and setting their properties, see Logging Application, Server, and Security Events.

To display the log property value

1. In the designer, drag a label and a button from the **Windows Forms** tab of the Toolbox to Form1.
2. Double-click the button to open the Code Editor and access the **Button** control's **Click** event handler.
3. Add this code to the method:

   ```
   ' Visual Basic
   Label1.Text = "You are connected to the " & EventLog1.Log & " Log."

   // C#
   label1.Text = "You are connected to the " + eventLog1.Log + " Log.";
   ```

To configure the Log property for your component

1. In the Properties window for the **EventLog** component, click the plus sign (+) next to **Dynamic Properties**.
2. Select the **Ellipsis** ( ) button next to the **Log** property.
3. In the **Dynamic Property** dialog box that appears, select the **Map property to a key in configuration file** check box, then click **OK**.

   This indicates that the **Log** property value should be stored using the default key — the class name and property name. In

this case, the key will be **EventLog1.Log** or **eventLog1.Log**, depending on the programming language you are using.

4. Access the Code Editor for Form1, and find the **InitializeComponent** procedure for **MyLog**.

   You should see that code has automatically been changed to reflect that the value is now stored externally:

```
' Visual Basic
Me.EventLog1.Log = _
   CType(configurationAppSettings.GetValue("EventLog1.Log", _
   GetType(System.String)), String)
Me.EventLog1.Source = "MyLog"

// C#
this.eventLog1.Log =
   ((string)(configurationAppSettings.GetValue("eventLog1.Log",
   typeof(string))));
this.eventLog1.Source = "MyLog";
```

5. Save your files and build the project.
6. In Solution Explorer, double-click the .config file for your project to view it in the editor. You should be able to locate the entry for the **EventLog1.Log** property value:

```
' Visual Basic
<add key="EventLog1.Log" value="Application" />

// C#
<add key="eventLog1.Log" value="Application" />
```

7. Run your application. When you click the button, the label should say that you are connected to the Application log.
8. Close your application.

You also have the ability to modify the saved values outside of running the application. You might want to do this to set the initial values for an application.

To modify the stored value of the project .config file

1. In Notepad, open the .config file that is stored in the project folder.
2. Locate the **EventLog1.Log** entry you looked at in the previous procedure.
3. Change the value in that line to **System** or **Security**, and then save and close the file.
4. Press **F5** to run the application. When you click the button, the new value should be reflected.

The configuration file in the project folder is named App.config. When you build the project, it is copied to the output folder and renamed to *Assemblyname*.config, where *assemblyname* is the project output, for example, MyLibrary.dll or MyApplication.exe. You need to modify this copy of the .config file to affect a deployed application. For more information, see Changing the Value of Dynamic Properties.

To modify the stored value of the deployed .config file

1. In Notepad, open the .config file for your application from its built location (often the \bin or \obj\debug folder of the project folder).
2. Locate the **EventLog1.Log** entry you looked at in the previous procedure.
3. Change the value in that line to **System** or **Security**, and then save and close the file.
4. Run the application from its built location. This is the same folder where you modified the .config file in step 1. When you click the button, the new value should be reflected.

See Also

Introduction to Dynamic Properties | Configuring Applications Using Dynamic Properties

Creating Applications

This section contains information on creating Windows-based applications or Web-based applications, adding data access to your application, and creating components. In addition, it covers additional features you add to your basic applications, such as security, and things you do, such as troubleshooting problems, once your application has taken shape.

In This Section

Windows Forms and Web Forms Recommendations
Discusses considerations for choosing between the two forms packages.
Event Handlers in Visual Basic and Visual C#
Provides links to discussions of various aspects of events: event handling, delegates, differences between Windows- and Web-based event handling, differences between Visual Basic and Visual C#.
Creating Web Applications and Services
Discusses the Web programming spectrum, from creating HTML pages to developing business functionality that you expose via Internet protocols.
Creating Windows Applications Portal
Provides links to Visual Studio .NET topics about how to create traditional Windows-based applications.
.NET Framework Class Library in Visual Studio
Provides links to Visual Studio .NET topics that support the development of features based on classes in the .NET Framework.
Globalizing and Localizing Applications
Discusses the steps needed to make your application useful for an international audience.
Securing Applications Portal
Discusses the types of security problems that can arise and how to deal with them.
Troubleshooting Problems Portal
Contains links to topics on troubleshooting various types of applications.
Testing Scalability and Performance Portal
Contains links to discussions of design choices and testing the results of those choices.

Related Sections

Common Tasks in Visual Basic and Visual C#
Lists the most common programming tasks in various features areas (such as Web applications and debugging) and provides links to procedural information.
Decision Chart
Provides an interactive guide to help you choose which approach or technology to use, as well as links to more information.
Visual Basic Code Example Topics
Lists quickstart-like code examples that help you perform common tasks in Visual Basic .NET.
Accessing Data
Discusses aspects of incorporating data access functionality into your applications.
Programming with Components
Describes components of all kinds, including those you use on servers.
Managing Projects
Describes the project system for managing your applications as you write them.
Building, Debugging, and Testing
Information about breakpoints, handling exceptions, edit and continue, as well as fine-tuning your application.
Deploying Applications and Components
Provides information about creating setup executables, packaging files, and publishing Web sites.
Planning World-Ready Applications
Gives links to topics with guidance on planning and developing world-ready applications.
Programming Basics Namespaces in Visual Studio
Lists the major namespaces for collections, character encodings, file I/O, and threading. Also provides links to .NET Framework reference topics about the types (classes, structures, enumerations, delegates, and interfaces) in the namespaces, as well as links to topics about using them in Visual Basic and Visual C# applications.
Programming with Office
Provides information on using Microsoft Office and Visual Studio .NET as part of your business application.

Windows Forms and Web Forms Recommendations

When designing applications that involve a user interface, you have two choices: Windows Forms and Web Forms. Both have full design-time support within the development environment and can provide a rich user interface and advanced application functionality to solve business problems. How, then, to decide which technology is appropriate for a given application?

Certain application considerations might make the choice obvious: For example, if you are creating an e-commerce Web site that will be accessible to the public over the Internet, then of course you would develop the application using Web Forms pages. If you are building a processing-intensive, highly responsive application that needs to take advantage of the full functionality of the client machine — such as an office productivity application — of course you would use Windows Forms. However, in other cases the choice might not be so clear.

In the sections that follow, you can find information about the features and characteristics of each type of technology that will help you determine which is most suited to your application.

Windows Forms

Windows Forms are used to develop applications where the client is expected to shoulder a significant amount of the processing burden in an application. These include classic Win32 desktop applications, the kinds that were traditionally developed in previous versions of Visual Basic and Visual C++. Examples include drawing or graphics applications, data-entry systems, point-of-sale systems, and games.

A common feature these applications share is that they rely on the power of the desktop computer for processing and high-performance content display. Some Windows Forms applications might be entirely self-contained and perform all application processing on the user's computer. Games are often written this way. Others might be part of a larger system and use the desktop computer primarily for processing user input. For example, a point-of-sale system often requires a responsive, sophisticated user interface that is created on the desktop computer, but is linked to other components that perform back-end processing.

Because a Windows application that uses Windows Forms is built around a Windows framework, it has access to system resources on the client computer, including local files, the Windows registry, the printer, and so on. This level of access can be restricted to eliminate any security risks or potential problems that arise from unwanted access. Additionally, Windows Forms can take advantage of the .NET GDI+ graphics classes to create a graphically rich interface, which is often a requirement for data-mining or game applications.

Web Forms

ASP.NET Web Forms are used to create applications in which the primary user interface is a browser. Naturally, this includes applications intended to be available publicly via the World Wide Web, such as e-commerce applications. But Web Forms are useful for more than just creating Web sites — many other applications lend themselves to a "thin front end" as well, such as an intranet-based employee handbook or benefits application. An important feature is that there is no distribution cost, since users already have installed the only piece of the application that they need— the browser.

Web Forms applications are, by definition, platform-independent — that is, they are "reach" applications. Users can interact with your application regardless of what type of browser they have and even what type of computer they are using. At the same time, Web Forms applications can be optimized to take advantage of features built into the most recent browsers such as Microsoft Internet Explorer 5 to enhance performance and responsiveness. (In many cases, this optimization is built into the Web Forms components you are using, which can automatically detect browser levels and render pages accordingly.)

Web Forms offer some features that are useful even in non-Web contexts. Because they rely on HTML, they are suitable for text-intensive applications of any sort, and especially those in which text formatting is important. Because browsers are usually limited in their access to a user's system resources, they are useful precisely in situations where you want to limit users' access to portions of your application.

Comparing Windows Forms and Web Forms

The following table provides a side-by-side comparison of different application criteria and how Windows Forms and Web Forms technologies address these criteria.

Feature/Criterion	Windows Forms	Web Forms

Deployment	Windows Forms allow "no-touch" deployment, where applications can be downloaded, installed, and run directly on the users' machines without any alteration of the registry.	Web Forms have no client deployment; the client requires only a browser. The server must be running the Microsoft .NET Framework. Updates to the application are made by updating code on the server.
Graphics	Windows Forms include GDI+, which allows sophisticated graphics to be used for games and other extremely rich graphical environments.	Interactive or dynamic graphics require round trips to the server for updates when used on Web Forms. GDI+ can be used on the server to create custom graphics.
Responsiveness	Windows Forms can run entirely on the client computer; they can provide the quickest response speed for applications requiring a high degree of interactivity.	If you know that users will have Internet Explorer 5 or later, a Web Forms application can take advantage of the browser's dynamic HTML (DHTML) capabilities to create a rich, responsive user interface (UI). If users have other browsers, most processing (including UI-related tasks such as validation) requires a round trip to the Web server, which can affect responsiveness.
Form and text flow control	Windows Forms grid positioning gives you precise two-dimensional control (x and y coordinates) over the placement of controls. To display text on Windows Forms requires that you insert it into controls (for example, the **Label** control, the **Textbox** control, or the **RichTextBox** control). Formatting is limited.	Web Forms are based around HTML-style flow layout and therefore support all the features of Web page layout. They are particularly rich in text formatting support. Control layout can be managed adequately (with some limitations, such as no overlapping controls). If the users have DHTML-capable browsers, you can specify much more precise layout with two-dimensional (x- and y-coordinate) layout.
Platform	Windows Forms require the .NET Framework running on the client computer.	Web Forms require only a browser. DHTML-capable browsers can take advantage of extra features, but Web Forms can be designed to work with all browsers. The Web server must be running the .NET Framework.
Access to local resources (file system, Windows registry, and so forth)	Applications, when permitted, can have complete access to local computer resources. If required, the application can be restricted with precision from using specific resources.	Browser security prevents the application from accessing resources on the local computer.
Programming model	Windows Forms are based on a client-side, Win32 message-pump mode, where instances of components are created, used, and discarded by the developer.	Web Forms rely on a largely asynchronous, disconnected model, where components are loosely coupled to the application front end. Typically, application components are invoked via HTTP. This model may not be suitable for applications requiring extreme throughput from the user end or for those with high-volume transactions. Similarly, Web Forms applications may not be suitable for database applications that require high levels of concurrency control (for example, pessimistic locking).
Security	Windows Forms use granular permissions in its implementation of code access security to protect computer resources and sensitive information. This allows careful exposure of functionality, while retaining security. For instance the Printing Permission, which at one level would allow printing to the default printer only, at another level would allow printing to any printer.	Authorization to gain access to the resources of a web application is typically controlled on a per-URL basis by authenticating the credentials (for example, a name/password pair) of the requestor. Web Forms enable the ability to control the identity under which server application code is executed. Applications can execute code with the identity of the requesting entity, which is known as impersonation. Applications can also dynamically tailor content based on the requestor's identity or role. For example, a manager could receive access to a site, or a higher level of content than someone with lower permissions.

See Also

What's New In Visual Basic and Visual C# | Introduction to Web Forms Pages | Introduction to Windows Forms | What's New in Windows Forms and Controls | What's New in Web Development

Event Handlers in Visual Basic and Visual C#

You can use events to notify objects when interesting things happen. You can associate methods with particular events; these methods are called *event handlers*. When an event is raised, the event handler(s) that receive the event are executed. Events can be assigned to multiple handlers, and the methods that handle particular events can be changed dynamically.

There are subtle differences in the way that Visual Basic and Visual C# implement events and event handling. Those aspects that are similar for both languages are discussed in the same topic. Those aspects that different are discussed in language-specific topics.

In the Visual Basic and Visual C# Documentation

Event Handling in Windows Forms
Provides links to a variety of topics on handling events in Windows Forms for Visual Basic and Visual C#.
Server Event Handling in Web Forms Pages
Provides links to several topics discussing handling server events in Web Forms pages for Visual Basic and Visual C#.

In the Visual C# Reference Documentation

Events Tutorial
Demonstrates how to declare, raise, and handle events using Visual C#.
Events Sample
Provides sample code demonstrating event handling with Visual C#.

In the Visual Basic Reference Documentation

Events and Delegates
Provides links to Visual Basic-specific topics on using events.
Event Implementation for Components and Custom Controls with Visual Basic
Discusses implementing events and event handlers in custom controls and components with Visual Basic.

Additional Information

Handling and Raising Events
Provides links to topics on handling and raising events using the .NET Framework.
Events in ASP.NET Server Controls
Describes how to raise events from custom ASP.NET server controls.
Events and Delegates
Provides an overview of the event model and describes delegates in the .NET Framework.

Creating Web Applications and Services

An important part of Visual Studio .NET is the ability to create distributed applications based around the Web. Visual Studio .NET allows you to create the application user interface using Web Forms pages and to create components using XML Web services.

Web Forms is the ASP.NET technology that allows you to create a user interface for Web-based applications, whether customers are accessing your application from a traditional Web browser or a mobile device. Using Web Forms pages, you can create a browser-neutral UI that does its processing on the Web server, freeing you of the need to create browser-specific (or device-specific) versions of your user interface.

XML Web services are components that run on the server and typically include business logic. Like traditional components, they encapsulate specific functionality and can be called from different programs. However, they are available via Web protocols, making them compatible with programs running in different languages, on different computers, and even on different operating systems.

> **Note** You can also use Windows Forms as the user interface for a distributed ASP.NET Web application. For more information, see Windows Forms as the Presentation Tier of a Three-Tier Application.

In This Section

Introduction to ASP.NET Web Applications in Visual Studio
: Discusses the components that make up an ASP.NET Web application and how they various technologies involved work together.

Code for Web Applications
: Lists code examples for common Web tasks.

Web Walkthroughs
: Links to all the topics that give step-by-step demonstrations of Web programming in Visual Basic and Visual C#.

Security Considerations for ASP.NET Web Applications
: Links to the topics that provide an introduction to security in Web applications.

Introduction to Web Application Debugging
: Provides a high-level overview of using the Visual Studio for Web applications, including configuration and using the debugger with client script.

Web Forms Pages
: Provides links to information about the technologies and tools for creating the browser-based user interface for ASP.NET Web applications.

ASP.NET Server Controls
: Provides links to information about the controls designed specifically to work with Web Forms.

XML Web Services in Managed Code
: Covers the technologies for creating, distributing, and consuming XML Web services using Visual Basic or Visual C#.

Related Sections

Windows Forms and Web Forms Recommendations
: Discusses considerations for choosing between the two forms packages.

Visual Basic Code Example Topics
: Lists quickstart-like code examples that help you perform common tasks in Visual Basic .NET.

Decision Chart
: Provides an interactive guide to help you choose which approach or technology to use, as well as links to more information.

Walkthrough: Creating an XML Web Service Using Visual Basic or Visual C#
: Provides step-by-step instructions for how to create an XML Web service that covers Fahrenheit to Celsius.

Walkthrough: Accessing an XML Web Service Using Visual Basic or Visual C#
: Provides step-by-step instructions for how to discover and make use of an existing XML Web service.

Walkthrough: Creating a Distributed Application
: Provides step-by-step instructions for creating a multitier, distributed application that has both a rich Windows client and a browser interface.

Programming the Web with XML Web Services
: Describes what XML Web services are.

Web Namespaces in Visual Studio
: Lists the major namespaces for Web Forms pages and XML Web services and provides links to .NET Framework reference topics about the types (classes, structures, enumerations, delegates, and interfaces) in the namespaces, as well as links to topics about using them in Visual Basic and Visual C# applications.

Introduction to Web Projects

Provides information on the file management tasks associated with Web projects.

Creating Windows Applications

Provides links to information about traditional Windows applications.

Getting Started With Enterprise Templates in Distributed Applications

Introduces enterprise templates, which reduce the complexity and costs associated with development of distributed applications.

Enterprise Samples

Introduces two sample applications that demonstrate .NET technology in the enterprise: the Duwamish Books sample and the Fitch and Mather sample.

Debugging XML Web Services in Managed Code

Provides instructions on multiple methods for debugging an XML Web service.

Deploying XML Web Services in Managed Code

Provides instructions on various ways to deploy an XML Web service.

Debugging ASP.NET Web Applications

Provides instructions and prerequisites on debugging an ASP.NET application either at development time or when the application is already deployed and running.

Deployment of a Web Setup Project

Describes methods for deploying files and applications to a Web server.

Introduction to ASP.NET Web Applications in Visual Studio

Visual Studio .NET allows you to create applications that leverage the power of the World Wide Web. This includes everything from a traditional Web site that serves HTML pages, to fully featured business applications that run on an intranet or the Internet, to sophisticated business-to-business applications providing Web-based components that can exchange data using XML.

This topic provides an overview of the types of Web applications you can create in Visual Studio, how they relate to one another and to other Visual Studio technologies, and suggestions about where to learn more about the types of Web applications you want to create.

> **Note** For details about the relative advantages of Web applications and Windows applications, see Windows Forms and Web Forms Recommendations.

Visual Studio ASP.NET Web Applications

A Visual Studio Web application is built around ASP.NET. ASP.NET is a platform — including design-time objects and controls and a run-time execution context — for developing and running applications on a Web server.

ASP.NET in turn is part of the .NET Framework, so that it provides access to all of the features of that framework. For example, you can create ASP.NET Web applications using any .NET programming language (Visual Basic, C#, Managed Extensions for C++, and many others) and .NET debugging facilities. You access data using ADO.NET. Similarly, you can access operating system services using .NET Framework classes, and so on.

ASP.NET Web applications run on a Web server configured with Microsoft Internet Information Services (IIS). However, you do not need to work directly with IIS. You can program IIS facilities using ASP.NET classes, and Visual Studio handles file management tasks such as creating IIS applications when needed and providing ways for you to deploy your Web applications to IIS.

Where Does Visual Studio Fit In?

As with any .NET application, if you have the .NET Framework, you can create ASP.NET applications using text editors, a command-line compiler, and other simple tools. You can copy your files manually to IIS to deploy the application.

Alternatively, you can use Visual Studio. When you use Visual Studio to create Web applications, you are creating essentially the same application that you could create by hand. That is, Visual Studio does not create a different kind of Web application; the end result is still an ASP.NET Web application.

The advantage of using Visual Studio is that it provides tools that make application development much faster, easier, and more reliable. These tools include:

- Visual designers for Web pages with drag-and-drop controls and code (HTML) views with syntax checking.
- Code-aware editors that include statement completion, syntax checking, and other IntelliSense features.
- Integrated compilation and debugging.
- Project management facilities for creating and managing application files, including deployment to local or remote servers.

If you have used Visual Studio before, these kinds of features will seem familiar, because they are similar to features that you get for creating applications in previous versions of Visual Basic and Visual C++. With Visual Studio .NET you can use these kind of features to create ASP.NET Web applications.

Elements of ASP.NET Web Applications

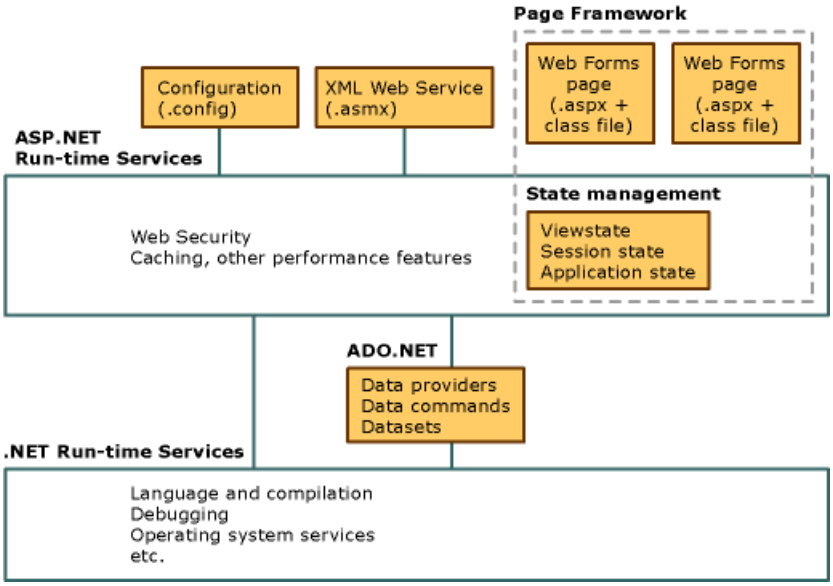
Creating ASP.NET Web applications involves working with many of the same elements you use in any desktop or client-server application. These include:

- **Project management features** When creating an ASP.NET Web application, you need to keep track of the files you need, which ones need to be compiled, and which need to be deployed.
- **User interface** Your application typically presents information to users; in an ASP.NET Web application, the user interface is presented in Web Forms pages, which send output to a browser. Optionally, you can create output tailored for mobile devices or other Web appliances.
- **Components** Many applications include reusable elements containing code to perform specific tasks. In Web applications, you can create these components as XML Web services, which makes them callable across the Web from a Web application, another XML Web service, or a Windows Form, for example.
- **Data** Most applications require some form of data access. In ASP.NET Web applications, you can use ADO.NET, the data

services that are part of the .NET Framework.

- **Security, performance, and other infrastructure features** As in any application, you must implement security to prevent unauthorized use, test and debug the application, tune its performance, and perform other tasks not directly related to the application's primary function.

The following diagram provides an overview of how the pieces of ASP.NET Web applications fit together and fit into the broader context of the .NET Framework.



The sections below outline the different elements of an ASP.NET Web application and provide links to more information about how to work with each element in Visual Studio.

Overview: ASP.NET

The information in this topic provides only an introduction to ASP.NET. For details about what ASP.NET is and what components are involved in an ASP.NET application, see Introduction to ASP.NET.

Project Management: ASP.NET Web Application Projects and Deployment

To work with an ASP.NET Web application using Visual Basic or Visual C#, you use the ASP.NET Web Application project template. As with other Visual Studio projects, a Web application project is a central repository for all the information required to design, run, and manage the application.

When you create a Web application project, Visual Studio creates the necessary files and folders on the server, sets the appropriate security settings on them, and creates the IIS application. For details, see ASP.NET Web Application Template.

Each Web application can maintain a Web.config file that follows the format and conventions of .NET configuration files. In ASP.NET, the configuration file allows you to establish project-specific settings for security, compilation options, tracing, error handling, and more.

For details about how to create and manage ASP.NET Web application projects in Visual Studio, see the following topics.

- Working with Web Projects Provides links to topics that describe Web application projects in Visual Studio.
- Web Forms Pages and Projects in Visual Studio Provides details about the types of files created in a Web Application project.
- Deployment of a Web Setup Project Provides an introduction to deploying Web applications.
- ASP.NET Configuration Describes the configuration file system and provides links to topics about the contents of the Web.config file.

User Interface: Web Forms and the ASP.NET Page Framework

You can create HTML pages and Web Forms pages as the user interface for your application. HTML pages are generally used for static content. In contrast, Web Forms pages give you a programmable interface that works much like a Visual Basic form, except that the user interface is rendered in a Web browser or other Web device.

> **Note** You can use a Windows Form as the user interface in an application that calls XML Web services. For an example, see Walkthrough: Creating a Distributed Application.

Web Forms pages are built on the ASP.NET Page framework. Each Web Forms page is an object that derives from the ASP.NET Page class, which acts as a container for controls. The page actually consists of two files: an .aspx file that contains the UI elements (static HTML text and control elements) and a class file that contains the code that runs the page. For details, see Introduction to Web Forms Pages.

When users request a Web Forms page, the Page framework runs the Web Forms page object and all the individual controls on it. The output of the **Page** class and of the controls is HTML. For details, see Web Forms Page Processing.

Because the Web is inherently stateless — by default, components in a Web application are alive only long enough to process a single request — Web applications face challenges in preserving values as the user works with the application. To help, the Page framework includes facilities for managing state. These include page-based "view state" (a method for preserving values in controls) and access to non-page-based state facilities such as Session state (user-specific) and Application state (global to the application). For details, see Web Forms State Management.

The Page framework supports a set of controls you can use to program user interaction with your Web Forms pages. User actions in the form are captured and processed by the Page framework in a way that lets you treat them as standard events. You can choose from a large selection of controls available in Visual Studio. In addition, you can create your own custom controls. For details, see Introduction to Web User Controls and Developing ASP.NET Server Controls.

> **Note** You can also create output for mobile devices. To do so, you use the same ASP.NET page framework, but you create Mobile Web Forms instead of Web Forms pages and use controls specifically designed for mobile devices. For details, see Creating ASP.NET Mobile Wep Applications.

For details about Web Forms pages and how to work with them in Visual Studio, see the following topics:

- Web Forms Pages Lists topics that provide an overview of Web Forms, including background on ASP.NET and on the Web Forms code model.
- Creating and Managing Web Forms Pages Provides information on how to create, add, and manage Web Forms pages.
- Programming Web Forms Provides information on the specific aspects of programming in a Web Forms environment.

Web-based Components: XML Web Services

An XML Web service is a component that can be called over a TCP/IP network by other applications. It performs a specific function — anything from calculations and credit card validation to complex order processing— and returns values to the calling application.

What makes XML Web services unique is that they can be called across the Web. XML Web services are invoked using HTTP or SOAP requests and exchange data with other components using XML. As such, they can form an integral part of ASP.NET Web applications, providing services not only to your applications, but also to any application that has Web access, making them ideal for business-to-business transactions.

For information about XML Web Services, how they work with ASP.NET applications, and how you can create them in Visual Studio, see Programming the Web with XML Web Services.

Web Application Data Access

Most ASP.NET Web applications involve at least some level of access to data. ASP.NET does not directly include data access facilities. Instead, Web applications use ADO.NET data services.

ADO.NET provides a complete framework for accessing and managing data from a variety of sources, including databases and XML files or streams. ADO.NET includes *providers* — classes that allow you to connect to data sources, execute commands, and read results. You can optionally keep data in a *dataset*, which is a disconnected, in-memory cache.

Data access in Web applications, whether in a Web Forms page or an XML Web service, introduces special challenges:

- **Statelessness** The components of Web applications usually do not preserve state, which makes it impractical to maintain live connections to a data source (or other resources).
- **Scalability** Because Web applications can have user loads that vary substantially over short periods of time, data access has to be designed with scalability in mind.

Visual Studio provides a variety of tools for working with data, including Toolbox support for data elements, various configuration wizards, and so on. The following topics provide information on data in general — that is, ADO.NET — as well as about using data in Web applications.

- Introduction to Data Access with ADO.NET Provides general information on the design of ADO.NET and some implications of the ADO.NET model for application development.

- Introduction to Data Access in Web Forms Pages Provides an overview of data access in Web applications, including information about some of the common issues associated with Web application data access.

Web Application Infrastructure: Security, Performance, and More

In addition to giving you the means to create user interface elements and callable components, ASP.NET provides a context for those elements to run in. For example, ASP.NET communicates with IIS to handle requests for Web Forms pages and XML Web services, parse the files, call related components, and so on.

Much of this work occurs beneath the surface, at a level where you do not typically need to program when you are creating desktop applications.

> **Note** Visual Studio provides limited support for working with ASP.NET infrastructure features. For example, you can edit the Web application configuration file (Web.config) using the Visual Studio Code Editor. But the .NET Framework is both pluggable and extensible, providing you with low-level access if necessary.

There are other aspects of Web applications that you must often contend with that are part of the application's infrastructure. These include:

- **Security** You frequently have to authenticate and authorize users of your Web application. There are special problems associated with security in a Web application, because users are getting access to server-based resources and because you have very little control over the client side of the application (the browser or mobile device). ASP.NET includes security features that you can configure and program against in your Web application.
- **Performance and optimization** You can tune your application's performance by caching pages and data. ASP.NET maintains an output cache that stores pages that have been requested before; by specifying caching settings, you can control how long pages are cached and under what circumstances they are refreshed.
- **Tracing** Because Web applications run on the server — often a remote server — they do not have output other than the application output (a Web Forms page, for example). ASP.NET therefore gives you the opportunity to include trace information directly in a Web Forms page. For details about adding tracing to ASP.NET Web applications, with links to topics that address specific trace procedures, see ASP.NET Trace.

For details about Web Forms **security**, see the following topics:

- Introduction to Web Application Security Describes the issues associated with security in Web Forms.
- Overview of Web Application Security Threats Provides a high-level discussion of the types of security vulnerability that Web applications are subject to.
- Basic Security Practices for Web Applications Provides a list of security recommendations that apply to even the simplest Web application.
- ASP.NET Web Application Security Provides an overview of security in ASP.NET, with links to a variety of topics about ASP.NET security architecture, security scenarios, and so on.
- Access Permissions for Web Applications Provides an introduction to the security context in which ASP.NET applications run.
- ASP.NET Configuration Explains a Web application configuration file system, which is where you make many security settings.

For details about Web Forms **performance and optimization**, see the following topics:

- ASP.NET Caching Features Provides an overview of the ASP.NET output cache facility for tuning performance, and links to caching procedures.
- ASP.NET Configuration Explains a Web application configuration file system, which is where you make many performance settings.
- Application Center Web site (http://www.microsoft.com/applicationcenter/) Provides information about testing Web application performance.

See Also

Web Forms Pages | XML Web Services in Managed Code | ASP.NET Technology Backgrounder | Web Walkthroughs

Code for Web Applications

This topic is designed to help you find code that demonstrates how to perform common programming tasks in Web applications using Visual Basic .NET.

Note For tasks that pertain to Web Forms controls specifically, see Code for Web Forms Controls.

The following categories are represented in the list below:

• Page Programming • Event Handling • Accessing Data • Managing State	• Error Handling • Security • XML Web Services • Related Code Examples

You can also find sample code for Web applications in the following locations:

- Web Applications Example Topics
- Web Forms Controls Example Topics
- Visual Basic Sample Applications
- Visual Basic QuickStarts and Other Samples
- .NET Framework Reference Information
- XML Web Services Created Using ASP.NET and XML Web Service Clients
- Creating ASP.NET Mobile Web Applications

Page Programming

Example of...

Creating a simple Web Forms page

Redirect users to another page

Passing information from one page to another

Detecting what browser the user has

Reading the query string passed in the URL

Event Handling

Example of...

Basic event handling for Web server controls

Creating event handlers at run time

Binding an existing method to an event

Handling an event in both client script and server code

Handling multiple events with the same event handler

Handling an event globally

Accessing Data

Example of...

Simple data access

Accessing read-only data

Caching datasets between round trips

Data-binding individual controls

Updating a database

Using a grid to read and write data

Displaying paged data

Defining a data source dynamically

Data-binding a list control (ListBox, DropDownList, etc)

Data-binding HTML server controls using HTML syntax

Data-binding controls at run time

Managing State

Example of...

Saving information in viewstate in the page

Sharing values in Session state

Saving a dataset in Session state

Error Handling

Example of...

Basic error handling

Creating a global error handler

Security

Example of...

Using HTML encoding to guard against malicious data

Filtering HTML elements out of user input

Allowing only predetermined characters in user input

Guarding against unhandled exceptions

Passing credentials to a database in a connection string

Storing information in the Windows registry on the server

XML Web Services

Example of...

Creating a simple XML Web service

Creating a simple application that consumes an XML Web service

Creating a simple XML Web service accessed from a Windows Form

Creating a simple distributed application using an XML Web service

Declaring an XML Web service

Inheriting from the base WebService class

Defining XML Web service methods

Exposing an XML Web service methods

Accessing an XML Web service

Accessing an XML Web Service asynchronously

Related Code Examples

Code for Data Access

Code Examples for Visual Basic Programmers

Web Walkthroughs

Walkthroughs give step-by-step instructions for common scenarios, which makes them a good place to start learning about the product or a particular feature area.

The following walkthroughs will help orient you to the use of Web Forms pages and the features of ASP.NET in your Visual Basic and Visual C# applications:

Creating a Basic Web Forms Page
: Illustrates the basic techniques of how to create a Web Forms page in Visual Basic or Visual C#, add controls to it, and run it.

Creating a Web Application Using Visual C# or Visual Basic
: Describes how to write a Web Forms application and then integrate a business-object component that gives you a 10% discount for purchases.

Creating a Web Application Using a Third-Party Business Object
: Describes how to write a Web Forms application using Managed Extensions for C++ and then integrate a third-party business-object component that gives you a 10% discount for purchases.

Validating User Input in a Web Forms Page
: Describes how to use Web Forms validation controls to check user entries without code, including required entries, data types, patterns, and specific values.

Displaying Data in a Web Forms Page
: Illustrates a basic data-access scenario, providing step-by-step instructions for how to create a Web Forms page that displays data in a grid control.

Creating Read-Only Data Access in a Web Forms Page
: Describes how use an ADO.NET data command and data reader to provide optimized access to read-only data in a Web Forms page.

Updating Data Using a Database Update Query in Web Forms
: Describes the basic techniques of creating a Web Forms page that uses a data command to update the database.

Using a DataGrid Web Control to Read and Write Data
: Gives step-by-step details on how to use a grid control to allow users to view and edit data.

Creating Paged Data Access using a Web Forms Page
: Describes how to create a Web Forms page that allows users to view data records a few records at a time, paging back and forth to view earlier and later records.

Creating a Web User Control
: Describes how to create a Web Forms user control — a page configured to act as a control — and use it on another page.

Converting a Web Forms Page to a User Control
: Describes how to convert a basic Web Forms page into a user control that can be hosted on another page.

Creating a Web Custom Control
: Describes how to create a custom control, add it to the toolbox, and use it on a Web Forms page.

Displaying an XML Document in a Web Forms Page using Transformations
: Gives step-by-step details on how to read an XML file and display it in different ways on a Web Forms page.

See Also

Walkthroughs in Visual Basic and Visual C# | Visual Studio Walkthroughs | Samples and Walkthroughs | Visual Basic Code Example Topics

Security Considerations for ASP.NET Web Applications

Security is a vital aspect of your Web applications. The first step in creating a secure application is to understand what types of security issues arise in Web-based applications. You must also understand the basic strategies you use to protect the application and your system.

The topics listed below provide an introduction to security in Web applications. Because security is a broad subject, the "Related Sections" below provides links to places in the documentation where security is discussed in more detail.

In This Section

Introduction to Web Application Security
Provides an overview of security for Web applications, describing what types of issues you need to think about when creating applications in Visual Studio.

Web Application Security at Design Time in Visual Studio
Discusses the specific requirements for security that pertain to accessing resources you need during development. These requirements are distinct from those that apply to users of your application.

Web Application Security at Run Time
Discusses the security of your application when it is deployed and running.

Overview of Web Application Security Threats
Discusses the types of threats that your Web applications are subject to.

Basic Security Practices for Web Applications
Provides a list of the basic security recommendations that apply to all Web applications.

Scripting Exploits
Discusses ways that malicious users can send potentially damaging information to your application.

Protecting Against Script Exploits in a Web Application
Demonstrates how to use HTML encoding and HTML filtering to protect your application against malicious user input.

Displaying Safe Error Messages
Shows how to display error messages without giving away information that a malicious user might find helpful in attacking your system.

Access Permissions for Web Applications
Provides an overview of options for managing access to resources in a Web application.

Accessing SQL Server from a Web Application
Discusses design strategies for SQL Server access for your Web application.

Related Sections

Security Portal
Provides links to topics that provide an overview of security issues, including a bibliography.

Security Model
Discusses aspects of choosing a security model for a distributed application created using ASP.NET.

ASP.NET Web Application Security
Provides links to topics that describe how security works in the ASP.NET Framework.

Introduction to Web Application Security

Security is a critical part of your Web applications. Web applications by definition allow users access to a central resource — the Web server — and through it, to others such as database servers. By understanding and implementing proper security measures, you guard your own resources as well as provide a secure environment in which your users are comfortable working with your application.

This topic provides an overview of security for Web applications, describing what types of issues you need to think about when creating applications in Visual Studio. It also provides links to further information where you can learn more details about security features of Microsoft Internet Information Services (IIS) and of ASP.NET.

Authentication, Authorization, and Impersonation

Working with security requires that you understand these fundamental security concepts:

- *Authentication* confirms that users are who they say they are. For example, a user must provide a user name and password that are checked against an authority (a database, for example, or a Windows domain server).
- *Authorization* is the process of granting or denying access to resources for specific users.

These are concepts that are probably familiar from your use of Windows. In any sort of network, you log in (that is, you authenticate yourself). After doing that successfully, you can access specific files, folders, printers, and other resources — that is, you have authorization to get access to resources based on your login credentials. Windows provides a sophisticated security system that allows administrators to create user accounts and authorize those users to access folders, files, and so on.

There are security requirements in Web applications as well. However, the public nature of a Web application introduces some variations into this model. For example, in a public Web site, you would not expect every user to have to log into the computer or network where the Web server is running. To deal with this scenario, Web applications can *impersonate* users. Rather than requiring users to provide authentication credentials, a Web application might request a resource with some pre-established credentials.

Web Application Security in IIS and ASP.NET

Security for your Web application starts with the Web server (IIS). As a Windows-based service, IIS is fully integrated into Windows security. As with any other process, to access a file, IIS requires proper authentication. When users send a request from their browser to IIS, IIS must read the file from a Windows folder, and whatever Windows-defined authentication applies to the file applies to the IIS request as well. That is, to get access to a resource, IIS must provide proper credentials, the same as any other process.

When your Web applications run, they run under ASP.NET, which has its own security facilities. These come into play when your application requires access to resources. For example, if you want to read or write a file in your Web application, it is the ASP.NET security context that determines whether the request succeeds.

Not every user will have proper authentication to read files on a Windows server, however — particularly in Web applications available publicly on the Internet. IIS and ASP.NET therefore provide several mechanisms for establishing authentication. The descriptions below summarize these methods, necessarily leaving out many details about how the methods work. For details about IIS security, see the topics about access control in the IIS documentation on the Microsoft Web site (http://www.microsoft.com/windows2000/en/server/iis/). For details about ASP.NET security, see ASP.NET Web Application Security.

Anonymous Access

For applications where unknown users will be making requests (typically, public Web applications), IIS supports an "anonymous" user, namely, one who has no authentication credentials. In this scenario, the server on which IIS is running has an extra Windows user defined on it, with a user name of IUSR_<machinename>. This user account is typically defined with very restricted access rights.

When IIS gets a request from an unknown user, IIS turns around and makes the request to Windows using the <machinename>_anonymous user name as its credentials. That is, IIS impersonates the anonymous users for purposes of accessing the resource.

Basic and Digest Authentication

IIS also supports the Web-standard basic authentication model. In this scenario, users without credentials are prompted to supply a user name and password. These are returned to IIS where they become available to the application. Basic authentication

provides a useful way to provide restricted access in a public Web application. However, because the user passes a user name and password to IIS as clear text, it is not highly secure.

Digest authentication is similar to basic authentication, but uses encryption to send user information to the server. However, this style of authentication requires a Windows domain controller.

Windows Integrated Security

If the user making the request has already been authenticated in a Windows-based network, IIS can pass the user's credentials through when requesting access to a resource. (The credentials do not include the user name and password, only an encrypted token indicating the user's security status.) This is a typical scenario in a corporate intranet, where users log into the network before using your application. However, integrated security is not practical in applications that must go through firewalls. Even within the same network, integrated security has limitations for accessing resources that are on remote computers. For details, see Access Permissions for Web Applications.

Certificate Authentication

Finally, IIS supports the use of digital certificates and the secure sockets layer (SSL). In this scenario, either the server or the client has a certificate — a digital identification — that they have obtained from a third-party source. The certification is passed to your application with requests. It can be mapped to a Windows user account and granted appropriate permissions. This authentication has the advantage that, because each request or response is validated by the authority of the certificate, it lends itself to applications that require a trust relationship.

See Also

Access Permissions for Web Applications | Web Application Security at Design Time in Visual Studio |
Web Application Security at Run Time | Accessing SQL Server from a Web Application | ASP.NET Impersonation | Security Model

Web Application Security at Run Time

Developing your application requires that you work with one set of security issues. The other set of issues — and the ones that are generally more prominent in any discussion of Web security — address the security of your application when it is deployed and running.

Web applications by definition allow users access to a central resource — the Web server — and through it, to others such as database servers. By understanding and implementing proper security measures, you:

- Guard your own resources against unauthorized access.
- Restrict access levels by user or role.
- Establish data integrity and confidentiality, providing a secure environment in which your users are comfortable working with your application.
- Establish control over how your application can get access to restricted resources.
- Ensure that your application's code runs as intended.

This topic provides a general discussion of how you can accomplish these goals, and it includes links to additional topics where you can get more details on the technologies involved.

You can protect your application from unauthorized access by taking advantage of these types of security features:

- Security features offered by Internet Information Services (IIS) as part of its general Web server function. These include Windows file, computer, and user-level security.
- Security you can build into your ASP.NET application to provide application-specific access.

Security Process in ASP.NET

IIS provides a broad level of security for the Web site as a whole. However, IIS security mechanisms are very generic, in that the same mechanisms are used for all applications. Moreover, IIS security options — for example, using Windows integrated security — might not always be convenient for your application. (For intranet applications, on the other hand, you might want to use Windows integrated security for simplicity.)

Therefore, in order to provide more granular access to your application, you can use ASP.NET security. ASP.NET security works in conjunction with IIS security but extends it so that you can create more granular security and can customize features such as how you get user credentials.

IIS receives requests first from clients and performs any security checks that have been established for your application using IIS management tools. For example, if the application has been configured in IIS to allow anonymous access, IIS performs no credentials check. After performing this initial check of authentication, IIS sends the request to ASP.NET, which can perform a second level of checking. ASP.NET allows you to specify access restrictions in your application using a variety of criteria: you can restrict access to specific pages, or to specific users, and so on.

Authentication

ASP.NET supports the following ways to authenticate users. Several of them overlap with IIS authentication. For details, see Introduction to Web Application Security.

Authentication Type	Description
Anonymous Access	For applications where unknown users will be making requests (typically, public Web applications). Overlaps with IIS.
Basic and Digest Authentication	(Overlaps with IIS) In this scenario, users without credentials are prompted to supply a user name and password.

Windows Integrated Security (NTLM)	(Overlaps with IIS) If the user making the request has already been authenticated in a Windows-based network, IIS can pass the user's credentials through when requesting access to a resource.
Certificate Authentication	(Overlaps with IIS) In this scenario, the client has a certificate — a digital identification — that has been obtained from a third-party source. The certification is passed to your application with requests.
Kerberos	(Overlaps with IIS) Kerberos authentication protocol defines the interactions between a client and a network Authentication Service known as a Key Distribution Center (KDC). Windows 2000 implements a KDC as the authentication service on each domain controller. For details, see Basic Concepts for the Kerberos Protocol.
Forms authentication	If a user needs to be authenticated, ASP.NET redirects the request to a page that you specify. This page usually contains a form in which you get user name information. (For extra security, the form can be exchanged using HTTPS protocol.) When your application gets the form information, it can perform an application-specific check of the user's credentials. An important point is that the process of authentication is under your control (unlike that for IIS), which allows you to specify what the form looks like and how you choose to store user information. If a user is successfully authenticated, ASP.NET issues an encrypted cookie containing a token identifying the user for subsequent access.
Passport authentication	Using ASP.NET, you can check for users who have credentials they get from a Microsoft Passport-compliant application. If a user is successfully authenticated, ASP.NET issues an encrypted cookie containing a token identifying the user for subsequent access.

Forms authentication is generally the best choice in an ASP.NET application, because it gives you substantial control over how users are authenticated, and because it allows you to store an authentication in a token.

For details about IIS security, see the topic "About Access Control" in the IIS documentation on the Microsoft Web site (http://www.microsoft.com/windows2000/en/server/iis/). For details about ASP.NET authentication, see ASP.NET Authentication.

Authorization

When your Web application runs, it requests resources from the Web server and, often, from other processes as well, such as a database. The ASP.NET process runs in a user context that determines how your application will request those resources. By default, the ASP.NET process runs as a special local user, called ASPNET, with limited permissions. You can specify a different user context for the ASP.NET process, including "system" (which runs your application in administrator context) or a user whose credentials you explicitly provide.

In your ASP.NET application, you can specify that different users have authorized access to different resources. If your application is using Windows authentication, you can use Windows permissions to determine authorization to access specific files or folders on the server.

Alternatively, you can use URL-based authorization, in which authorization can be granted or denied according to different criteria:

- Specific users, or identities, which are based on the credentials provided by the user.
- Roles, which are entities defined to allow multiple users to share privileges based on a common role or function.
- Verbs, which are the HTTP processes (such as GET and POST) for accessing portions of your application.

For example, you can specify that all users can get pages (perform the GET verb) from your application, but that only specific users can post pages to it. Similarly, you might specify that all users are allowed to GET pages but specific roles are denied the right to post.

You can grant URL authorization for the application as a whole or on a directory-by-directory basis. A typical use is to allow all users to view pages in a public directory, but to keep restricted pages in a different directory that is authorized only for specific users or roles.

For more information, see Authorization, ASP.NET Authorization, and Access Permissions for Web Applications.

ASP.NET Configuration Files

You establish ASP.NET security choices using settings in a Web.config configuration file. The file allows you to include predefined

elements for various security options, including sections for authentication and authorization. The relevant sections of the Web.config file might look like the following:

```
<authentication mode="Forms">
   <forms name="login.aspx" loginUrl="[url]" >
      <credentials passwordFormat="SHA1">
         <user name="Mary" password="9611E4F94EC4972D5A537EA28C69F89AD28E5B36"/>
      </credentials>
   </forms>
</authentication>

<authorization>
   <allow verb="GET" users="*" />
   <allow verb="POST" users="Mary" />
   <deny verb="POST" users="*" />
</authorization>

<identity impersonate ="false"/>
```

Your application can contain more than one configuration file. By default, there is a configuration file at the application root that specifies security for the application as a whole — that is, security settings are inherited by subfolders. However, you can also create configuration files in individual folders to create security settings for that folder.

For more information, see ASP.NET Architecture and Forms Authentication Using an XML Users File.

XML Web Service Security

XML Web services run as Web applications using ASP.NET and therefore participate in the same security model as any ASP.NET application. For example, an XML Web service might be configured to use Basic authentication or Windows integrated security.

At design time, when you attempt to add a reference to an XML Web service — that is, when you request the XML Web service's discovery documents — the XML Web service will perform standard Web application authentication according to how it was configured. For example, if the XML Web service is configured to use Basic authentication, it will expect to get a user name and password from the requesting client. In Visual Studio, you typically use the Add Web Reference dialog box to discover XML Web services. If the XML Web service is using Basic authentication, for example, the Add Web Reference dialog box will prompt you for credentials.

If you are building an application that includes calls to an XML Web service, you need to be sure that you have appropriate credentials before making the call, or the call will fail. At run time, you can pass credentials to the XML Web service by setting the Credentials property of the client-side object representing the XML Web service before you call its methods.

Because other ASP.NET security options might not be sufficiently flexible, XML Web services can implement a custom authentication solution in which credential information is passed in SOAP headers. In this solution, credentials are passed in an optional part of the message exchanged between client and server. You can then write a custom **HttpModule** (implementing the IHttpModule interface) that can listen for the information in the header and call your authentication code.

As with other ASP.NET applications, the XML Web service might implement specific role-based authorizations to limit access to specific parts of the application.

For details, see XML Web Services Infrastructure and Securing XML Web Services Created Using ASP.NET.

Establishing Data Integrity and Confidentiality

Authentication and authorization establish who your users are and what resources they can access. These security features are designed primarily to protect your Web application from unauthorized use.

However, there is a separate aspect to security as well, which is to protect your users' information and provide them with the confidence that they can exchange sensitive information with you. For example, if your application asks users for credit card or other account numbers, personal information, or any data that users might not want others to know, you must provide a way for them to submit this information to you securely.

The solution is to use the Secure Sockets Layer (SSL) of IIS. This allows users to exchange encrypted information with you using HTTPS (secure) protocol. SSL provides security in both directions: your information is transmitted to the user using encryption, and information the user posts to your application is likewise encrypted.

Establishing SSL and Encryption

To use SSL and encryption, you must obtain a server certificate for your company or identity. The certificate is a digital signature that identifies your site in a way that cannot be impersonated. For Internet (public) applications, you obtain a server certificate from a recognized third-party certification authority. For private (intranet) applications, you can issue a server certificate yourself. You might do this to secure an internal application such as a personnel site.

Your server certificate also enables you to use encryption. SSL uses an encryption method called *public key encryption*. In this form of encryption, there are two keys: a public key used to encrypt data and a private key that you keep secret and use to decrypt information encrypted with the public key. The server certificate you obtain includes a public key. When users want to use SSL, your application sends the certificate and the public key to the browser. The browser and server then use the public key to establish a way to encrypt their information exchange.

> **Note** Using SSL requires that the browser support an encryption strength of at least 40 bits. This level is available in most browsers.

Once you have obtained a server certificate, you can instruct users to use SSL by having them use https:// as the prefix for getting and posting Web pages. IIS and the browser will automatically use an encrypted channel to exchange information.

For details on how to implement SSL, see Microsoft Knowledge Base article Q307267, "HOW TO: Secure XML Web Services with Secure Socket Layer in Windows 2000." For information on encryption, see Cryptography Overview. For information about certificates and about configuring SSL, see the topics in the IIS documentation on the Microsoft Web site (http://www.microsoft.com/windows2000/en/server/iis/).

Using .NET Code Security

A final aspect of security is in ensuring that the code in your application is protected from misuse, whether by being inadvertently run in an improper context or by being used in a malicious way. Because ASP.NET is part of the .NET Framework, you can also take advantage of code access security and role-based security to establish permissions for what code is allowed to do. For information, see ASP.NET Process Identity, Database Security, and Code Access Security.

See Also

Access Permissions for Web Applications | Security Model

Overview of Web Application Security Threats

Someone is going to try to hack your Web application. That is, if unknown users can get to your Web application, the odds are almost certain that someone will try to do something unauthorized or even malicious to your application. Most servers that are accessible to the public on the Internet are typically probed for vulnerabilities at least four or five times a day.

Therefore, no matter what your Web application is, you should build security into it. You should think about security the same way that you do about backup. Although you might not think that you will ever have a problem, it is always safest to assume that you will and to take precautions, just in case.

Perfect Security is Impossible

The first lesson about Web application security is that there is no such thing as perfect security. Even in secure systems, malicious users can wield sophisticated tools to try to get into your system. Fortunately, most attacks come from relatively unsophisticated users who simply look for sites that are easy to break in to.

Your initial goal should be to protect your site from the common and redundant exploits that comprise the vast majority of attacks.

Security Technology Is Only Part of the Solution

Implementing security is only part of the solution. Another important part is vigilance. Even if your system has many security safeguards, you need to watch it closely in these ways:

- Monitor your system's event logs. Watch for repeated attempts to log into your system or for a very high number of requests against your Web server.
- Learn about and install the latest security patches from Microsoft and other vendors. For example, the Microsoft security site (www.microsoft.com/security) has a list of the latest security bulletins for all Microsoft products. Other vendors have similar sites.

Threat Modeling

An important part of building a secure application is to understand the threats to it. Microsoft has developed a way to categorize threats that goes by the acronym STRIDE (Spoofing-Tampering-Repudiation-Information disclosure-Denial of service-Elevation of privilege). The sections below briefly describe these threats and how they apply to Web applications. For more information, see Designing for Securability.

Spoofing

To *spoof* is to impersonate a user or process in an unauthorized way. At its simplest, spoofing can mean typing in a different user's credentials. A malicious user might also change the contents of a cookie to pretend that he is a different user or that the cookie comes from a different server.

In general, you can prevent spoofing by using stringent authentication. Any time someone requests access to non-public information, make sure they are who they say they are. You can also defend against spoofing by keeping credential information safe. For example, do not keep a user name (or worse, a password) in a cookie, where a malicious user can easily find or modify it.

Tampering

Tampering means changing or deleting a resource without authorization. One example is defacing a Web page, where the malicious user gets into your site and changes files. An indirect way to tamper is using a *script exploit*. A malicious user manages to get code (script) to execute by masking it as user input from a form or as a link.

A primary defense against tampering is to use Windows security to lock down files, directories, and other Windows resources. The application should also run with the minimum privileges it can. You guard against script exploits by being extremely distrustful of any information that comes from a user or even from a database. Whenever you get information from an untrusted source, sanitize it by making sure it does not contain any executable code.

Repudiation

A repudiation threat is to hide evidence of an attack. In a Web application, this can mean impersonating an innocent user's credentials. You can guard against repudiation by, again, using stringent authentication. In addition, use the logging features of Windows to keep an audit trail of any activity on the server. For details, see Event Logging in the Platform SDK documentation.

Information Disclosure

Information disclosure simply means stealing or revealing information that is supposed to be private. A classic example is stealing passwords, but it can involve access to any file or resource on the server.

The best defense against information disclosure is to have no information to disclose. For example, if you avoid storing passwords, malicious users cannot steal them. (An alternative to storing passwords is to store only a hash of the password. When a user presents credentials, you can hash the user's password and compare only the hashes of the two.) If you do store secrets, use Windows security to secure it. As always, you should use authentication to be sure that only authorized users get access to restricted information. You can also encrypt information to help keep it secret.

Denial of Service

A *denial of service* attack is to deliberately cause an application to be less available than it should be. A typical example is to overload a Web application so that it cannot serve ordinary users. Alternatively, malicious users might try to simply crash your server.

IIS allows you to *throttle* applications, which means that it limits the number of requests it will serve. You might be able to deny access to users or IP addresses known to be malicious. Protecting against crashes is a matter of running robust code. You should test your application thoroughly and recover gracefully from error conditions, if possible.

Elevation of Privilege

An elevation of privilege is to use malicious means to get more permissions than normally assigned. For example, in a successful elevation of privilege attack, a malicious user manages to get administrative privileges to your Web server, giving themselves a free hand to wreak havoc.

To protect against elevation of privilege, run the application in a least-privilege context if practical. For example, it is recommended that you do not run ASP.NET applications as the SYSTEM (administrative) user.

See Also

Security Portal | Securability | Best Practices for Securability | Introduction to Web Application Security | ASP.NET Web Application Security | Security Bibliography

Basic Security Practices for Web Applications

The topic of creating a secure Web application is a long one. It requires study to understand security vulnerabilities. You also need to familiarize yourself with the security facilities of Windows, the .NET Framework, and ASP.NET. Finally, it is essential to understand how to use these security features to counter threats.

However, even if you are not experienced with security, there are basic measures that you should take to protect your Web application. The list below provides minimum-security guidelines that apply to all Web applications.

- General Web Application Security Recommendations
- Run Applications with Least Privileges
- Know Your Users
- Guard Against Malicious User Input
- Access Databases Securely
- Create Safe Error Messages
- Keep Secrets Safely
- Use Cookies Securely
- Guard Against Denial-of-Service Threats

General Web Application Security Recommendations

Some of the most basic security recommendations are also the most obvious ones. However, even the most elaborate application security can fail if a malicious user can use simple ways to get to your computers.

- Back up often and keep your backups physically secure.
- Keep your Web server computer physically secure so that unauthorized users cannot get to it, turn it off, take it, and so on.
- Use the Windows NTFS file system, not FAT32. NTFS offers substantially more security than FAT32. For details, see the Windows documentation.
- Secure the Web server computer and all computers on the same network with strong passwords.
- Secure IIS. For details, see the article Manage Security of Your Windows IIS Web Services in the Microsoft security Web site (http://microsoft.com/technet/treeview/default.asp?url=/technet/security/bestprac/mcswebbp.asp).
- Close unused ports and turn off unused services.
- Run a virus checker that monitors traffic.
- Establish and enforce a policy that forbids users from keeping their passwords written down in an easy-to-find location.
- Use a firewall. For recommendations, see Checklist: Install a Firewall) on the Microsoft security site. (http://microsoft.com/security/articles/firewall.asp)
- Learn about and install the latest security patches from Microsoft and other vendors. For example, the Microsoft security site (www.microsoft.com/security) has a list of the latest security bulletins for all Microsoft products. Other vendors have similar sites.
- Use Windows event logging and examine the logs frequently for suspicious activity. This includes repeated attempts to log on to your system and an extremely high number of requests against your Web server.

Run Applications with Least Privileges

When your application runs, it runs within a context that has specific privileges on the local computer and potentially on remote computers. Follow these guidelines:

- Run the application in the context of a user with the minimum practical privileges. By default, under IIS version 5, an ASP.NET application runs in the context of a local user called ASPNET. In IIS 6 on Windows Server 2003, the account is called NETWORK SERVICE.For details, see ASP.NET Process Identity and Configuring ASP.NET Process Identity.
- Do not run your application within the context of a system user (administrator).
- Set permissions (ACLs, or Access Control Lists) on all the resources required for your application. Use the least permissive setting. For example, if practical in your application, set files to be read-only. For details, see the Windows documentation.
- Keep files for your Web application in a folder below the application root. Do not allow users the option of specifying a path for any file access in your application. This helps prevent users from getting access to the root of your server.
- Do not make anyone a member of the Debugger Users group who does not need to debug Visual Studio applications. This group has higher privileges than the VS Developers group. For details, see Adding Debugger Users and Configuring DCOM.

Know Your Users

In many applications, it is possible for users to access the site without having to provide credentials. If so, your application accesses resources by running in the context of a predefined user. By default, this context is the local ASPNET user (Windows 2000 or Windows XP) or NETWORK SERVICE user (Windows Server 2003) on the Web server computer.

To restrict access to users who are authenticated, follow these guidelines:

- If your application is an intranet application, configure it to use Windows integrated security. That way, the user's logon credentials can be used to access resources.
- If you need to gather credentials from the user, use one of the ASP.NET authentication strategies. For more information, see ASP.NET Authentication. For an example, see Simple Forms Authentication.

Guard Against Malicious User Input

As a general rule, never assume that input you get from users is safe. It is easy for malicious users to send potentially dangerous information from the client to your application. To guard against malicious input, follow these guidelines:

- In forms, filter user input to check for HTML tags, which might contain script. For details, see Protecting Against Script Exploits in a Web Application.
- Never echo (display) unfiltered user input. Before displaying untrusted information, encode HTML to turn potentially harmful script into display strings.
- Similarly, never store unfiltered user input in a database.
- If you want to accept some HTML from a user, filter it manually. In your filter, explicitly define what you will accept. Do not create a filter that tries to filter out malicious input; it is very difficult to anticipate all possible malicious input.
- Do not assume that information you get from the header (usually via the Request object) is safe. Use safeguards for query strings, cookies, and so on. Be aware that information the browser reports to the server (user agent information) can be spoofed, in case that is important in your application.
- If possible, do not store sensitive information in a place accessible from the browser, such as hidden fields or cookies. For example, do not store a user name or password in a cookie.

> **Note** View state is stored in a hidden field in an encoded format. By default, it includes a message authentication code (MAC) so that the page can determine whether view state was tampered with.

Access Databases Securely

Databases typically have their own security. An important aspect of a secure Web application is designing a way for the application to access the database securely. Follow these guidelines:

- Use the inherent security of your database to limit who can access database resources. The exact strategy depends on your database and your application:
  - If practical in your application, use integrated security so that only Windows-authenticated users can access the database. Integrated security is more secure than using connection strings.
  - If your application involves anonymous access, create a single user with very limited permissions, and perform queries by logging on as this user.
- Do not create SQL statements by concatenating strings that involve user input. Instead, create a parameterized query and use user input to set parameter values. For details, see Parameters in Data-Adapter Commands.
- If you must store a user name and password somewhere to use as the database login credential, store them securely. If practical, encrypt them. For details, see Encrypting and Decrypting Data. For an introduction to encryption, see Geek Speak Decoded #9: Cryptography and Encryption.

Create Safe Error Messages

If you are not careful, a malicious user can deduce important information about your application from the error messages it displays. Follow these guidelines:

- Do not write error messages that echo information that might be useful to malicious users, such as a user name.
- Configure the application not to show detailed errors to users. If you want to display detailed error messages for debugging, check first that the user is local to the Web server. For details, see Displaying Safe Error Messages.
- Create custom error handling for situations that are prone to error, such as database access.

Keep Secrets Safely

Secrets are any information that you do not want someone to know. A typical secret is a password or an encryption key. Of course, if a malicious user can get to the secret, then the data protected by the secret is compromised. Follow these guidelines:

- If your application transmits sensitive information between the browser and the server, consider using the Secure Sockets Layer (SSL). For details on how to implement SSL, see Microsoft Knowledge Base article Q307267, "HOW TO: Secure XML Web Services with Secure Socket Layer in Windows 2000."
- If you must store secrets, do not keep them in a Web page, even in a form that you think people will not be able to see it (such as in server code).
- If you encrypt information, do not create your own encryption algorithm. Use the Windows Data Protection API (DPAPI).

Use Cookies Securely

Cookies are an easy and useful way to keep user-specific information available. However, because cookies are sent to the browser's computer, they are vulnerable to spoofing or other malicious use. Follow these guidelines:

- Do not store any critical information in cookies. For example, do not store a user's password in a cookie, even temporarily. As a rule, do not keep anything in a cookie that, if spoofed, can bring you trouble. Instead, keep a reference in the cookie to a location on the server where the information is.
- Set expiration dates on cookies to the shortest practical time you can. Avoid permanent cookies if possible.
- Consider encrypting information in cookies.

Guard Against Denial-of-Service Threats

An indirect way that a malicious user can compromise your application is by making it unavailable. The malicious user can keep the application too busy to service other users, or if nothing else can simply crash the application. Follow these guidelines:

- Close or release any resource you use. For example, always close data connections and data readers, and always close files when you are done using them.
- Use error handling (for example, try-catch). Include a finally block in which you release resources in case of failure.
- Configure IIS to use process throttling, which prevents an application from using up a disproportionate amount of CPU time. For details, see the technical article Using the New Features in Internet Information Server 5.0: Part 1.
- Test size limits of user input before using or storing it.
- Put size safeguards on database queries. For example, before you display query results in a Web Forms page, check that there are not an unreasonable number of records.
- Put a size limit on file uploads, if those are part of your application. You can set a limit in the Web.config file using syntax such as the following, where the `maxRequestLength` value is in kilobytes:

```
<configuration>
   <system.web>
        <httpRuntime maxRequestLength="4096" />
   </system.web>
</configuration>
```

See Also

Overview of Web Application Security Threats | Security Portal | Securability | Best Practices for Securability | Designing Secure ASP.NET Applications | ASP.NET Web Application Security | Security Bibliography

Scripting Exploits

From the perspective of a browser, a Web page is simply a long string of characters. The browser processes the string sequentially, displaying some characters while interpreting other characters such as `<b>` and `<script>` according to special rules. If a malicious user can insert some of those special characters into a page, the browser will not know that the characters are not supposed to be there, and it will process them as part of the page.

A simplistic script exploit might work as follows. Imagine an application that allows users to post comments about the latest movies for other users to read. The exploit steps might be:

1. The application displays a form where users enter comments. The malicious user writes a comment that includes a `<script>` block in it.
2. The form is posted and the malicious user's comment is stored in a database.
3. Another user visits the site. When the page is constructed, it reads comments out of the database and puts them into the page. The malicious user's `<script>` block is written into the page as if it were a text comment.
4. When the second user's browser displays the page, it gets to the `script` block and executes it.

There are other ways that malicious users can exploit script as well. The common thread is that most script exploits require the application to accept the malicious input and inject it (echo it) into a page where it will be executed by the browser. The potential damage from such an exploit depends on the script that is executed. It can be trivial, such as an annoying message that pops up in the browser. But it can also do serious damage by stealing cookies, stealing user input (such as a password), and if Internet security is lax, even running native code on the user's computer.

For information about preventing scripting exploits, see Protecting Against Script Exploits in a Web Application.

SQL Statement Exploits

A variation on a script exploit is one that causes malicious SQL statements to be executed. This can occur if an application prompts users for information and then concatenates the user's input into a string representing the SQL statement. For example, an application might prompt for a customer name with the intention of executing a statement such as the following:

```
"Select * From Customers where CustomerName = " & txtCustomerName.Value
```

But a malicious user who knows something about the database could use the text box to enter an embedded SQL statement with the customer name, resulting in a statement like the following:

```
Select * From Customers Where CustomerName = 'a' Delete From
Customers Where CustomerName > ''
```

When the query is executed, the database is compromised.

Guarding Against Scripting Exploits

The primary defense against scripting exploits is to simply never trust information coming from a user. Assume that any data posted to your application from a browser can contain malicious script.

Similarly, any time that you write a string into a page, you should assume the string could contain malicious script. (Unless you programmatically created the string yourself.) For example, when you read strings out of a database, you should assume that they can contain malicious script. The most security-conscious developers distrust even their own databases, on the theory that a malicious user might have found a way to tamper with the database.

ASP.NET provides you with several ways to protect against scripting exploits:

- ASP.NET protects you against this automatically with request validation. By default, if the Request object contains HTML-encoded elements or certain HTML characters (such as `—` for an em dash), the ASP.NET page framework raises an error. It is recommended that you create a handler in your application for this error.
- If you want to display strings in your application but do not trust them, apply HTML encoding to them. For example, with encoding, the tag `<b>` becomes `<b>`. You might do this if the strings you are displaying are from a database whose contents you are not sure you can trust.
- If you want your application to accept some HTML (for example, some formatting instructions from users), turn off automatic checking and create a filter that defines exactly what HTML the application will accept.

Security Note Never disable automatic request validation without adding your own checks or filter.

Note Do not create a filter that attempts to filter out only unacceptable elements, because it is difficult to anticipate every possible bad input. Instead, if you do create a filter, create one that has a defined list of acceptable input.

- To protect against SQL statement exploits, never create SQL queries using string concatenation. Instead, use a parameterized query and assign user input to parameter objects. For details, see Parameters in Data-Adapter Commands.

See Also

Protecting Against Script Exploits in a Web Application | Overview of Web Application Security Threats | Basic Security Practices for Web Applications | Security Portal

Protecting Against Script Exploits in a Web Application

Most scripting exploits occur when users can get executable code (script) into your application. By default, ASP.NET provides request validation, which raises an error if a form post contains any HTML whatsoever.

You can protecting against script exploits in these ways:

- Apply HTML encoding to strings before accepting or displaying them, so that the strings do not include any executable elements.
- If your application needs to accept some HTML, disable request validation and create your own HTML filter.

The procedures in this topic describe how to perform these tasks.

Applying HTML Encoding

HTML encoding converts HTML elements using HTML reserved characters so that they are displayed rather than executed.

To apply HTML encoding

- Before displaying strings, call the Server object's HtmlEncode method. HTML elements are converted into string representations that the browser will display rather than interpret as HTML.

  The following example illustrates HTML encoding. In one instance, the user input is encoded before being displayed. In the second instance, data from a database is encoded before being displayed.

  > **Note** This example will only work if you disable request validation in the page by adding the @ Page attribute ValidateRequest="false" . Never disable request validation without adding your own check or filter.

```
' Visual Basic
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e _
      As System.EventArgs) Handles Button1.Click
   Label1.Text = Server.HtmlEncode(TextBox1.Text)
   Label2.Text = _
       Server.HtmlEncode(dsCustomers.Customers(0).CompanyName)
End Sub

// C#
private void Button1_Click(object sender, System.EventArgs e)
{
    Label1.Text = Server.HtmlEncode(TextBox1.Text);
    Label2.Text =
        Server.HtmlEncode(dsCustomers1.Customers[0].CompanyName);
}
```

Filtering HTML Elements

By default, Web Forms pages detect any HTML elements and reserved characters in information posted to the server. This prevents users from trying to embed script into your application. When the page detects HTML, it raises an error. You can catch this error using a **Page_Error** or **Application_Error** handler. For details, see Displaying Safe Error Messages.

However, if your application needs to accept some HTML elements, turn off request validation and create a filter that allows only the HTML elements that you want to accept.

> **Note** Do not create a filter that attempts to filter out only unacceptable elements, because it is difficult to anticipate every possible bad input. Instead, if you do create a filter, create one that defines acceptable input.

To filter HTML elements

1. Disable request validation by the adding the attribute `ValidateRequest="false"` to the @ Page directive.

   > **Security Note** Never disable automatic request validation without adding your own checks or filter.

2. Encode the string using the HtmlEncode method.
3. Call the String.Replace method to convert the encoded HTML tags you want to accept back to their HTML forms.

> **Tip** If you are familiar with regular expressions, you can use one to perform the filtering efficiently. For details, see .NET Framework Regular Expressions.

The following example illustrates a simple filter that accepts bold and underscore elements (, , <u>, </u>). All other user input is encoded before being displayed.

> **Security Note** Many HTML tags allow script in their attributes. For example, the tag
> <img src="javascript:alert('hi')">
> is legal. If you want to accept HTML tags that are more complex than simple formatting tags, you must be sure that a malicious user cannot pass script to your application in the guise of an allowed HTML tag.

```
' Visual Basic
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Han
dles Button1.Click
   Dim userinput As String = TextBox1.Text
   userinput = Server.HtmlEncode(userinput)
   ' Accepts <b>, </b>, <u>, </u>, case-insensitive
   userinput = userinput.Replace("&lt;b&gt;", "<b>")
   userinput = userinput.Replace("&lt;/b&gt;", "</b>")
   userinput = userinput.Replace("&lt;B&gt;", "<B>")
   userinput = userinput.Replace("&lt;/B&gt;", "</B>")
   userinput = userinput.Replace("&lt;u&gt;", "<u>")
   userinput = userinput.Replace("&lt;/u&gt;", "</u>")
   userinput = userinput.Replace("&lt;U&gt;", "<U>")
   userinput = userinput.Replace("&lt;/U&gt;", "</U>")
   Label1.Text = userinput
End Sub

// C#
private void Button1_Click(object sender, System.EventArgs e)
{
    String userinput = TextBox1.Text;
    userinput = Server.HtmlEncode(userinput);
    // Accepts <b>, </b>, <u>, </u>, case-insensitive
    userinput = userinput.Replace("&lt;b&gt;", "<b>");
    userinput = userinput.Replace("&lt;/b&gt;", "</b>");
    userinput = userinput.Replace("&lt;B&gt;", "<B>");
    userinput = userinput.Replace("&lt;/B&gt;", "</B>");
    userinput = userinput.Replace("&lt;u&gt;", "<u>");
    userinput = userinput.Replace("&lt;/u&gt;", "</u>");
    userinput = userinput.Replace("&lt;U&gt;", "<U>");
    userinput = userinput.Replace("&lt;/U&gt;", "</U>");
    Label1.Text = userinput;
}
```

See Also

Scripting Exploits | Overview of Web Application Security Threats | Basic Security Practices for Web Applications | Security Portal

Displaying Safe Error Messages

When your application displays error messages, it should not give away information that a malicious user might find helpful in attacking your system. For example, if your application unsuccessfully tries to log into a database, it should not display an error message that includes the user name it is using.

There are a number of ways to control error messages:

- Configure the application not to show verbose error messages to remote (application) users. You can optionally redirect errors to an application page.
- Include error handling whenever practical and construct your own error messages. In your error handler, you can test to see if the user is local and react accordingly.
- Create a global error handler at the page or application level that catches all unhandled exceptions and routes them to a generic error page. That way, even if you did not anticipate a problem, at least users will not see an exception page.

To configure the application to turn off errors for remote users

- In the Web.config file for your application, make the following changes to the `customErrors` element:
  - Set the `mode` attribute to `RemoteOnly` (case sensitive). This configures the application to show detailed errors only to local users (you, the developer).
  - Optionally include a `defaultRedirect` attribute that points to an application error page.
  - Optionally include `<error>` elements that redirect specific errors to specific pages. For example, you can redirect standard 404 errors (page not found) to your own application page.

  The following example shows a typical `customErrors` block in the Web.config file.

```
<customErrors mode="RemoteOnly" defaultRedirect="AppErrors.aspx">
   <error statusCode="404" redirect="NoSuchPage.aspx"/>
   <error statusCode="403" redirect="NoAccessAllowed.aspx"/>
</customErrors>
```

To include error handling

1. Use try-catch-finally blocks around any statements that might generate errors.
2. Optionally, test for a local user with the Context object's UserHostAddress property and modify error handling accordingly. The value 127.0.0.1 is equivalent to "localhost" and indicates that the browser is on the same computer as the Web server.

   The following shows an example error-handling block. If an error occurs, a Session state variable is loaded with details about the message and the application then displays a page that can read the Session variable and display the error. (The error is deliberately written to provide no exploitable details to the user.) If the user is local, different error details are provided. In the **finally** block, an open resource is released.

```
' Visual Basic
Try
   SqlConnection1.Open()
   SqlDataAdapter1.Fill(Me.DsPubs1)
Catch ex As Exception
   If HttpContext.Current.Request.UserHostAddress = "127.0.0.1" Then
      Session("CurrentError") = ex.Message
    Else
      Session("CurrentError") = "Error processing page."
    End If
    Server.Transfer("ApplicationError.aspx")
Finally
       SqlConnection1.Close()
End Try

// C#
try
```

```
{
    sqlConnection1.Open();
    sqlDataAdapter1.Fill(dsCustomers1);
}
catch (Exception ex)
{
    if(HttpContext.Current.Request.UserHostAddress == "127.0.0.1")
    { Session["CurrentError"] = ex.Message; }
    else
    { Session["CurrentError"] = "Error processing page."; }
    Server.Transfer("ApplicationError.aspx");
}
finally
{
    this.sqlConnection1.Close();
}
```

You can also create an error handler that catches all unhandled exceptions at the page level or for the application as a whole.

To create global error handler

- To create a global handler in a page, create a handler for the Page_Error event. To create an application-wide error handler, in the Global.asax file, add code to the Application_Error method. These methods are called if an unhandled exception occurs anywhere in your page or application. You can get information about the most recent error from the HttpServerUtility.GetLastError method.

  Note If you have a global error handler, it takes precedence over error handling specified in the defaultRedirect attribute of the Web.config customErrors element.

  The following shows a sample handler that gets information about the current error, puts it into a Session variable, and calls a generic error-handling page that can extract and display the error information.

```
' Visual Basic
Sub Application_Error(ByVal sender As Object, ByVal e As EventArgs)
   Session("CurrentError") = "Global: " & Server.GetLastError.Message
   Server.Transfer("lasterr.aspx")
End Sub

// C#
protected void Application_Error(Object sender, EventArgs e)
{
    Session["CurrentError"] = "Global: " +
        Server.GetLastError().Message;
    Server.Transfer("lasterr.aspx");
}
```

See Also

Page.ErrorPage Property | NET Samples - ASP.NET Deployment | Exception Management in .NET | Overview of Web Application Security Threats | Basic Security Practices for Web Applications | Security Portal

Accessing SQL Server from a Web Application

When a Web application involves database access, it must provide credentials to SQL Server (that is, it must log in to SQL Server) just as any other user or process would. In a Web application, this can introduce complications. For example, if the Web application runs anonymously, there might not be credentials to pass to SQL Server.

There are a number of ways to design SQL Server access for your Web application. The strategy you choose depends on how your computers are configured and whether you are on an intranet. The simplest options are:

- Use Windows integrated security. This option passes the user's credentials to SQL Server. Because of delegation issues, this is practical primarily if SQL Server is on the same computer as IIS and your users must be on the same domain as the Web server computer. For details, see Accessing SQL Server Using Windows Integrated Security.
- Access the SQL Server as the local ASPNET or NETWORK SERVICE process (the default user identity for a Web application in your application). This option works well for anonymous access, but works only if SQL Server and the Web server are on the same computer. For details, see Accessing SQL Server as a Local User.
- Map the Web application process to a Windows domain user and then log into the database as that user. This works well for anonymous access if SQL Server and the Web server are on separate computers. For details, see Accessing SQL Server Using a Mapped Windows Domain User.
- Pass an explicit user name and password in a connection string. This option is practical if you can prompt the user for credentials, but can be less secure than other options. You can also choose it as an easier alternative to mapping to a Windows domain user. For details, see Accessing SQL Server with Explicit Credentials.

If you are familiar with other Windows features that help maintain the protection of users' information such as Kerberos, Active Directories, Passport, and so on, you can also take advantage of those to create database access. For more information, see Kerberos Authentication and Trust and Active Directory Security.

See Also

Access Permissions for Web Applications | Database Security | Security Portal | Overview of Web Application Security Threats | Basic Security Practices for Web Applications

Accessing SQL Server Using Windows Integrated Security

If your application runs on a Windows-based intranet, you might be able to use Windows integrated security for database access. Integrated security requires:

- That SQL Server be running on the same computer as IIS.
- That all application be users on the same domain so that their credentials are available to the Web server. (That is, Windows integrated security is not practical for a public Web site.)

> **Note** If SQL Server and IIS are not on the same computer, use an alternative access method. For details, see Accessing SQL Server with Explicit Credentials and Accessing SQL Server Using a Mapped Windows Domain User.

To access SQL Server using Windows integrated security, you need to configure four different areas of your application:

- IIS
- The application configuration file (Web.config)
- Connection strings
- SQL Server itself

Configuring IIS

You need to configure your application in IIS to turn off anonymous access and turn on Windows authentication.

To configure IIS for Windows integrated security

1. In Windows, open the Internet Information Services administration tool.
   - In Windows 2000, this is available from the Start menu by pointing to Programs, then Administrative Tools, and then Internet Services Manager.
   - In Windows XP, this is available from Administrative Tools in the Control Panel.
2. Open the node for your server, and then open nodes until you find the node for your application, typically under Default Web Site.
3. Right-click your application and choose **Properties**.
4. In the **Directory Security** tab, click **Edit**.
5. In the **Authentication Methods** dialog box, clear the **Anonymous Access** box and make sure **Integrated Windows authentication** is checked.
6. Click **OK** to close all the dialog boxes.

Configuring the Web.config File

In the application configuration file (Web.config), you establish the authentication mode that your application uses and establish that the application will impersonate the user's credentials — that is, that it will run as that user.

To configure Web.config to allow Windows integrated security

- Open the Web.config file for your application and add the following elements to it:

  ```
  <authentication mode="Windows" />
  <identity impersonate="true"/>
  ```

  The <authentication> element might already be there.

  > **Note** Elements in Web.config are case sensitive.

Creating Connection Strings

When you create a connection string to access SQL Server, you must include attributes that tell SQL Server that you are using Windows integrated security.

To configure connection strings for Windows integrated security

- In any connection string for SQL Server, include the attribute `Trusted_Connection=Yes` and remove the username and password attributes.

  The following shows a typical connection string configured for Windows integrated security:

  ```
  "workstation id=WebServer1;packet size=4096;
  Trusted_Connection=Yes;data source=mySqlServers";
  persist security info=False;initial catalog=northwind"
  ```

Configuring SQL Server

You must set up SQL Server to recognize the users who will be accessing it.

To configure SQL Server for Windows integrated security

1. From the Windows **Start** menu, choose **Microsoft SQL Server**, and then choose **Enterprise Manager**.
2. Open the node for the server and expand the node for the database you want to give users permissions for.
3. Right-click the **Users** node and choose **New Database User**.
4. In the **Database User Properties** dialog box, enter *domain\username* in the **Login name** box, and then click **OK**. Alternatively, configure the SQL Server to allow all domain users to access the database.

See Also

Accessing SQL Server from a Web Application | Access Permissions for Web Applications | Database Security | ASP.NET Authentication | Security Portal

Accessing SQL Server Using a Mapped Windows Domain User

By default, ASP.NET applications run in the context of the local user called ASPNET in Windows 2000 and Windows XP or NETWORK SERVICE in Windows Server 2003. These user accounts have limited permissions. However, the ASPNET or NETWORK SERVICE user is local to the Web server computer and is therefore not recognized as a user on remote computers. To get around this limitation, you can have your application run in the context of a Windows domain user that is recognized on both the Web server computer and on the SQL Server computer.

Mapping your application process to a Windows domain user requires that you configure the following processes:

- **The Web server computer** You must be sure that the Windows domain user you specify has sufficient privileges (but no more) to run a Web application.
- **Your application** You need to configure the Web.config file to specify that ASP.NET use integrated security, which allows ASP.NET to recognized the mapped user name.

  Note For information on the machine.config and Web.config files, see ASP.NET Configuration.

- **Connection string** When you create connection strings for connection objects in your application, you need to specify that they will use integrated security.
- **SQL Server** You need to add the specified domain user as a SQL Server login user.

Configuring Users on the Web Server Computer

To set user permissions for the Windows domain user

- Using Windows administrative tools on the Web server computer, make sure that the mapped Windows domain user has necessary privileges. For details, see ASP.NET Required Access Control Lists (ACLs).

Mapping to the Windows User and Enabling Impersonation

After establishing the correct permissions for the ASPNET or NETWORK SERVICE user account and for the user to map to, you configure the application to impersonate that user.

To configure your Web application for integrated security

- Open the Web.config file for your application and add the following elements:

  ```
  <authentication mode="Windows" />
  <identity impersonate="true" userName="domain\username" password="password"/>
  ```

  The `<authentication>` element might already be there.

  Note Elements in Web.config are case sensitive.

Using Windows Security in the Connection String

Finally, when you create connection strings for database access, configure them to use Windows integrated security.

To use Windows security in the connection string

- When you create a connection string for your application, do not include a user name and password. Instead, set the connection string's Integrated Security attribute to SSPI.

  The following example shows a connection string that includes the appropriate attributes:

  ```
  data source=myserver;initial catalog=northwind;Integrated Security=SSPI
  ```

To configure SQL Server for integrated security

1. From the Windows **Start** menu, choose **Microsoft SQL Server**, and then choose **Enterprise Manager**.
2. Open the node for the server and expand the node for the database you want to give users permissions for.
3. Right-click the **Users** node and choose **New Database User**.
4. In the **Database User Properties** dialog box, enter *domain\username* in the **Login name** box, and then click **OK**.

See Also

Accessing SQL Server from a Web Application | Access Permissions for Web Applications | Database Security | ASP.NET Authentication | Security Portal | INFO: Implementing Impersonation in an ASP.NET Application (Q306158) (KnowledgeBase article)

Accessing SQL Server as a Local User

When an ASP.NET application runs, by default it runs in the context of a special local user called ASPNET (in Windows 2000 and Windows XP) or NETWORK SERVICE (in Windows Server 2003). This user account has just enough permissions on the Web server computer to allow the application to run.

If SQL Server and the Web server are on the same computer, you can define the ASPNET or NETWORK SERVICE user as a user on the SQL Server.

The advantages of this method are that it is secure, because the Web application runs within a limited security context. Using a single user name with SQL Server is also efficient because it enables SQL Server to take advantage of connection pooling, enhancing the scalability of the application.

To give the ASPNET or NETWORK SERVICE user SQL Server permissions

1. From the Windows **Start** menu, choose **Microsoft SQL Server**, and then choose **Enterprise Manager**.
2. Open the node for the server and expand the node for the database you want to give users permissions for.
3. Right-click the **Users** node and choose **New Database User**.
4. In the **Database User Properties** dialog box, enter *computername*\ASPNET in the **Login name** box, and then click **OK**.

See Also

Accessing SQL Server from a Web Application | Database Security | Access Permissions for Web Applications | Security Portal

Accessing SQL Server with Explicit Credentials

A reliable way to connect to SQL Server is to pass a user name and password in the connection string. You can use this strategy in different ways:

- Prompt the user for the user name and password.
- Use a predetermined user name and password.

> **Security Note** Using a connection string to access SQL Server is not as secure as using integrated security. Use a connection string with explicit credentials only if it is impractical in your application to use integrated security.

Prompting for Credentials

You can prompt for a user name and password (for example, using Forms authentication), and then pass credentials to SQL Server in the connection string. Prompting for credentials has some disadvantages. It requires that each user be recognized separately by SQL Server. It also is inefficient for connection pooling and thus can limit scalability. For an example of Forms authentication, see Simple Forms Authentication.

> **Security Note** Prompting the user for a username and password in your application is a potential security threat. Credential information must be transferred from the browser to your server, and ordinary HTTP transfers are unencrypted. For best security when prompting users to provide credentials in your application, you should use secure sockets layer (SSL), which encrypts information before sending it from the browser to the server. For details on how to implement SSL, see Microsoft Knowledge Base article Q307267, "HOW TO: Secure XML Web Services with Secure Socket Layer in Windows 2000.

To pass an explicit user name in the connection string dynamically

- After getting the user's name and password, concatenate them into the connection string in code, and then assign them to the ConnectionString property of a connection object.

  > **Security Note** You should be sure that the user name and password do not contain malicious content. The example below illustrates one method to do so. For more information, see Scripting Exploits.

  The following example shows a dynamically created connection string. The user name and password are assumed to be available in text boxes. The user input is filtered with a function that makes sure that it contains only pre-determined characters.

  > **Tip** If you are familiar with using regular expressions, you can create a more efficient version of this filter. For details, see Regular Expression Classes.

```
' Visual Basic
Private Function ValidateInput(ByVal validchars As String, _
        ByVal userinput As String) As Boolean
    Dim c As Char
    For Each c In userinput
        If validchars.IndexOf(c) < 0 Then
            Return False
        End If
    Next
    Return True
End Function

Private Sub OpenDatabase()
    Dim vChars As String = _
        "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz*&%$#!"
    Dim userid As String
    Dim password As String
    If ValidateInput(vChars, UserName.Text) Then
        userid = UserName.Text
    Else
```

```
        Throw New System.Exception("Invalid user name.")
    End If
    If ValidateInput(vChars, PasswordText.Text) Then
        password = PasswordText.Text
    Else
        Throw New System.Exception("Invalid password.")
    End If
    Dim connectionString As String
    connectionString =  "data source=myserver"
    connectionString &= ";initial catalog=northwind"
    connectionString &= ";user id=" & userid
    connectionString &= ";password=" & password
    SqlConnection1.ConnectionString = connectionString
    SqlConnection1.Open()
    ' Further code to work with the database
End Sub

// C#
private Boolean ValidateInput(String validchars, String userinput)
{
    foreach(Char c in userinput)
    {
        if(validchars.IndexOf(c) < 0)
        {
            return false;
        }
    }
    return true;
}

private void OpenDatabase()
{
    String vChars =
       "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz*&%$#!";
    String userid;
    String password;
    if(ValidateInput(vChars, UserName.Text))
    {
        userid = UserName.Text;
    }
    else
    {
        throw new System.Exception("Invalid user name.");
    }
    if(ValidateInput(vChars, PasswordText.Text))
    {
        password = PasswordText.Text;
    }
    else
    {
        throw new System.Exception("Invalid password.");
    }
    String connectionString;
    connectionString =  "data source=myserver";
    connectionString += ";initial catalog=northwind";
    connectionString += ";user id=" + userid;
            connectionString += ";password=" + password;
    sqlConnection1.ConnectionString = connectionString;
    sqlConnection1.Open();
```

```
    // Further code to work with the database
}
```

Using Predetermined Credentials

If it is not practical to prompt for credentials, you can use a predetermined user name and password. The recommended method is to store the predetermined user name and password on the server, and then read it and add it to the connection string at run time. An advantage of this technique is that your application can access the database using different credentials under different circumstances, depending on what it needs to do in the database.

> **Security Note** Never hard-code credentials as strings into programs in your application. Anyone who can get access to the code file, even the compiled code, will be able to get at the credentials.
>
> **Security Note** Always give a predetermined user name the minimal access privileges to a resource. Never use "sa" or any other administrative-level user name. Always use passwords.

You can store credentials in a variety of places. Two options are:

- In the Web.config file. This makes it easy to deploy the configuration information along with the application. Although the configuration file is stored in the same directory as the ASP.NET application, ASP.NET prevents clients from directly accessing files with a .config extension. For additional security, you can set the configuration file's NTFS file system permissions and encrypt the connection string within the file.
- In the Windows registry, as a custom subkey. The credentials are not stored in a Web-accessible directory, which decreases the potential visibility of the string. However, this approach complicates deployment, since installing the application requires changes to the target computer's registry. For additional security, you can use access control lists (ACLs) to restrict access to the registry subkey and encrypt the connection string within the subkey.

> **Note** For further information about storing database credentials, see Database Security.

If you do store credentials as part of an application, you should make an attempt to protect them with encryption. One possibility is to use the Data Protection API (DPAPI). For details, see the MSDN technical article Windows Data Protection.

To store credentials in Web.config

1. In the Web.config file, create a new `<add>` key in the `<appSettings>` element. The `<appSettings>` element must appear as a child of the `<configuration>` element. For details, see Configuration Sections Schema.

   The following example illustrates an <add> key that contains a user name and password, labeled "mappedname" and "mappedkey."

   > **Tip** It is a good idea to avoid using the strings "password" and "user name" (or "username") in your application, so that malicious users searching for those terms will not find them.

   ```
   <appSettings>
      <add key="mappedname" value="MyWebAppUser"/>
      <add key="mappedkey" value="Aje$31"/>
   </appSettings>
   ```

2. In your application, read the credentials from the static AppSettings property of the System.Configuration.ConfigurationSettings class.

   The following example shows how you can read credentials at run time and concatenate them into a connection string:

   ```
   ' Visual Basic
   Dim connectionString As String
   Dim user As String = System.Configuration.ConfigurationSettings. _
       AppSettings("mappedname")
   Dim pass As String = System.Configuration.ConfigurationSettings. _
       AppSettings("mappedkey")
   connectionString =  "data source=myserver;initial catalog=northwind"
   connectionString &= ";user id=" & user
   connectionString &= ";password=" & pass
   ```

```
SqlConnection1.ConnectionString = connectionString

// C#
String connectionString;
String user = System.Configuration.ConfigurationSettings.
    AppSettings["mappedname"];
String pass = System.Configuration.ConfigurationSettings.
    AppSettings["mappedkey"];
connectionString =  "data source=myserver;initial catalog=northwind";
connectionString += ";user id=" + user;
connectionString += ";password=" + pass;
sqlConnection1.ConnectionString = connectionString;
```

To store credentials in the registry

1. Add new subkeys to the **SOFTWARE** key in the **HKEY_LOCAL_MACHINE** hive.
2. In your application, access the subkey using the OpenSubKey method of the Registry.LocalMachine object.

   Note You must import the **System** and **Microsoft.Win32** namespaces in order to use the **Registry** class.

   The following example illustrates how to get a user name and password (stored as string values called "mappedname" and "mappedkey") from the subkey "Connection":

```
' Visual Basic
Dim user, pass, connectionString As String
user = Microsoft.Win32.Registry.LocalMachine.OpenSubKey( _
   "Software\Connection").GetValue("mappedname").ToString()
pass = Microsoft.Win32.Registry.LocalMachine.OpenSubKey( _
   "Software\Connection").GetValue("mappedkey").ToString()
connectionString = "data source=myserver;initial catalog=northwind"
connectionString &= ";user id=" & user
connectionString &= ";password=" & pass
SqlConnection1.ConnectionString = connectionString

// C#
String user, pass, connectionString;
user = Microsoft.Win32.Registry.LocalMachine.
    OpenSubKey("Software\\Connection").
    GetValue("mappedname").ToString();
pass = Microsoft.Win32.Registry.LocalMachine.
    OpenSubKey("Software\\Connection").
    GetValue("mappedkey").ToString();
connectionString = "data source=myserver;initial catalog=northwind";
connectionString += ";user id=" + user;
connectionString += ";password=" + pass;
sqlConnection1.ConnectionString = connectionString;
```

See Also

Accessing SQL Server from a Web Application | Access Permissions for Web Applications | ASP.NET Settings Schema | Database Security | ASP.NET Authentication | Security Portal

Introduction to Web Application Debugging

The ASP.NET page framework provides extensive support for debugging Web applications. However, because Web applications are distributed, there are some special issues associated with debugging them.

In general, you debug Web applications the same way you do other types of Visual Studio applications. You can set breakpoints, start the debugger, break into code, examine variables, and perform all the functions associated with the Visual Studio debugger. For details, see Using the Debugger.

You can debug managed and unmanaged objects, as well as all supported common language runtime languages and script languages. For more information, see Debugging Managed Code. In addition, the ASP.NET page framework provides a trace mode that enables you to insert instrumentation messages into your forms. For details, see Introduction to Instrumentation and Tracing.

Debugger Configuration

Web application debugging requires certain components on the computer where debugging is to happen and requires that you have adequate permissions.

Local Computer Configuration

If you are running Web applications locally on your computer — that is, the Web server is on your computer — then the computer has the correct components automatically.

You must still make sure that you have adequate permissions to debug. When you install Visual Studio or Visual Studio server components on a computer, it creates a Windows group called Debugger Users. Any developer who wants to be able to debug on that computer must be a member of that group. (Having a special group for debugger users allows the server administrator to control who has debugging privileges separately from users who have other permissions for that server.)

If you installed Visual Studio, then your user name is added to the Debugger Users group. Any other users who want to debug — other users, or you logged in under a different profile — must be manually added to the Debugger Users group.

> **Security Note** Membership in the Debugger Users group effectively grants users administrative-level privileges. Do not make anyone a member of this group who should not have this level of privilege.

Remote Computer Configuration

If the Web server is on another computer (a remote server), you must ensure that the computer is configured properly. This involves:

- Making sure that DCOM is installed on both your computer and on the server. Windows 2000 normally has DCOM installed already, so there is usually no action required on your part.
- Installing Visual Studio server-side components on the remote computer. You can do this by running the Visual Studio installation process on the remote computer and selecting the option for server components.
- Adding user permissions by adding user names to the Debugger Users group.
- Ensuring that any debugger users have permissions to attach to a Web server process. This means that server processes must either run as the user (impersonation) or that users who want to debug must have administrative privileges on the Web server. (Giving users administrative privileges on the server might not be in keeping with your security policies.) You can control ASP.NET impersonation using the <identity> setting of the Web.config file for your application. For details, see <identity> Element.

For details about configuring for remote debugging, see Debugging Web Applications on a Remote Server.

Whether you are running locally or on a remote computer, you must be sure that debugging is enabled for your Web application specifically. This is done in the <compilation> element of the Web.config file that is part of your Web application project. This setting instructs the compiler to insert debugging symbols into the application's compiled code so that you can use the debugger with it. For details, see Debug Mode in ASP.NET Applications.

Debugging Client Script

Client script runs within the browser, separately from the code in your Web application, which runs on the server. You can use the Visual Studio debugger to debug client script. The debugger does not allow you to follow execution from server code to client script; however, it does support most other debugging functionality for client script.

You can debug client script in various ways. From within Visual Studio, you can use debugger commands to attach to the browser process (Iexplore.exe) and break into the script. From there, you can use the debugger as you would for any other code.

For details, see Debugging Client-Side Scripts.

See Also

Debugging Script and Web Applications | Debugging

Web Forms Pages

Web Forms is an ASP.NET feature that you can use to create the user interface for your Web applications. Web Forms pages offer you a powerful and straightforward programming model that uses familiar rapid application development (RAD) techniques to build sophisticated Web-enabled user interfaces. The following topics provide background information that you will find useful in learning to work with Web Forms pages.

In This Section

Introduction to Web Forms Pages
Presents a high-level overview of Web Forms pages and describes why Web Forms pages solve problems unique to Web application development.
Creating and Managing Web Forms Pages
Provides instructions on creating and managing Web Forms pages.
Programming Web Forms
Provides information on the specific aspects of programming in a Web Forms environment.
Data Access in Web Forms Pages
Provides a summary of ASP.NET data binding technology in Web Forms pages.

Related Sections

Creating and Programming ASP.NET Server Controls
Lists the controls you can add to a Web Forms page and provides links for more information.
Introduction to ASP.NET Web Applications in Visual Studio
Provides an overview of the components of an ASP.NET Web application, including Web Forms and XML Web services, and how Visual Studio helps you create a Web application.
Decision Chart
Provides a graphical, linked guide to decisions about designing your application.
Web Forms State Management
Describes the approaches you can use in a Web Forms page to maintain page and control state.
ASP.NET Technology Backgrounder
Provides a summary of ASP.NET technology and where Web Forms pages fit into it.
Web Forms Code Model
Describes the object-oriented structure of Web Forms pages — how they derive from a base Page class and how they work.
Migrating to Web Forms
Provides an overview of issues in migrating HTML and ASP pages to Web Forms pages, and links to more information.
ASP.NET Optimization
Links to topics that provide suggestions for getting the best performance from your Web applications.

Introduction to Web Forms Pages

You can use Web Forms pages to create programmable Web pages that serve as the user interface for your Web application. A Web Forms page presents information to the user in any browser or client device and implements application logic using server-side code. Web Forms page output may contain almost any HTTP-capable language, including HTML, XML, WML, and ECMAScript (JScript, JavaScript).

Web Forms pages are:

- Based on Microsoft ASP.NET technology in which code that runs on the server dynamically generates Web page output to the browser or client device. For details about ASP.NET, see ASP.NET Technology Backgrounder.
- Compatible with any browser or mobile device. A Web Forms page automatically renders the correct browser-compliant HTML for features such as styles, layout, and so on. Alternatively, you can design your Web Forms pages to run on a specific browser such as Microsoft Internet Explorer 5 and take advantage of the features of a rich browser client.
- Compatible with any language supported by the .NET common language runtime, including Microsoft Visual Basic, Microsoft Visual C#, and Microsoft JScript .NET.
- Built on the Microsoft .NET Framework. This provides all the benefits of the framework, including a managed environment, type safety, and inheritance.
- Supported in Visual Studio with powerful rapid application development (RAD) tools for designing and programming your forms.
- Extensible using controls that provide RAD capability for Web development, allowing you to create rich user interfaces quickly.
- Flexible because you can add user-created and third party controls to them.

Components of Web Forms

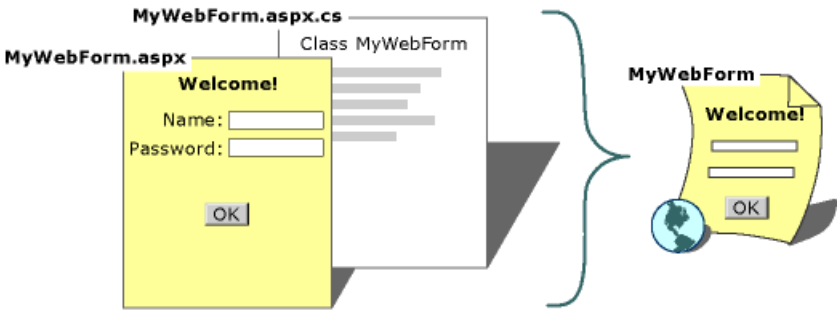
In Web Forms pages, the user interface programming is divided into two distinct pieces: the visual component and the logic. If you have worked with tools like Visual Basic and Visual C++ in the past, you will recognize this division between the visible portion of a form and the code behind the form that interacts with it.

The visual element is referred to as the Web Forms *page*. The page consists of a file containing static HTML, or ASP.NET server controls, or both simultaneously.

The Web Forms page works as a container for the static text and controls you want to display. Using the Visual Studio Web Forms Designer plus ASP.NET server controls, you can design the form as you would in any Visual Studio application. For more information, see Controls You Can Use on Web Forms Pages.

The logic for the Web Forms page consists of code that you create to interact with the form. The programming logic resides in a separate file from the user interface file. This file is referred to as the "code-behind" file and has an ".aspx.vb" or ".aspx.cs" extension. The logic written in the code-behind file can be written in Visual Basic or Visual C#.

Web Forms File Structure



Both files together make up MyWebForm.

The code-behind class files for all Web Forms pages in a project are compiled into the project dynamic-link library (.dll) file. The .aspx page file is also compiled, but somewhat differently. The first time a user browses to the .aspx page, ASP.NET automatically generates a .NET class file that represents the page, and compiles it to a second .dll file. The generated class for the .aspx page inherits from the code-behind class that was compiled into the project .dll file. When a user requests the Web page URL, the .dll files run on the server and dynamically produce the HTML output for your page. For more information, see Web Forms Page Processing.

For more information about how Web Forms pages are constructed, see Web Forms Code Model.

What Web Forms Pages Help You Accomplish

Web application programming presents challenges that do not typically arise when programming traditional client-based applications. Among the challenges are:

- **Implementing a rich Web user interface.** A user interface with a complex layout, a large amount of dynamic content, and full-featured user-interactive objects can be difficult and tedious to design and implement using basic HTML facilities. It is particularly difficult to create a rich user interface for applications likely to run on many different browser and client device platforms.
- **Separation of client and server.** In a Web application, the client (browser) and server are different programs often running on different computers (and even on different operating systems). Consequently, the two halves of the application share very little information; they can communicate, but typically exchange only small chunks of simple information.
- **Stateless execution.** When a Web server receives a request for a page, it finds the page, processes it, sends it to the browser, and then, effectively, discards all page information. If the user requests the same page again, the server repeats the entire sequence, reprocessing the page from scratch. Put another way, a server has no memory of pages that it has processed. Therefore, if an application needs to maintain information about a page, this becomes a problem that has to be solved in application code.
- **Unknown client capabilities.** In many cases, Web applications are accessible to many users using different browsers. Browsers have different capabilities, making it difficult to create an application that will run equally well on all of them.
- **Complications with data access.** Reading from and writing to a data source in a traditional Web applications can be complicated and resource-intensive.
- **Complications with scalability.** In many cases Web applications designed with existing methods fail to meet scalability goals due to the lack of compatibility between the various components of the application. This is often the single failure point for applications under a heavy growth cycle.

Meeting these challenges for Web applications can require substantial time and effort. Web Forms pages and the ASP.NET page framework address these challenges in the following ways:

- **Intuitive, consistent object model.** The ASP.NET page framework presents an object model that allows you to think of your forms as a unit, not as separate client and server pieces. In this model, you can program the form in a more intuitive way than in traditional Web applications, including the ability to set properties for form elements and respond to events. In addition, ASP.NET server controls are an abstraction from the physical contents of an HTML page and from the direct interaction between browser and server. In general, you can use server controls the way you might work with controls in a client application and not have to think about how to create the HTML to present and process the controls and their contents.
- **Event-driven programming model.** Web Forms pages bring to Web applications the familiar model of writing event handlers for events that occur on either the client or server. The ASP.NET page framework abstracts this model in such a way that the underlying mechanism of capturing an event on the client, transmitting it to the server, and calling the appropriate method is all automatic and invisible to the implementer. The result is a clear, easily written code structure that supports event-driven development.
- **Intuitive state management.** The ASP.NET page framework automatically handles the task of maintaining the state of your form and its controls, and it provides you with explicit ways to maintain the state of application-specific information. This is accomplished without heavy use of server resources and can be implemented with or without sending cookies to the browser.
- **Browser-independent applications.** The ASP.NET page framework provides for creating all application logic on the server, eliminating the need to explicitly code for differences in browsers. However, it still allows you to automatically take advantage of browser-specific features by writing client-side code to provide improved performance and a richer client experience.
- **.NET Framework common language runtime support.** The ASP.NET page framework is a technology of ASP.NET. ASP.NET is built on the .NET Framework, so the entire framework is available to any ASP.NET application. Your applications can be authored in any language compatible with the runtime, including Microsoft Visual Basic, Visual C#, and JScript .NET. In addition, data access is simplified using the data access infrastructure provided by the .NET Framework, including ADO.NET.
- **.NET Framework scalable server performance.** The ASP.NET page framework allows you to scale your Web application from one computer with a single processor to a multi-computer "Web farm" cleanly and without complicated changes to the application's logic.

See Also

ASP.NET Technology Backgrounder | Web Forms Code Model | Web Forms Page Processing |
Walkthrough: Creating a Basic Web Forms Page | Migrating to Web Forms | Walkthrough: Creating an Accessible Web Application

Web Forms Code Model

A Web Forms page consists of two parts: visual elements (HTML, server controls, and static text) and the page's programming logic. Visual Studio stores each of these components in a separate file. The visual elements are created in an .aspx, and the code is in a separate class file, called the code-behind class file (.aspx.vb or .aspx.cs).

It is also possible to create the visual elements and code in the same file, which is sometimes called a "single-file" Web Forms page. This style of Web Forms page has limited support in Visual Studio. For details, see "Single-File Web Forms Pages" below.

> **Note** Visual Studio does not support cross-language compiling in Web Forms pages and projects. This means it is not possible to include a Visual C# Web Forms page in a Visual Basic Web project, and vice versa.

The ASP.NET Page Class

Although a Web Forms page consists of two separate files, they form a single unit when your application is run. The code-behind class files for all Web Forms in a project are compiled into the dynamic-link library (.dll) file produced by the project. The Web Forms .aspx page file is also compiled, but somewhat differently. The first time a user browses to the .aspx page, ASP.NET automatically generates a .NET class file that represents the page and compiles it to a second .dll file. The generated class for the .aspx page inherits from the code-behind class that was compiled into the project .dll file.

This single .dll file is run at the server whenever the Web Forms page is requested. At run time, this .dll file processes the incoming request and responds by dynamically creating output and sending it back to the browser or client device.

If the page contains server controls, as is typical, the derived page class acts as a container for the controls. Instances of the controls are created at run time and likewise render output for the browser or client device.

For developers familiar with Active Server Pages (ASP), the ASP.NET page framework model represents something new. The ASP model is one of HTML extended with script code. The ASP page consists of script code, such as ECMAScript (JScript, JavaScript) or VBScript, coexisting with static HTML within the same file. The ASP parser reads the page, interprets it, and runs only the script code to get output results. ASP then merges the output of the script code with the static HTML output found in the page before sending the output back to the browser or client device.

In the ASP.NET **Page** class model, the entire Web Forms page is, in effect, an executable program that generates output to be sent back to the browser or client device. In this model, the page goes through a series of processing stages similar to those of other components — initialize, process, and dispose — with two differences:

- The page class performs these steps each time the page is called. The page is initialized, processed, and disposed of every time a round trip to the server occurs.
- The page class has a unique stage —render— that occurs toward the end of the page life cycle, during which output is generated. For more information, see Control Execution Lifecycle.

> **Note** For efficiency, the information required to recreate the page may be cached, but this action is independent of its life cycle.

Deriving from the Page Class

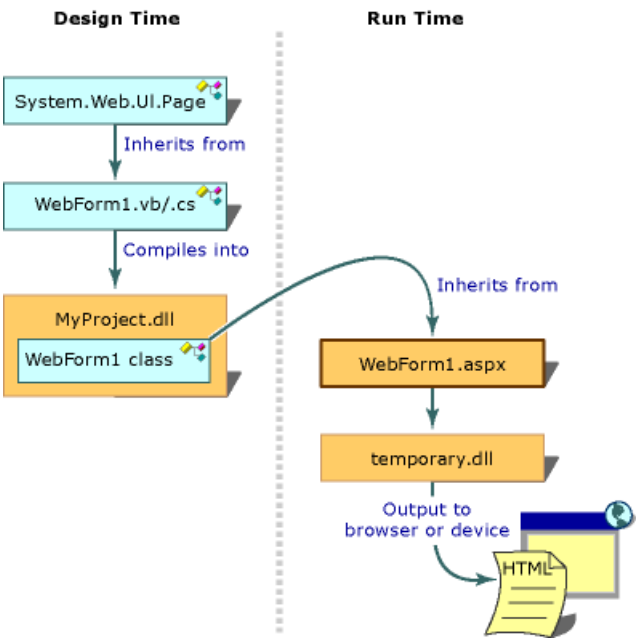
When Visual Studio creates the page and class files for a Web Forms page, it generates code that inherits from the base **Page** class. For example, if you create a new Web Forms page and name it WebPage1, a new class named WebPage1 is derived from **System.Web.UI.Page**.

Depending on whether you have chosen Visual Basic or Visual C# for your development, Visual Studio generates either of the following lines of code:

```
' Visual Basic
Public Class WebForm1
    Inherits System.Web.UI.Page
// C#
public class WebForm1 : System.Web.UI.Page
```

The .aspx page file in turn inherits from the derived WebPage1 class. The relationship of the base **Page** class, the derived class file, and the .aspx file is illustrated in the following diagram.

Web Forms Page Structure and the Page Base Class



Because the .aspx file is compiled dynamically when a user browses the page, its relationship to the class file is established with script directives at the top of the page. In Visual Studio, the relationship between .aspx files and class files is created and maintained automatically, even if you rename the Web Forms page. Specifically, the **Inherits** attribute of the **@ Page** directive is used to specify the class file from which the .aspx file is derived. A typical directive looks like this:

```
<%@ Page Language="vb" AutoEventWireup="false" Codebehind="WebForm1.aspx.vb" Inherits="WebApp
lication1.WebForm1"%>
```

The Codebehind, Inherits, and Src Attributes

In Web Forms that use code-behind files, the @ Page directive (or **@ Control** in user control files) contains attributes that specify the relationship of the .aspx file and its code-behind file. These attribute are:

- **Codebehind** In Visual Studio, this attribute references the name of a file that contains the class for the page. For example, if you create a Web Forms page in Visual Studio called `WebForm1`, the **Codebehind** attribute will point to `WebForm1.aspx.vb` or `WebForm1.aspx.cs`. This attribute is used only by the Visual Studio Web Forms Designer. It tells the designer where to find the page class so that the designer can create an instance of it for you to work with. The attribute is not used at run time.
- **Inherits** Identifies the class from which the page derives. In Visual Studio, this points to a class in the project assembly (.dll), as shown in the diagram above.

The code-behind model illustrated above is the model used by Visual Studio. The ASP.NET Framework supports a slightly different code-behind model for Web Forms pages. In the ASP.NET code-behind model, the visual elements are in an .aspx file and the code is in a separate code-only file, as in Visual Studio. However, there is no project, and the code is not pre-compiled. Instead, the code in the .vb or .cs file is compiled at run time, when the page is first requested by a user.

The inheritance model works as illustrated above, with the difference that the Web Forms class (WebForm1 class in the diagram) is not part of a project assembly. Instead, each page is a separate assembly. There is no difference in how your code runs in the two models.

In the ASP.NET code-behind model, there is no **Codebehind** page attribute, since that attribute is unique to Visual Studio. To tie the .aspx file to its corresponding code, the page directive contains a **Src** attribute, which references the file containing the source code for the file.

The **Src** attribute is not supported in Visual Studio. If you import a Web Forms page into Visual Studio that contains the **Src** attribute, the designer will raise an error. For details, see The file could not be loaded into the Web Forms designer.

Single-File Web Forms Pages

In addition to pages that are made up of an .aspx file and a separate class file, the ASP.NET architecture supports a "single-file" model in which the UI elements and code are in the same file. Single-file Web Forms pages are functionally very similar to pages consisting of two files. For example, you use the same controls on both types of pages. Users still request pages using the same .aspx extension for the file name, and the page is still run using server-side code and it streams HTML to the client. A single-file

page has the advantage that it can be easier to deploy.

There are a few differences in the way single-file pages are processed:

- The code for the page is not compiled into a separate class that the .aspx file then derives from. Instead, the .aspx file simply derives from the **Page** class.
- When the page is deployed, the source code is deployed along with the Web Forms page, since it is physically in the .aspx file. (The user does not see the code — only the results rendered when the page runs are sent to the user.)

You can work with single-file Web Forms pages in Visual Studio, but because Visual Studio is oriented toward the two-file, code-behind model, there is more limited support for single-file pages. Differences include:

- You cannot directly create a single-file Web Forms page in Visual Studio. When you create a new page, by default Visual Studio creates a separate .aspx file and class file. To create a single-file page, you must initially create it as a text file and then change its extension to .aspx.
- You cannot add non-visual components to the page (such as data components) by dragging them from the Toolbox, because the Web Forms Designer will not persist them in the page. Instead, you should add such components in code.
- You write code in HTML view, not in the Code Editor.
- When you are writing code, Intellisense is not supported — you do not get syntax checking or statement completion, tabbing, or code formatting.
- You must bind events to event handlers manually. For single-file Web Forms pages, Visual Studio does not support double-clicking to create the handler for a control's default event, nor the drop-down lists of classes and events in the Code Editor.
- Some debugging features, such as the ability to see a variable value by pointing the mouse at it, are not supported.
- Because the code in the page is not compiled into the project assembly, compile-time errors are not caught until the page runs.

For more information about single-file Web Forms pages, see Working with Single-File Web Forms Pages in Visual Studio .NET.

See Also

Introduction to Web Forms Pages | Web Forms Page Processing | Compilation and Deployment of Web Projects

Web Forms Page Processing

In general, the life cycle for a Web Forms page is similar to that of any Web process that runs on the server. Certain characteristics of Web processing — information passed via HTTP protocol, the stateless nature of Web pages, and so on — apply to Web Forms pages just as they do to most Web applications.

However, the ASP.NET page framework performs many Web application services for you. For example, the ASP.NET page framework captures information posted with the Web Forms page, extracts the relevant values, and makes the information accessible via object properties.

It is important to understand the sequence of events that occurs when a Web Forms page is processed. This knowledge will help you program your Web Forms pages and Web applications more effectively.

The Life Cycle of a Web Forms Page

You might find it helpful to understand some fundamental characteristics of how Web Forms pages work in Web applications before you examine the details of what goes on inside a page when it is processed.

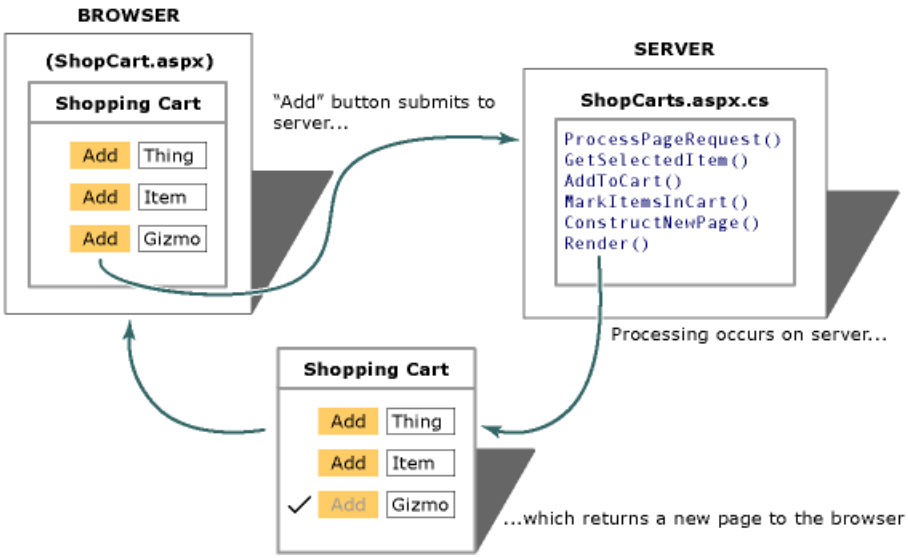
Round Trips

One of the most important things to understand is the division of labor in a Web Forms page. The browser presents the user with a form, and the user interacts with the form, causing the form to post back to the server. However, since all processing that interacts with server components must occur on the server, this means that for each action that requires processing, the form must be posted to the server, processed, and returned to the browser. This sequence of events is referred to as a *round trip*.

> **Note** You can create client script in Web Forms, which is useful for user input validation and for some types of UI programming. Client script does not interact with server components.

Imagine a business scenario: A user enters an order and you want to confirm sufficient inventory for the order, so your application posts the page to the server at an appropriate point in the user's order-entry process. A server process examines the order, performs an inventory lookup, perhaps takes some action defined in business logic (such as modifying the page to indicate an error), and then returns the page to the browser for the user to continue.

Round Trip



In Web Forms, most user actions — such as clicking a button — result in a round trip. For that reason, the events available in ASP.NET server controls are usually limited to click-type events. Most server controls expose a click event, one that requires an explicit user gesture.

By the same token, server controls do not expose high-frequency events such as **onmouseover**, because each time such an event is raised, another round trip to the server would occur, which would considerably affect response time in the form.

Recreating the Page (View State and State Management)

In any Web scenario, pages are recreated with every round trip. As soon as the server finishes processing and sending the page to the browser, it discards the page information. By freeing server resources after each request, a Web application can scale to support hundreds or thousands of simultaneous users. The next time the page is posted, the server starts over in creating and

processing it, and for this reason, Web pages are said to be *stateless* — the values of a page's variables and controls are not preserved on the server.

> **Note** The server can be configured to cache page information to optimize the pages, but for purposes of application programming, it is clearest to think of them as being disposed of as soon as the server has finished processing them.

In a traditional Web application, the only information that the server has about a form is the information that the user has added to the controls on the form, because that information is sent to the server when the form is posted. Other information, such as variable values and property settings, is discarded.

ASP.NET works around these limitations in the following ways:

- It saves page and control properties between round trips. This is referred to as saving the view state of the control.
- It provides state management facilities so you can save your own variable and application-specific or session-specific information between round trips.
- It can detect when a form is requested for the first time versus when the form is posted, which allows you to program accordingly. You may want a different behavior during a page postback versus an initial request.

Benefits of an Event-Driven Model versus a Linear Processing Model

If you have experience using Active Server Pages (ASP), you recognize that ASP is a linear processing model. An ASP page is processed in a top-to-bottom sequence. Each line of ASP code and static HTML is processed in sequence as it appears in the file. User actions cause the page to be posted to the server in a round trip. Since this action causes a round trip, the server must recreate the page. After the page is recreated, it is processed in the same top-to-bottom sequence as before, and therefore the page is not exhibiting truly event-driven behavior. To create an event-driven experience, you need to explicitly design it. In addition, you have to explicitly maintain page and control state at the most basic level. This model limits the richness of the user interfaces that can be assembled, and it increases the complexity of the code needed to support it.

In comparison, an event-driven model, as in a traditional Visual Basic application, contains programmable elements that are initialized and displayed on the form. Users interact with the elements, which cause events to be raised that in turn call event handlers. This model supports true event-driven behavior, which, by design, greatly extends the richness of the user interfaces that can be assembled, and it reduces the complexity of the code needed to support it.

ASP.NET replaces the linear processing model of ASP by emulating the behavior of an event-driven model. The ASP.NET page framework is provided to implicitly make the associations of an event to an event handler for you. Using the page framework allows you to easily create a user interface that reacts to user actions. For details on creating and using events and event handlers, see Server Event Handling in Web Forms Pages.

In addition, this same framework eases the implementation of page and control state management. For more information on state management, see Web Forms State Management.

For example, ASP.NET allows you to set up event handlers in server code for events that are passed from the browser. Assume the user is interacting with a Web Forms page that contains one button server control. The user clicks the button control and an event is raised that is transmitted via an HTTP post to the server where the ASP.NET page framework interprets the posted information and associates the raised event with an appropriate event handler. This event handler can be a default handler supplied by ASP.NET or it can be your custom implementation. The framework automatically calls the appropriate event handler for the button as part of the framework's normal processing. As a result, you no longer need to explicitly design event-like behavior into a linear processing model. For more information on Web Forms event handling, see ASP.NET Server Control Event Model.

Stages in Web Forms Processing

The ASP.NET page framework processes Web Forms pages in distinct stages. During each stage of Web Forms processing, events may be raised, and any event handler that corresponds to the event runs. These methods provide you with entry points — hooks — that allow you to update the contents of the Web Forms page.

The table below lists the most common stages of page processing, the events raised when they occur, and typical uses at each stage. These stages are repeated each time the form is requested or posted. The Page.IsPostBack property allows you to test whether the page is being processed for the first time.

> **Note** There are several more stages of Web Forms page processing than are listed in the following table. However, they are not used for most page processing scenarios. Instead, they are primarily used by server controls on the Web Forms page to initialize and render themselves. If you intend to write your own ASP.NET server controls, you need to understand more about these stages. For details about the complete set of page processing stages, see Developing ASP.NET Server Controls.

Stage	Meaning	Typical uses
ASP.NET Page Framework Initialization	The page's **Page_Init** event is raised, and the page and control view state are restored.	During this event, the ASP.NET page framework restores the control properties and postback data.
User Code Initialization	The page's **Page_Load** event is raised.	Read and restore values stored previously: • Using the **Page.IsPostBack** property, check whether this is the first time the page is being processed. • If this is the first time the page is being processed, perform initial data binding. • Otherwise, restore control values. • Read and update control properties.
Validation	The Validate method of any validator Web server controls is invoked to perform the control's specified validation.	(There is no user hook at this stage. You can test the outcome of validation in an event handler.)
Event Handling	If the page was called in response to a form event, the corresponding event handler in the page is called during this stage.	Perform your application-specific processing: • Handle the specific event raised. **Note** Events are not raised in a particular order, except that cached control events — as specified by the control's **AutoPostBack** property — are always processed before the posting event. • If the page contains Types of Validation for ASP.NET Server Controls, check the **IsValid** property for the page and for individual validation controls. • Manually save the state of page variables that you are maintaining yourself. • Check the IsValid property of the page or of individual validation controls. • Manually save the state of controls dynamically added to the page.
Cleanup	The **Page_Unload** event is called because the page has finished rendering and is ready to be discarded.	Perform final cleanup work: • Closing files. • Closing database connections. • Discarding objects. **Note** It is important that expensive resources, such as database connections, be explicitly closed. Otherwise, they will remain open until the next garbage collection occurs. On a heavily loaded server, many open resources can adversely affect performance.

See Also

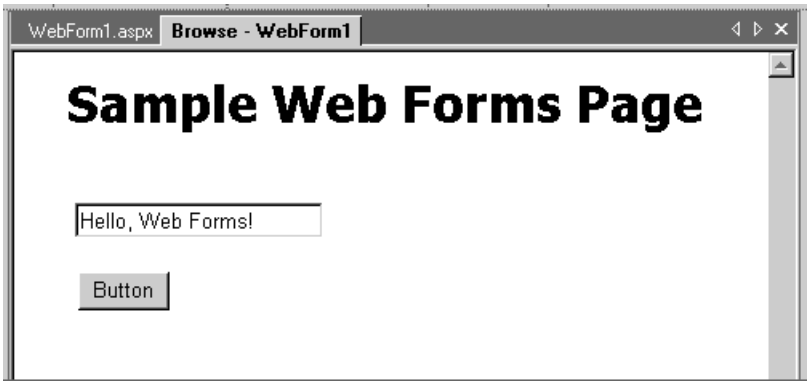
Introduction to Validating User Input in Web Forms | Data Access in Web Forms Pages | ASP.NET Server Control Event Model | Introduction to Web Forms State Management | ASP.NET State Management | Web Forms Pages

Walkthrough: Creating a Basic Web Forms Page

This walkthrough illustrates how to create a Web Forms page. You will see how to add controls to the page and create code for them that will run on the server. Tasks illustrated in this walkthrough include:

- Creating a new Web Forms page.
- Adding controls and static HTML text to the page.
- Creating an event handler for the control.
- Building and running the Web Forms page.

When you are finished, your Web Forms page will look something like the following:



Creating the Project and Form

The first step is to create a Web application and a Web Forms page.

To create the project and form

1. On the **File** menu, point to **New**, then click **Project**.
2. In the **New Project** dialog box, do the following:
   a. In the **Project Types** pane, choose either **Visual Basic Projects** or **Visual C# Projects**.
   b. In the **Templates** pane, choose **ASP.NET Web Application**.
   c. In the **Location** box, enter the complete URL for your application, including http://, the name of the server, and the name of your project. The Web server must have IIS version 5 or later and the .NET Framework installed on it. If you have IIS on your computer, you can specify `http://localhost` for the server. (If you normally use a proxy server to access the Internet, you might need to configure Internet Explorer to bypass the proxy server in order to use localhost.)

   Tip If you already have a solution open, choose **Close Solution** so that your new Web Forms project will be part of its own solution.

   When you click **OK**, a new Web Forms project is created at the root of the Web server you specified. In addition, a new Web Forms page called WebForm1.aspx is displayed in the Web Forms Designer in Design view.

Examining Web Forms Structure

Take a moment to see how Web Forms pages are structured and how the Web Forms Designer is laid out. The Web Forms Designer is open with a file called WebForm1.aspx. A Web Forms page consists of two separate files:

- The .aspx file contains the HTML text and the controls that make up the user interface of the page.
- A separate file, called WebForm1.aspx.vb or WebForm1.aspx.cs (depending on what language you specified for your project), contains the page's code — that is, it is the page's class file. It is sometimes referred to as the "code-behind" file. By default, Solution Explorer does not display the page's class file.

To see a page's class file in Solution Explorer

- Click the **Show All Files** button in the toolbar of Solution Explorer, then expand the node for WebForm1.aspx.

At the bottom of the Web Forms Designer are two tabs, **Design** and **HTML**, which show you different views of the .aspx file you are working with:

- Design view provides you with a WYSIWYG view where you can drag controls and use the Properties window to configure them.
- HTML view shows you the same information, but in the "raw" format of an HTML file. As in HTML files, the Web Forms Designer supports Intellisense for elements in HTML view.

You can work in either view. When you switch between them, each view is updated with changes you made in the other. As you will see later, you use the code editor to write code for the page's class file (.aspx.vb or .aspx.cs).

Adding Controls and Text

You can now add controls and text to the page.

Grid or Flow Mode

By default, the Web Forms page you are working with is in grid layout mode. In this mode, you can drag controls around the page and position them using absolute (x and y) coordinates. (For downlevel browsers, the controls are positioned using tables.)

If you prefer, you can work in flow layout mode, which is like a traditional HTML page — items are positioned from top to bottom. Each view has advantages: grid mode makes it easy to position items. In flow mode, it is easier to add static text. For details, see Positioning HTML Elements in Design View.

For this walkthrough, you will work in grid mode. You will learn how to position static text in this mode.

Adding Controls To Your Web Forms Page

Web Forms controls are called "server controls" because when the page runs, the controls are instantiated in server code as part of the page class. When users interact with the controls — for example, when users click a Web Forms button control — the code associated with the control runs on the server after the page is posted. In server code, you can write event handlers for server controls, set their properties, and so on.

Not every element on the Web Forms page is a server control. For example, by default, static HTML text is not a server control, and you cannot control it from server code. Even standard HTML controls (for example, an HTML submit button) are not server controls by default — they are not visible as first-class controls in server code. (HTML elements are programmable in client script, as in any HTML page.)

Therefore, to work with controls on a Web Forms page, you need to add them as server controls. Server controls come in two types:

- **HTML server controls** These are HTML elements that are marked (that you convert) to be programmable in server code. Typically, you convert HTML elements to HTML server controls only if you have some reason to want to program them from server code.
- **Web server controls** These are controls specific to Web Forms that provide more features than HTML server controls and do not map directly to HTML elements.

For more details about these controls, see Introduction to ASP.NET Server Controls.

In this section, you will add one of each type control.

To add an HTML server control to your Web Forms page

1. Click the Design tab at the bottom to switch to Design view.
2. From the **HTML** tab of the Toolbox, drag a **Text Field** element onto the page.

   Switch to HTML view to see that your action has added a tag similar to the following:

   ```
   <INPUT type="text">
   ```

3. Switch to Design view.
4. Convert the HTML text element to a server control by right-clicking it and choosing **Run As Server Control**.

   A glyph (▶) appears in the upper-left corner of the control to indicate that the control is a server control. Converting an HTML element makes it into an **HtmlInputText** server control.

   Two attributes are added to the HTML element as part of this conversion:

   - The **id** attribute identifies the control in your code. For the **HtmlInputText** server control, this attribute is set by

default to the name **Text1**.

- The **runat** attribute is set to **server**. This marks the element as a server control and makes it visible to the Web Forms server code as a programmable element.

To add a Web server control to your Web Forms page

- From the **Web Forms** tab of the Toolbox (not the **HTML** tab), drag a **Button** Web server control onto the page.

  Tip The **Web Forms** tab is only available when the designer is in Design view.

  This step creates a Web server control element in the designer rather than an HTML button. If you switch to HTML view, you will see the following:

  ```
  <asp:Button id="Button1" runat="server"></asp:Button>
  ```

  This element does not correspond directly to an HTML element. Instead, when the page runs, an instance of the Button Web server control is created and processed. During page processing, the control outputs some HTML elements into the page (in this case, an HTML `<input type=submit>` element).

Adding HTML Text to the Page

In grid layout mode, it is easy to position controls. But what about static HTML text? Because all the elements on the page are positioned with x and y coordinates, you cannot just type text into the page where you want it to appear, as you can in flow layout mode.

The solution is to add an HTML **Label** control in which you can add the static text. You can then position this label (essentially a <DIV> element) anywhere on the page.

To add static text in grid layout mode

1. From the **HTML** tab of the Toolbox, drag a **Label** control onto the page. Position it and size it for the text you want to enter.
2. Click the label to select it, and then click it again. (Do this slowly enough that you are not double-clicking the element.)

   The label is put into text-edit mode, which is marked by a shaded border.

3. Type the static text that you want. For example, type "Sample Web Forms Page".
4. Select the text and then use the tools on the Formatting toolbar to set the text's block format, font, size, and so on.
5. Click away from the label to leave text-edit mode.

Creating an Event Handler

Server controls on Web Forms pages can raise various events. Many events are triggered by something the user does in the browser. For example, a **Button** Web server control can raise a **Click** event when a user clicks a button on your page.

The code to handle the raised event is executed on the server. When the user clicks a button, the page is posted back to the server. The ASP.NET page framework parses the event information, and if you have an event handler corresponding to the event, your code is called automatically. When your code finishes, the page is sent back to the browser with any changes as a result of the event handler code.

To create an event handler for the Button Web server control

1. Double-click the **Button** Web server control.

   The designer opens the class file for the current form and creates a skeleton event handler for the button control's **Click** event. The code will look like this:

   ```
   ' Visual Basic
   Private Sub Button1_Click(ByVal sender as System.Object, ByVal e as System.EventArgs) Han
   dles Button1.Click

   End Sub

   // C#
   private void Button1_Click(object sender, System.EventArgs e)
   ```

```
{

}
```

2. Write code in the method to display a message in the **HtmlInputText** server control by setting its **Value** property. For example:

```
' Visual Basic
Private Sub Button1_Click(ByVal sender as System.Object, ByVal e as System.EventArgs) Han
dles Button1.Click
   Text1.Value = "Hello, Web Forms!"
End Sub

// C#
private void Button1_Click(object sender, System.EventArgs e)
{
      Text1.Value = "Hello, Web Forms!";
}
```

Building and Running the Web Forms Page

Before you run your Web Forms page, you must compile the page's class file. You can then view the page in any browser.

To build and run the page

1. In Solution Explorer, right-click the WebForm1.aspx page, and then select **View in Browser**.

   Visual Studio compiles the page and displays it in the **Browse tab**.

2. Click the button on your Web Forms page.

   The text "Hello, Web Forms!" appears in the text box.

3. To stop running the form and return to design mode, close the **Browse** tab.

Next Steps

This is a simple Web Forms page, but it illustrates the fundamental steps that you use to create and edit Web Forms. From this start, you can begin to explore the tools provided in Visual Studio to help you create sophisticated Web applications. Suggestions for more exploration include:

- Investigate the types of controls you can use on Web Forms pages. For details, see Introduction to ASP.NET Server Controls.
- Experiment with working in flow layout mode instead of grid layout mode. For details, see Positioning HTML Elements in Design View.
- Check user input into your Web Forms page using validation controls. For details, see Introduction to Validating User Input in Web Forms.
- Explore the use of more complex controls such as the **Calendar** Web server control. For details, see Calendar Web Server Control.
- Learn about the security issues associated with Web Forms. For details, see Overview of Web Application Security Threats.
- Bind the form to a data source and display data in individual controls or in complex controls on the form. For more information, see Data Access in Web Forms Pages. Controls such as the **DataList** and the **DataGrid** are provided to ease the implementation of including data-driven user interfaces. For more information, see Introduction to the DataList Web Server Control and Introduction to the DataGrid Web Server Control.

See Also

Introduction to Web Forms Pages | Creating and Managing Web Forms Pages | ASP.NET Server Controls | Button Web Server Controls

Migrating to Web Forms

If you have used earlier versions of Microsoft Visual Studio or individual products such as Visual Basic or Visual InterDev, you may have existing Web applications based on previous Web programming technology. Migrating these Web applications to the Web Forms code model allows you to take advantage of all the extended features of ASP.NET.

In This Section

Migrating ASP Pages to Web Forms Pages
Provides instruction on how to migrate an Active Server Pages (ASP) page to an ASP.NET Web Forms page.
Migrating HTML Pages to Web Forms Pages
Provides instruction on how to migrate an HTML page to an ASP.NET Web Forms page.

Related Sections

Web Forms Pages
Provides an introduction to Web Forms technology.
ASP.NET Technology Backgrounder
Provides a summary of ASP.NET technology and where Web Forms pages fit into it.
Comparison of ADO.NET and ADO
Provides a side-by-side comparison of ADO and ADO.NET data-access technologies.
Introduction to Web Forms Pages
Provides an overview of the essential components of Web Forms pages.
ASP.NET Server Controls
Provides an introduction to the control technologies available for use on Web Forms pages.
Migrating ASP Pages to ASP.NET
Provides information on issues concerning migration from ASP to ASP.NET.
Upgrading WebClass Projects to Visual Basic .NET
Explains how to upgrade WebClass projects (also known as IIS Applications) to ASP.NET Web Application projects.

Migrating ASP Pages to Web Forms Pages

You can work with and run existing ASP pages (.asp files) as-is in Visual Studio. You can use ASP pages and ASP.NET pages in the same project. You can edit ASP files in the Web Forms Designer and run them as you would other pages in your project.

> **Note** Some ASP and Visual InterDev 6.0 features are not supported by ASP.NET, because these features require a different run-time environment. For more information, see Migrating ASP Pages to ASP.NET.

However, you might find it useful to convert ASP pages to ASP.NET Web Forms pages so that you can take advantage of the enhanced features of the newer architecture, including the benefits of using ASP.NET server controls and better performance and scaling.

Migrating an ASP page to a Web Forms page accomplishes the following:

- Creates the proper file extension for the page.
- At your option, generates a code-behind class file for the page.
- Corrects certain HTML syntax that was allowed in ASP files but is not allowed in Web Forms pages. For example, the Web Forms Designer will put quotation marks around any ASP code used to set attribute values in an element.
- Adds the appropriate page directives to the page so that it is linked with the code-behind file. For details, see @ Page.

The migration does not do the following:

- Convert inline ASP code (delimited with <% and %>) to code in the class file. However, the existing ASP code will work properly when the page runs.
- Convert VBScript code to Visual Basic .NET. You will need to fix errors that arise out of syntax problems, as documented below.
- Convert Visual InterDev design-time controls (DTCs) to Web Forms controls. If you have a DTC on the page, Visual Studio substitutes a placeholder. The DTC will not operate at run time. It is recommended that you replace DTCs with ASP.NET server controls.
- Convert existing HTML elements to HTML server controls or Web server controls. You can perform this step manually after the migration, as noted below.
- Configure the page to be able to run COM+ 1.0 components. You can enable COM interoperability by setting the ASPCompat attribute of the page's @ Page directive. For resources that discuss COM interoperability, see "Additional Information" below.
- Convert ADO data access code to ADO.NET. Using ADO in a Web Forms page requires COM interoperability. (See previous point.) For resources, see "Additional Information" below.

When the migration is complete, you can run the page normally. Generally, however, after performing the migration you will want to do further work on the page, such as substituting ASP.NET server controls for existing HTML elements.

To migrate an ASP page into a Web Forms project

1. In Solution Explorer right-click the project, point to **Add**, then click **Add Existing Item**.
2. In the **Add Existing Item** dialog box, find the ASP page to migrate, then choose **Open**.

   > **Tip** In the **Files of type** box, select **Web Files** to view all Web files.

   When you migrate the file, it is physically copied from the file's original location to the location of your Web Forms project, and it appears in Solution Explorer.

3. In Solution Explorer, right-click the file name, choose **Rename**, and change the file name extension to ".aspx".

   This causes two confirmation dialog boxes to appear, one after the other.

4. Do the following:
   a. The first dialog box notes that changing the file name extension may invalidate the file format for use with some applications. Click **Yes.**
   b. The second dialog box indicates that there is no code-behind class file associated with the new .aspx file. Click **Yes** to add the code-behind class file.

      > **Tip** You can click the **Show All Files** button in Solution Explorer to see all the files associated with this project. If you expand the tree below your newly migrated file, you should see the code-behind class file.

5. Open the new .aspx file.

   By default, the file opens in Design view. If there are syntax errors on the page, the designer will open the file in HTML view. In that case:

   a. Try switching to Design view. It will not be displayed; instead you will see a message indicating syntax errors.
   b. Fix the errors and try switching to Design view again.

   > **Note** By default, switching between views causes the Web Forms Designer to apply automatic formatting. This can fix some syntax errors, such as missing quotation marks. For details, see Format, HTML/XML, Text Editor, Options Dialog Box.

6. In HTML view, inspect the file to ensure that the file contains an HTML `<form>` element within the HTML `<body>` tags. If not, add a `<form>` element. In the opening tag, add the `Runat = "server"` attribute so that it looks similar to the following:

```
<form id="Form1" method="post" runat="server">
```

7. Determine whether the file contains calls to COM components — for example, ADO code. If so, add the **AspCompat** attribute to the page directive in HTML view:

```
<%@ Page [...] AutoEventWireup="false" AspCompat="true" %>
```

8. Save the form, compile the project, and view the page in the browser. For details see Compiling and Running Web Forms Pages.
9. Check for issues that appear in the Task List window and resolve them. For more information see Task List Window.
10. Repeat Steps 8 and 9 until your form is working properly.

At this point, you will not have any server controls on the page. However, you can convert existing HTML controls to HTML server controls. For details, see Converting HTML Server Controls to HTML Elements.

You only need to migrate the controls that you want to be accessible to server code during run time. The HTML controls that you do not migrate will behave as before.

> **Note** If you wish to add additional ASP.NET server controls to your newly migrated Web Forms page, you must close the file and open it again for the Web Forms Designer to properly initialize the Web Forms tab of the Toolbox. You may then drag server controls to your Web Forms page.

Additional Information

This topic offers only an overview of how to convert ASP pages to Web Forms pages. For more information, including language issues, COM compatibility, and more, see the following sources:

- The technical article Migrating to ASP.NET: Key Considerations
- The technical article Converting ASP to ASP.NET.

See Also

Migrating to Web Forms | Introduction to ASP.NET Server Controls | Creating and Managing Web Forms Pages

Migrating HTML Pages to Web Forms Pages

You can work with and run existing HTML pages as-is in Visual Studio. You can edit the files in the HTML Designer and run them as you would other pages in your project. However, if the HTML page includes a form or if it relies on browser-sensitive client script, you might find it useful to convert the page to an ASP.NET Web Forms page.

Migrating an ASP page to a Web Forms page accomplishes the following:

- Creates the proper Web Forms file extension for the page.
- At your option, generates a code-behind class file for the page.
- Adds the appropriate page directives to the page so that it is linked with the code-behind file. For details, see @ Page.

The migration does not do the following:

- Convert existing HTML elements to HTML server controls or Web server controls. You can perform this step manually after the migration.
- Convert client script (in JavaScript or VBScript) to Web Forms ASP.NET code. Client script will run without modification in Web Forms pages, since it runs entirely in the browser. However, you might want to reimplement the same functionality in Web Forms server code instead so that there are no browser dependencies in the page.

When the migration is complete, you can run the page normally. Generally, however, after performing the migration you will want to do further work on the page, such as converting existing HTML elements to ASP.NET server controls.

To migrate an HTML page into a Web Forms project

1. In Solution Explorer, right-click the project, point to **Add**, then click **Add Existing Item**.
2. In the **Add Existing Item** dialog box, find the HTML page to migrate, then choose **Open**.

   > **Tip** In the **Files of type** box, select **Common Web Files**.

   When you migrate the file, it is physically copied from the file's original location to the location of your Web Forms project, and it appears in Solution Explorer.

3. In Solution Explorer, right-click the file name, choose **Rename**, and change the file name extension to ".aspx".

   This causes two confirmation dialog boxes to appear, one after the other.

4. Do the following:
   a. The first dialog box notes that changing the file name extension may invalidate the file format for use with some applications. Click **Yes.**
   b. The second dialog box indicates that there is no code-behind class file associated with the new .aspx file. Click **Yes** to add the code-behind class file.

   > **Tip** You can click the **Show All Files** button in Solution Explorer to see all the files associated with this project. If you expand the tree below your newly migrated file, you should see the code-behind class file.

5. Open the new .aspx file.

   By default, the file will open in Design view. If there are syntax errors on the page, the designer will open the file in HTML view. In that case:

   a. Try switching to Design view. It will not be displayed; instead you will see a message indicating syntax errors.
   b. Fix the errors and try switching to Design view again.

   > **Note** By default, switching between views causes the Web Forms Designer to apply automatic formatting. This can fix some syntax errors, such as missing quotation marks. For details, see Format, HTML/XML, Text Editor, Options Dialog Box.

6. In HTML view, inspect the file to ensure that the file contains an HTML `<form>` element within the HTML `<body>` tags. If not, add a `<form>` tag one and in the opening form tag, set the attribute `Runat = "server"` attribute so that it looks similar to the following:

```
<form id="Form1" method="post" runat="server">
```

7. Save the form, compile the project, and view it in the browser. For details see Compiling and Running Web Forms Pages.
8. Check for issues that appear in the Task List window and resolve them. For more information see Task List Window.
9. Repeat Steps 7 and 8 until your form is working properly.

At this point, you will not have any server controls on the page. However, you can convert existing HTML control to HTML server controls. For details, see Converting HTML Server Controls to HTML Elements.

You only need to migrate the controls that you want to be accessible to server code during run time. The HTML controls that you do not migrate will behave as before. For more information, see Adding HTML Server Controls to a Web Forms Page.

> **Note** If you wish to add additional Web server controls to your newly migrated Web Forms page, you must close the file, and open it again for the Web Forms Designer to properly initialize the **Web Forms** tab of the Toolbox. You may then drag Web server controls to your Web Forms page. For more information, see Introduction to ASP.NET Server Controls.

See Also

Migrating to Web Forms | Introduction to ASP.NET Server Controls | Creating and Managing Web Forms Pages

Walkthrough: Creating an Accessible Web Application

Creating accessible Web pages allows you to reach as many customers as possible. People with disabilities are not the only users that appreciate accessible pages. Users with slow connections or text-only browsers also rely on accessibility options. Accessible design allows automation tools such as search engines to search, index, and manipulate the information on your pages. Finally, future telecommunications laws may require information distributed over the Internet to be accessible in the same manner as traditional software.

For more information, see Accessibility Design Guidelines for the Web.

From Microsoft's Accessibility Web site, here are the "12 Tips for Designing Accessible Web Pages":

- Good alternative(ALT) text for all graphics.
- Proper use of image maps.
- Useful link text.
- Good keyboard navigation.
- Alternates to all controls and applets.
- Alternate pages that do not use frames.
- Proper use of tables and their alternatives.
- Support the reader's formatting options.
- Do not require the use of style sheets.
- File formats the reader can use.
- Avoid using scrolling marquees.
- Provide titles for most objects.

If you are unable to meet accessibility goals, consider providing alternate text-only Web pages.

Planning Ahead for Accessibility

Controls on the standard toolbox for Web Forms support many of the accessibility guidelines, including exposing the keyboard focus and exposing the screen elements. The controls' properties can be used to support other accessibility guidelines as shown in the following table.

Control Property	**Accessibility Considerations**
TabIndex	Use the **TabIndex** property to create a sensible navigational path through the form. It is important for controls without intrinsic labels, such as text boxes, to have their associated label immediately precede them in the tab order.
Text	Use the HTML <u> tag to display the access key for a control. Using access keys contributes to providing documented keyboard access to all features. (Use the **AccessKey** property to enable an access key for a control.)
Font Size	Use heading tags rather than specific sizes.
Alternate Text	Supply alternate text for all images.
AccessKey	Use this property to provide keyboard access to controls.

Creating the Project

This walkthrough creates a Web site for an algebra class. The user interface consists of an **Image** for the class logo, a **Panel** with a few **HyperLinks** for lectures, a **DataGrid** to display a table containing the class syllabus, and a **HyperLink** to a page describing the class textbook.

The focus of this walkthrough is the accessibility of the user interface, and how to code against the Web Server controls to support accessibility. It demonstrates the accessibility features of several frequently used controls, including the **Link**, **DataGrid**, **Image**, and **Label** controls. The **HyperLink**s will all navigate to the same Web page, but that Web page will be blank.

To begin making the application

1. Create a new ASP.NET Web Application in either Visual Basic or Visual C#. Name the project **AlgebraClass**. (For details see Creating New Solutions and Projects.)

   The start-up form, WebForm1.aspx, appears in the Web Forms Designer.

2. On the **Project** menu, click **Add Web Form**. The **Add New Item** dialog box appears. Name the new Web Form Lecture.aspx and click **Open**. This form will be used to provide the **NavigationUrl** for the **HyperLink** examples on WebForm1.
3. Close the form named Lecture.aspx, saving the changes.

Adding the Controls to the Form

When adding the controls to the form for this application, keep in mind the following guidelines in order to make an accessible application:

- When using tables, use the TITLE attribute to provide names for the columns and rows of a table. Also, make sure that tables make sense if read left-to-right, top-to-bottom.
- Use real heading tags, for example, H1, H2, and so on, rather than formatted text, to support the user's selected formatting options.
- Provide useful link text. For example, if the text is "Click here for Lecture 1 notes", "Lecture 1 notes" is more useful link text than "Click here".
- Provide a sensible navigational path through the page, following the normal flow of text for the given language.
- Use the TITLE attribute for most objects.

To add accessible controls to your page

1. In the Properties window, select **Document**, and set the following properties:

   - Set the **title** property to "Algebra Class"
   - Set the **bgColor** to the **Background** color of the **System Colors** set.

     To do this, click the ellipses button (...) to open the **Color Picker** dialog box, choose the **System Colors** tab, and click **Background**.

   - Seth the **pageLayout** property to **GridLayout**.

2. Add the controls to the form from the Web Forms tab of the Toolbox. In the Properties window, set the properties as described below. See the picture at the end of the table for a model of how to arrange the controls on the form. For more information about adding controls to Web Forms pages, see Adding Web Server Controls to a Web Forms Page.

   To correctly position the **HyperLink** controls on the **Panel** control, you will need to select the **Panel** control and then position your pointer in the **Panel**. The **HyperLink** will be added at the pointer position. You will need to add a carriage return at the end of the first hyperlink to vertically align the second hyperlink.

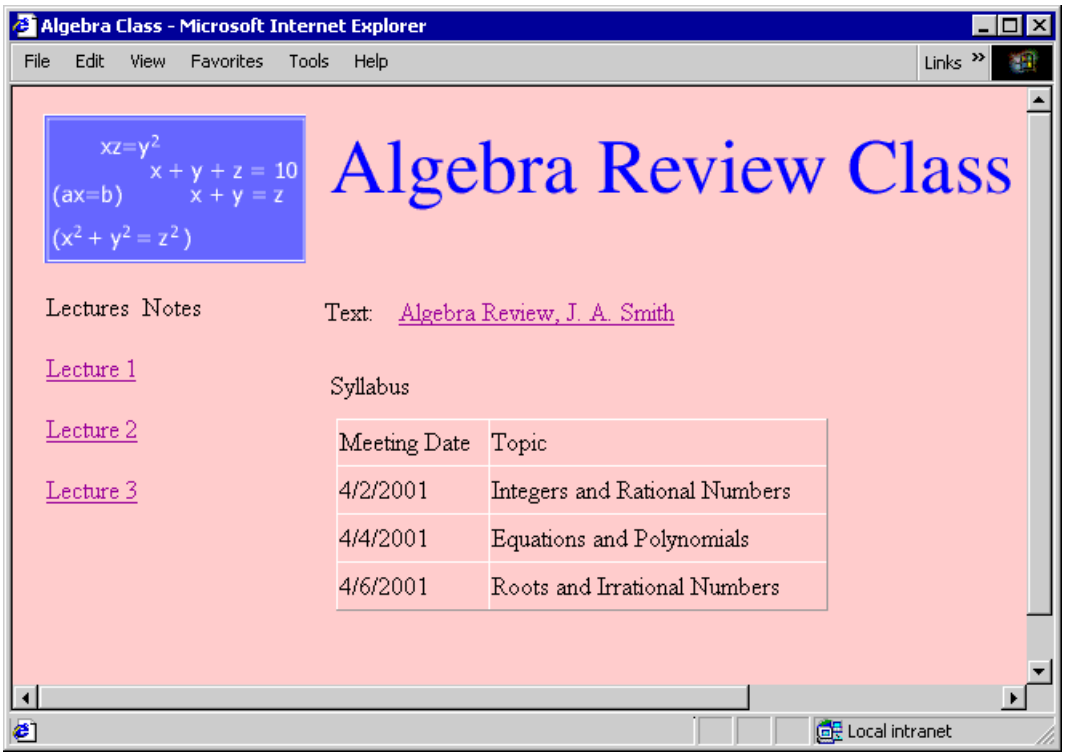
Object	Property	Value
Image	AlternateText	Algebra equations.
	ImageUrl	Any bitmap
	TabIndex	0
Label	ID	ClassName
	ForeColor	"Control" on the **System** Tab of the color palette. Click (...) to access the color palette.
	Text	Algebra Review Class
	Font Size	XX-Large
	TabIndex	1
Panel	ID	Lectures
	TabIndex	2
HyperLink (add to Lectures panel)	ID	Lecture1
	Text	Lecture 1
	AccessKey	1
	NavigateUrl	Lecture.aspx
	TabIndex	3
HyperLink (add to Lectures panel)	ID	Lecture2
	Text	Lecture 2

	AccessKey	2
	NavigateUrl	Lecture.aspx
	TabIndex	4
HyperLink (add to Lectures panel)	ID	Lecture3
	Text	Lecture 3
	AccessKey	3
	NavigateUrl	Lecture.aspx
	TabIndex	5
Label	ID	TextbookLabel
	Text	Text:
	TabIndex	6
Hyperlink	ID	TextbookLink
	Text	Algebra Review, J. A. Smith
	NavigateUrl	Lecture.aspx
	TabIndex	7
Label	ID	SyllabusTitle
	Text	Syllabus
	TabIndex	8
DataGrid	ID	SyllabusGrid
	TabIndex	9

3. In the designer, select the **Panel** control. In the designer, change the text of the panel to **Lecture Notes**.
4. In the designer, click the HTML tab. Modify the HTML tag to indicate that the language of the page is English.

```
<HTML lang="en">
```

Your form will look something like the following at run time:



To add functionality to the form and its controls

1. Add the following function to create data for the **DataGrid**. In most applications, you would retrieve your data from a database, but for this application, the following method is sufficient.

```
' Visual Basic
Function CreateSyllabusEntries() As DataTable
   Dim syllabus As New DataTable()
```

```
   syllabus.Columns.Add("Meeting Date", GetType(String))
   syllabus.Columns.Add("Topic", GetType(String))
   syllabus.Rows.Add(New Object() {"4/2/2001", _
      "Integers and Rational Numbers"})
   syllabus.Rows.Add(New Object() {"4/4/2001", _
      "Equations and Polynomials"})
   syllabus.Rows.Add(New Object() {"4/6/2001", _
      "Roots and Irrational Numbers"})
   Return syllabus
End Function

// C#
protected DataTable CreateSyllabusEntries()
{
   DataTable syllabus = new DataTable();
   syllabus.Columns.Add("Meeting Date",
      System.Type.GetType("System.String"));
   syllabus.Columns.Add("Topic", System.Type.GetType("System.String"));
   syllabus.Rows.Add(new object[] { "4/2/2001",
      "Integers and Rational Numbers" } );
   syllabus.Rows.Add(new object[] { "4/6/2001",
      "Equations and Polynomials" } );
   syllabus.Rows.Add(new object[] { "4/8/2001",
      "Roots and Irrational Numbers" } );
   return syllabus;
}
```

2. Edit the `Page_Load` method to bind the syllabus data to the **DataGrid**.

```
' Visual Basic
Private Sub Page_Load(ByVal sender As System.Object, _
ByVal e As System.EventArgs) Handles MyBase.Load
   Dim syllabus As DataTable = CreateSyllabusEntries()
   SyllabusGrid.DataSource = syllabus
   SyllabusGrid.DataBind()
End Sub

// C#
private void Page_Load(object sender, System.EventArgs e)
{
   DataTable syllabus = CreateSyllabusEntries();
   SyllabusGrid.DataSource = syllabus;
   SyllabusGrid.DataBind();
}
```

3. Use the **ItemCreated** event of the DataGrid to add TITLE attributes to the table cells created by the DataGrid. At a minimum, you should set the TITLE for the column headers. You may also want to set the TITLE for rows. For information on event handlers, see Creating Event Handlers in Web Forms Pages.

```
' Visual Basic
Private Sub SyllabusGrid_ItemCreated(ByVal sender As Object, _
ByVal e As System.Web.UI.WebControls.DataGridItemEventArgs) _
Handles SyllabusGrid.ItemCreated
   Dim cells As TableCellCollection = e.Item.Cells
   If e.Item.ItemType = ListItemType.Header Then
      ' Add TITLE attributes to the column headers.
      cells.Item(0).Attributes.Item("TITLE") = "Date"
      cells.Item(1).Attributes.Item("TITLE") = "Topic"
   ElseIf (e.Item.ItemType = ListItemType.Item) _
```

```
      Or (e.Item.ItemType = ListItemType.AlternatingItem) Then
      ' Add descriptive titles to individual cells.
      Dim rowView As DataRowView = _
         CType(e.Item.DataItem, DataRowView)
      cells.Item(0).Attributes.Item("TITLE") = _
         rowView.Row.ItemArray(0).ToString()
      cells.Item(1).Attributes.Item("TITLE") = _
         rowView.Row.ItemArray(1).ToString()
   End If
End Sub

// C#
private void SyllabusGrid_ItemCreated(object sender, System.Web.UI.WebControls.DataGridIt
emEventArgs e)
{
   TableCellCollection cells = e.Item.Cells;
   if (e.Item.ItemType == ListItemType.Header)
   {
      // Add TITLE attributes to the column headers.
      cells[0].Attributes["TITLE"] = "Date";
      cells[1].Attributes["TITLE"] = "Topic";
   }
   else if ((e.Item.ItemType == ListItemType.Item) ||
      (e.Item.ItemType == ListItemType.AlternatingItem))
   {
      // Add descriptive titles to individual cells.
      DataRowView rowView = (DataRowView)e.Item.DataItem;
      cells[0].Attributes["TITLE"] =
         rowView.Row.ItemArray[0].ToString();
      cells[1].Attributes["TITLE"] =
         rowView.Row.ItemArray[1].ToString();
   }
}
```

4. Run the application.

Testing the Application

Use the following checklist to test the accessibility of the application.

- Test the access keys. For this application, you defined the access keys ALT+1, ALT+2, and ALT+3 for the **HyperLink** controls. The access keys will take you to the links. Press ENTER to follow the links.
- Turn off graphics to make sure that the ALT text displayed makes the page usable. In Internet Explorer version 6, click the Tools menu and choose the Internet Options command. In the Advanced tab, turn off the graphics options in the Multimedia section. You must refresh the page to see the ALT text.
- Turn off sounds to make sure that no important instructions are lost. In Internet Explorer, version 6, click the Tools menu and choose the Internet Options command. In the Advanced tab, turn off the graphics options in the Multimedia section.
- View the application in a browser that allows you to turn off style sheets. Turn off the style sheets to make sure the page is still readable.
- Use the High Contrast option and make sure the page is still readable. To test support of High Contrast, choose the **Accessibility Options** icon in **Control Panel**. Click the **Display** tab and select the **Use High Contrast** check box. Navigate through all user interface elements to ensure that the color and font changes are reflected. Also, ensure that images or patterns drawn behind text are omitted.

  > **Note** Windows NT 4 does not have an **Accessibility Options** icon in **Control Panel**. Thus, this procedure for changing the **SystemInformation.HighContrast** setting does not work in Windows NT 4.

- Use custom font sizes in the Display section of the Control Panel.
- Use the largest font size supported (available only when High Contrast is selected) to make sure the page is still readable.

- Resize the browser window and check the readability.
- Navigate using the keyboard to make sure the navigational order is sensible, that the TAB key traverses all links, and that CTRL+TAB moves between panes or sections.
- Select all text and copy it to the clipboard to make sure that it makes sense.
- See the Bobby Web site for an analysis tool you can use to examine your pages (http://www.cast.org/bobby/).

See Also

Accessibility Design Guidelines for the Web

Creating and Managing Web Forms Pages

The following topics provide information about how to create and work with Web Forms in your Web-based applications.

In This Section

Web Forms Pages and Projects in Visual Studio
Describes the Web projects you can create in Visual Studio .NET for working with Web Forms.
Adding Web Forms Pages to a Web Project
Provides details on creating new Web Forms pages and adding existing Web Forms pages to a project.
Compilation and Deployment of Web Projects
Provides an overview on how Web projects are compiled and deployed.
Compiling and Running Web Forms Pages
Provides details on building and running a Web Form.

Related Sections

Web Forms Pages
Presents a high-level overview of Web Forms features in Visual Studio and provides links for more information.
Creating Web Projects
Provides details on starting a new project, either a full application with starter files included or an empty Web project.
ASP.NET Server Controls
Provides an overview of the types of controls you can use on a Web Forms page.
Editing HTML
Provides details on adding HTML elements to your page and manipulating them.
Creating and Programming ASP.NET Server Controls
Provides details on how to add controls to a Web Forms page, set control properties, and create event-handling methods.
Working with Web Projects
Provides details on project management when making your ASP.NET Web application.

Web Forms Pages and Projects in Visual Studio

Web Forms pages are one way to create the user interface of an ASP.NET-based Web application. You can add Web Forms pages to several types of Visual Studio projects. Most often, when you want to work with Web Forms pages, you will use the ASP.NET Web Application project template. For more information on Web project templates, see Introduction to Web Projects. For information on ASP.NET Web applications as a whole, see Introduction to ASP.NET Web Applications in Visual Studio.

To work with a Web Forms page in Visual Studio, you use the Web Forms Designer. The designer includes a WYSIWYG view, called Design view, for laying out the elements of the page. Alternatively, you can switch the designer to HTML view, which gives you direct access to the ASP.NET syntax of the elements on the page. Finally, the designer includes a code editor with Intellisense that you can use to create the page initialization and event handler code for your page. For more information, see Introduction to Web Forms Pages.

Project Files Created With Web Forms

When you create a Web project, Visual Studio constructs a Web application directory structure on the target Web server, and a project structure on your local computer. The files that relate to your Web Forms pages are as follows:

> **Note** You need to have appropriate privileges on the Web server computer to create and manage Web application files. The correct way to get those privileges is to be a member of the VS Developers group that is automatically created on the Web server. For details, see Web Application Security at Design Time in Visual Studio.

File Created	Description
WebForm1.aspx and WebForm1.aspx.vb files (or WebForm1.aspx.cs for Visual C# Web applications)	These two files make up a single Web Forms page. The .aspx file contains the visual elements of the Web Forms page, for example HTML elements and Web Forms controls. The WebForm1.aspx.vb class file is a hidden, dependent file of WebForm1.aspx. It contains the code-behind class for the Web Forms page, containing event-handler code and the like. For more information see Web Forms Code Model.
AssemblyInfo.vb (for Visual Basic projects) or AssemblyInfo.cs (for C# projects)	A project information file (AssemblyInfo.vb or AssemblyInfo.cs file) that contains metadata about the assemblies in a project, such as name, version, and culture information. For details on data contained in the assembly file, see Assembly Manifest.
Web.config	An XML-based file that contains configuration data on each unique URL resource used in the project. For more information see Application Configuration Files.
Global.asax and Global.vb or Global.cs files	An optional file for handling application-level events. This file resides in the root directory of an ASP.NET application. The Global.vb class file is a hidden, dependent file of Global.asax. It contains the code for handling application events such as the **Application_OnError** event. At run time, this file is parsed and compiled. For more information on Global.asax file usage, see Global.asax File.
Styles.css	A cascading style sheet file to be used within the project. For more information see Introduction to Cascading Style Sheets.

Deployment

The server on which you create your Web Forms pages is typically not the server on which you will deploy the pages and their associated files for production. To deploy a Web project containing Web Forms pages, you may follow one of two approaches:

- Copy your project to the Web server
- Create a deployment project

For more information on these options, see Deployment of a Web Setup Project.

See Also

Working with Web Projects | Using Solution Explorer | Web Access Methods | Introduction to Solutions, Projects, and Items |

Adding Web Forms Pages to a Web Project

After you have started a Web project, you can create new Web Forms pages and add existing Web Forms pages to the project. For instructions on creating an ASP.NET Web application, see Creating Web Projects.

> **Note** You can add a Web Forms page to an existing Web project or add a page from another project. However, all pages in a single project must use the same programming language. For example, you cannot add a Visual C# Web Forms page to a Visual Basic project or vice versa.

Visual Studio enables shared working arrangements. A development team could split the development of new Web Forms pages between members, write code individually in separate projects and, using source control, add them to a single project. For more information on source control, see Source Control Basics.

When you create a Web Forms page, Visual Studio performs the necessary operations to include this file in your project. For an overview, see Web Forms Pages and Projects in Visual Studio.

To create a Web Forms page

1. In Solution Explorer, right-click the project name, click **Add**, then click **Add Web Form**.

   The **Add New Item** dialog box is displayed.

2. In the **Categories** pane, verify the **Web Project Items** node is highlighted and select **Web Form** in the **Templates** pane.
3. Supply a name for the form in the **Name** box at the bottom, then click **Open**.

   The new Web Forms page is created and displayed in the Web Forms Designer. If you want to give your form a different name, rename it in Solution Explorer. For an overview of managing Web Form pages, see Creating and Managing Web Forms Pages.

To add an existing Web Forms page to a project

1. In Solution Explorer, right-click the project name and click **Add**, then click **Add Existing Item**.

   The **Add Existing Item** dialog box is displayed.

2. Navigate to the directory containing the Web Forms page, select the page, and click **Open**.
3. If the existing file does not have a code-behind class file (.aspx.vb or .aspx.cs file), Visual Studio prompts you to create one. Typically, if the file page you are adding is a single-file Web Forms page, you do not need to create a code-behind class file, and you can click **No**.

   The Web Forms page is added to the project and opened in Design view.

   > **Note** When you add an existing file, the file is copied to your project. (The file is not simply added by reference). If you change the file in your project, the original file is left unchanged.

You can also rename a page by renaming it in Solution Explorer.

To change the name of a Web Forms page in a project using Solution Explorer

1. In Solution Explorer, right-click the file you want to change and click **Rename**.
2. Type the new file name you want, then press ENTER.

See Also

Web Forms Pages and Projects in Visual Studio | Creating and Managing Web Forms Pages

Compilation and Deployment of Web Projects

When using Visual Studio .NET to create ASP.NET Web applications, you use a standard deployment model: your project is compiled and the resulting files are deployed. The Web Forms code-behind class file (.aspx.vb, or .aspx.cs), but not the .aspx file, is compiled into a project .dll file along with all other class files included in your project. This single project .dll file is then deployed to the server, without any source code. When a request for the page is received, the project .dll file is instantiated and executed.

> **Note** For more information about how Web Forms pages are compiled, see Web Forms Code Model.

An implication of this model is that you can change the .aspx file after the project is built, but you cannot change the code in the class file without recompiling. In other words, you can change visual elements of your Web Forms pages without the need for recompiling and redeploying.

> **Security Note** At design time in Visual Studio, your code in your project always runs fully trusted. This has two implications. The first is that you can potentially run unsafe code by importing it into your project and running it as fully trusted. The second is that code in your project might work properly when you are testing it on your own computer, but might fail due to lack of adequate permissions in a deployed application. For more information, see Code Access Security.

Compiling and Deploying an ASP.NET Web Application Project

Suppose you have a simple ASP.NET Web application project called WebApplication1 that contains the following:

- A Web Forms page called WebForm1.aspx.
- Some additional modules containing business logic, written in the language of your choice, called BusinessLogic1 (.vb or .cs) and BusinessLogic2 (.vb or .cs).

You have edited the Web Forms page and are ready to compile the project. The following occurs when you compile it:

- All files in the project that have been modified since the last build are saved.
- The WebForm1.aspx file and default project files are copied to the Web server.
- The WebForm1 class file (.aspx.vb or .aspx.cs), BusinessLogic1 component, and BusinessLogic2 component are compiled into the project .dll file, which is copied to the server in the default \Bin directory of the target virtual directory.

> **Note** If there are compilation errors, the deployment will not succeed. For more information on compiling an ASP.NET Web application project, see Compiling and Running Web Forms Pages. For more information on deployment, see Deploying Applications and Components.

See Also

Compiling and Running Web Forms Pages

Compiling and Running Web Forms Pages

When you are ready to build and run your Web Forms page, you must first compile the ASP.NET Web application project. Once the Web application is compiled, the pages included within it can be run. There are three methods you can utilize to build the application and run your Web Forms page.

Method	Description
With the debugger	Starts the default browser and loads the specified start page. Running a page in the debugger allows you to step through code line by line and make use of additional analysis tools and run-time information. If Visual Studio detects that key files have changed, it will also build the project before launching the browser with the specified start page. **Note** In order to run a project with the debugger, you must be a member of the Debugger Users group on the server computer, and the computer must be configured to support debugging. For details, see Introduction to Web Application Debugging.
Without the debugger	Allows you to run your code as it normally would run outside of the context of development tools, and therefore no run-time information is available through these tools. If Visual Studio detects that key files have changed, it will build the project before launching the browser with the specified start page.
View in Browser	Compiles the project and opens a Web page chosen from Solution Explorer. The project is compiled and run in the default browser within Visual Studio.

Note When your Web application runs, it executes within the context of a special local user, called ASPNET, that determines the permissions that your application will have to access resources. For details, see Access Permissions for Web Applications.

To build and run a Web Forms page with the debugger

1. In Solution Explorer, right-click your Web Forms page and click **Set As Start Page**.
2. On the **Debug** menu, click **Start**. This command instructs Visual Studio to run the ASP.NET Web application in the debugger.
3. To stop running the form and return to design mode, close the browser, or on the **Debug** menu click **Stop Debugging**.

Tip F5 is the keyboard shortcut for **Start**, and SHIFT+F5 is the keyboard shortcut for **Stop Debugging**.

To build and run a Web Forms page without the debugger

1. In Solution Explorer, right-click your Web Forms page and click **Set As Start Page**.
2. On the **Debug** menu, click **Start without Debugging**. Visual Studio saves all files in the project and then builds it. The project's start page is launched in the default browser.

   Tip CTRL+F5 is the keyboard shortcut for **Start without Debugging**.

3. To stop running the form and return to design mode, close the browser, or click **Stop Debugging** on the **Debug** menu.

To build and run a Web Forms page using the View in Browser feature

1. In Solution Explorer, right-click the Web Forms page you want to preview, and click **View in Browser**.
2. Visual Studio builds the ASP.NET Web application and launches the specified start page within the default browser. The page is labeled **Preview** in the Visual Studio environment.
3. To stop running the preview, close the preview page.

See Also

Compilation and Deployment of Web Projects | Introduction to Web Application Debugging

Programming Web Forms

An important part of developing Web applications is programming the user interface. The following topics provide information about programming Web Forms pages, with specific emphasis on the unique aspects of working in a Web-based environment.

In This Section

Server Event Handling in Web Forms Pages
Gives an overview of event-handling general types, and links to specific event-handling procedures.
Page and Application Context
Describes the intrinsic objects in ASP.NET that provide access to the current application, each user session, the current HTTP request, the requested Web Forms page, and so on.
Redirecting Users to Another Page
Gives instruction on redirecting users from one Web Forms page to another.
Detecting Browser Types in Web Forms
Gives directions for detecting browser types in Web Forms pages.
Web Forms State Management
Gives an overview of ASP.NET state management options.

Related Sections

Web Forms Pages
Suggests additional topics of interest concerning Web Forms.
Web Forms Page Processing
Explains how Web Forms pages are processed, including what stages (events) they go through and what you can do at each stage.
ASP.NET Server Controls
Describes the controls available for use on Web Forms pages.

Server Event Handling in Web Forms Pages

As with traditional forms, when you work with server controls on a Web Forms page, much of the page's logic involves responding to events raised by controls. However, events work somewhat differently in Web Forms pages than in client-based forms such as Windows Forms.

The following topics explain the event-handling model for Web Forms pages and provide instructions on how to create event handlers.

In This Section

ASP.NET Server Control Event Model
Explains how events that originate on the client (browser) are handled in server-side code.
Creating Event Handlers in Web Forms Pages
Gives details on how to create event handlers using the Web Forms Designer.
Binding to an Existing Event Handler in Web Forms Pages
Gives details on how to bind an existing event handler to an event in Web Forms pages.
Creating an Event Handler at Run Time in Web Forms Pages
Gives details on how to create event handlers for an event at run time.
Connecting Multiple Events to a Single Event Handler in Web Forms Pages
Gives details on how to connect multiple events to a single event handler in Web Forms pages at run time.

Related Sections

Handling and Raising Events
Links to detailed discussions of how events and event handlers function in ASP.NET.
ASP.NET Server Controls
Describes the controls available for use on Web Forms pages.

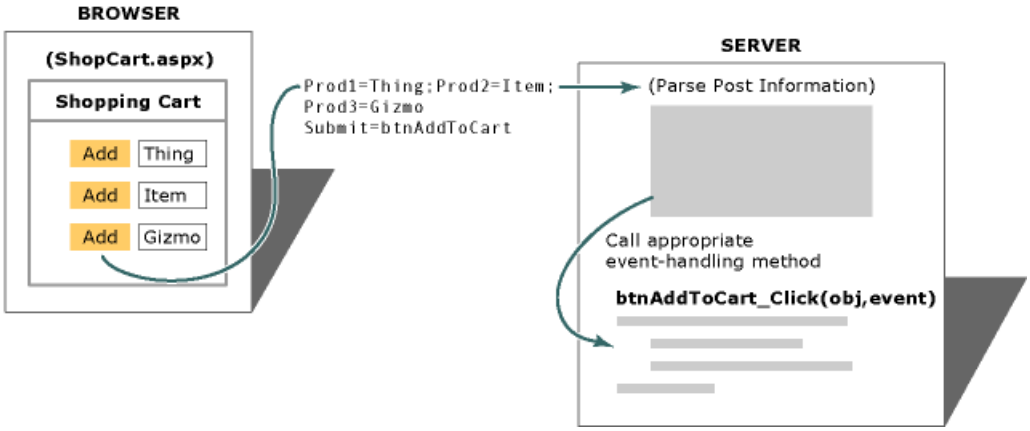
ASP.NET Server Control Event Model

Events raised by ASP.NET server controls work somewhat differently than events in traditional client forms or in client-based Web applications. The difference arises primarily because of the separation of the event itself from where the event is handled.

In client-based applications, events are raised and handled on the client. In Web Forms pages, on the other hand, events associated with server controls are raised on the client but handled on the Web server by the ASP.NET page framework.

For events raised on the client, the Web Forms control event model requires that the event information be captured on the client and an event message transmitted to the server, via an HTTP post. The page framework must interpret the post to determine what event occurred and then call the appropriate method in your code on the server to handle the event. For more information, see Events in ASP.NET Server Controls.

The Web Forms control event model



ASP.NET handles virtually all the mechanics of capturing, transmitting, and interpreting the event. When you create event handlers in a Web Forms page, you can do so without thinking about the mechanics of how the event information is captured and made available to your code. Instead, you can create event handlers in much the same way you would do so in a traditional client form. Even so, there are some aspects of event handling in Web Forms pages that you should be aware of.

Intrinsic Event Set

Because most Web Forms events require a round trip to the server for processing, they can affect the performance of a form. Therefore, server controls offer a limited set of intrinsic events, usually only click-type events. Some server controls support a special version of the onchange event, which is raised when the control's value changes. For example, the **CheckBox** Web server control raises a change event when the user clicks the box. Events that occur often (and can be raised without the user knowing it), such as an onmouseover event, are not supported for server controls.

> **Note** Some server controls support a set of higher-level events. For example, the **Calendar** Web server control raises a **SelectionChanged** event that is a more abstract version of a **Click** event.

Event Arguments

Web and HTML server control events follow a standard .NET Framework pattern for event handler methods. All events pass two arguments: an object representing the object that raised the event, and an event object containing any event-specific information. The second argument is usually of type **System.EventArgs**, but for some controls is of a type specific to that control. For example, for an **ImageButton** Web server control, the second argument is of the type **ImageClickEventArgs**, which includes information about the coordinates where the user has clicked.

Postback and Non-Postback Events in Web Server Controls

In Web server controls, certain events, typically click events, cause the form to be posted back to the server. Change events in HTML server controls and Web server controls, such as the **TextBox** control, are captured but do not immediately cause a post. Instead, they are cached by the control until the next time a post occurs. Then, when the page is processed on the server again, all the pending events are raised and processed.

> **Note** If the browser supports it, validation controls can check user input using client script, without a round trip to the server. For details, see Introduction to Validating User Input in Web Forms.

During server page processing, events are processed first, in no particular order. When all change events have been processed, the click event that caused the form to be posted is processed.

> **Note** You should not create application logic that relies on the change events being raised in a specific order unless you have detailed knowledge of page event processing. For more information, see Postback Event Sample.

You can specify that a change event cause a form post. Web server controls that support a change event include an **AutoPostBack** property. When this property is **true**, the control's change event causes the form to post immediately, without waiting for a click event. For example, by default, a **CheckBox** control's **CheckedChanged** event does not cause the page to be submitted. However, by setting the control's **AutoPostBack** property to **true**, you specify that as soon as a user clicks the check box, the page is sent to the server for processing.

> **Note** For the **AutoPostBack** property to work properly, the user's browser must be set to allow scripting. This is the default in most cases. However, some users disable scripting for security reasons. For details, see ASP.NET Server Controls and Browser Capabilities.

Bubbled Events

Web server controls such as the **Repeater**, **DataList**, and **DataGrid** controls can contain button controls that themselves raise events. For example, each row in a **DataGrid** control can contain one or more buttons created dynamically by templates.

Rather than each button raising an event individually, events from the nested controls are *bubbled* — that is, they're sent to the container. The container in turn raises a generic event called **ItemCommand** with parameters that allow you to discover which individual control raised the original event. By responding to this single event, you can avoid having to write individual event handlers for child controls.

The **ItemCommand** event includes the two standard event arguments, an object referencing the source of the event and an event object containing event-specific information.

> **Note** The **DataGrid** and **DataList** Web server controls support additional events, such as **EditCommand**, **DeleteCommand**, and **UpdateCommand**, which are special cases of bubbled events.

With buttons, you can use the **CommandArgument** property to pass a user-specified string to the event handler to help you identify what button raised the event. For example, in a **DataList** control, buttons raise the **ItemCommand** event. You can set the **CommandArgument** property of each button to a different value — perhaps one button's value is "ShowDetails" and another button's value is "AddToShoppingCart" and then capture those values in the event handler later.

Event Delegates in Web Forms Pages

An event is a message — effectively, something like "a button has been clicked". In your application, you want the message to be translated into a method call in your code, such as "Button1_Click". The binding between the event message and a specific method — that is, an event handler — is done using an event delegate. For more information, see Events and Delegates.

In Web Forms pages, you typically do not need to explicitly code delegates. If you use the Web Forms Designer in Visual Studio, the designer generates code that automatically binds events to methods. In Visual Basic, this is done using the Handles keyword in the event handler declaration, as in this example:

```
Private Sub Page_Load(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles MyB
ase.Load
```

In Visual C#, the designer generates an explicit event-handler delegate in the page, similar to the following:

```
private void InitializeComponent()
{
   this.Load += new System.EventHandler(this.Page_Load);
}
```

Alternatively, the ASP.NET page framework also supports an automatic way to associate page events and methods. If the **AutoEventWireup** attribute of the Page directive is set to **true** (or if it is missing, since by default it is **true**), the page framework calls page events automatically, specifically the **Page_Init** and **Page_Load** methods. In that case, no explicit **Handles** clause or delegate is needed.

The disadvantage of the **AutoEventWireup** attribute is that it requires that the page event handlers have specific, predictable names. This limits your flexibility in how you name your event handlers. Therefore, in Visual Studio, the **AutoEventWireup** attribute is set to **false** by default and the designer generates explicit code to bind page events to methods.

If you do set **AutoEventWireup** to **true**, Visual Studio will generate code to bind the events and the page framework will automatically call events based on their names. This can result in the same event code being called twice when the page runs. As a

consequence, you should always leave **AutoEventWireup** set to **false** when working in Visual Studio.

Responding to Both Client and Server Events in ASP.NET Server Controls

For the most part, you will be primarily interested in events that are raised in server code. However, if it is appropriate for your application, you can also handle client-side events for ASP.NET server controls by writing client script.

> **Note** You cannot use HTML syntax to bind to client-side events for Web server controls. Instead, you have to add event-binding attributes in code. For an example, see the table below.

For example, you might have an HTML image button element that you have converted to an HTML server control. Typically, in a Web Forms page you would handle the image button's click event in server code. However, you might also want to use client code to change the image when the user moves the mouse over it. You can do that by creating a client script for the image button's **onmouseover** event. (In this example, the assumption is that you are working with a browser that supports HTML 4.0, such as Microsoft Internet Explorer 4.0 or later.)

> **Note** In the case that both the client-side event handler and a server-side event handler have the same event-handler name, the client-side event handler always runs first, then the server event handler. However, it is confusing to allow this scenario, so some planning for a naming convention is strongly suggested.

Handling a Click Event in Client and Server Code

There is one event — the client onclick event — that introduces a difficulty if you want to handle it on both the client and the server. The problem arises because all button server controls (and other controls whose **AutoPostBack** property is set to **true**) submit the page to the server. However, in HTML, only a few controls inherently submit a form:

- The HTML submit button (`<INPUT type=submit>`), which is the HtmlInputButton control with its type set to **Submit** or **Image**. You can add this control from the HTML tab of the Toolbox.
- The **Button** Web server control (`<asp:button>`).

For all other controls that are specified to submit the page, a small client script is written into the page and called to submit the form when the control is clicked. Those controls therefore already use the client-side **OnClick** event to invoke this submit script.

You can create client-side **OnClick** handlers for all controls, but you need to choose the approach that works with each type of control. The following table summarizes the strategies you use for different types of controls.

Control	Strategy
HtmlInputButton, which includes HTML server control buttons whose type is set to Submit, Reset, or Image	Include an **onclick** attribute in the HTML syntax for the control: `<INPUT Type="Submit" Runat="Server" Value="caption" onclick="clientfunction()" ...>` On the server side, these types of buttons raise a ServerClick event instead of a simple **Click** event. The client event is raised first, and then the form is submitted and the server event is handled.
All other HTML controls (those that do not submit a form by default)	Include an **onclick** attribute in the HTML syntax for the control, but follow it with a semicolon (;): `<INPUT Type="Button" Runat="Server" Value="caption" onclick="clientfunction();" ...>` This causes your function to be called first, before the client-side submit script is called.
Web server controls, including buttons (`<asp:button>`) and other controls (for example, `<asp:checkbox>`)	You cannot specify a client-side event in the HTML syntax for a Web server control. Instead, add an event attribute to the control at run time in server code using code such as the following: `Button1.Attributes.Add("onclick", "clientfunction();")` **Note** For the **Button** Web server control, you do not need the semicolon, because that control automatically submits the page.

Application and Session Events

In addition to page and control events, the ASP.NET page framework provides ways for you to work with events that can be raised when your application starts or stops or when an individual user's session starts or stops:

- Application events are raised for all requests to an application. For example, **Application_BeginRequest** is raised when any Web Forms page or XML Web service in your application is requested. This event allows you to initialize resources that will be used for each request to the application. A corresponding event, **Application_EndRequest**, provides you with an opportunity to close or otherwise dispose of resources used for the request.
- Session events are similar to application events (there is a **Session_Start** and a **Session_End** event), but are raised with each unique session within the application. A session begins when a user requests a page for the first time from your application and ends either when your application explicitly closes the session or when the session times out.

You can create handlers for these types of events in the Global.asax file. For details, see Global.asax File and Global.asax Syntax.

See Also

Server Event Handling in Web Forms Pages | Global.asax File | Introduction to Web Forms State Management | Working with Httpapplication Instances

Creating Event Handlers in Web Forms Pages

Controls have a default event, which is the event most commonly associated with that control. For example, the default event for a button is the click event. You can create event handlers for both the default event and other events; however, the procedure is different for each type of event.

To create the event handler for a default event

- In Design view of the Web Forms Designer, double-click the control or page.

  The Code Editor opens with the insertion point in the event handler.

There is a different process for creating a handler for non-default events. The process differs for Visual Basic and Visual C#, due to slight differences in the Web Forms Designer user interface for those languages.

To create an event handler for non-default events in Visual Basic

1. Open or switch to the Code Editor for the page.
2. At the top of the window, select the control from the left-hand drop-down list, and select the event from the right-hand drop-down list.

   Note Events for which methods already exist are displayed in bold.

   The event handler is created.

   Visual Basic creates an event handler and adds it to your form's class.

To create an event handler for non-default events in Visual C#

1. In Design view, select the control, and then press F4 to display the Properties window.
2. In the Properties window, click the **Events** button ( ).

   The Properties window displays a list of the events for the control, with boxes to the right that display the names of event handlers bound to those events.

3. Locate the event you want to create a handler for and then:

   - In the event name box, type the name of an event handler.

   -or-

   - Double-click the event name box to create a handler whose name follows the convention *controlID_eventname*.

   The new event handler is created with the name you typed or with the generated name.

See Also

Binding to an Existing Event Handler in Web Forms Pages | Creating an Event Handler at Run Time in Web Forms Pages | Connecting Multiple Events to a Single Event Handler in Web Forms Pages | ASP.NET Server Control Event Model | Handling and Raising Events | Server Event Handling in Web Forms Pages | ASP.NET Server Controls

Binding to an Existing Event Handler in Web Forms Pages

If your page already contains an event-handling method, you can bind an event to it rather than creating a new handler. The process is slightly different for Visual Basic and Visual C#.

To bind to an existing event handler in Web Forms using Visual Basic

- Modify the event-handler declaration to include a Handles keyword specifying the event to bind to the method, as in the following example:

```
Sub MyEventHandler (ByVal sender as System.Object, ByVal e as System.EventArgs) Handles B
utton1.Click
```

> **Note** It is up to you to make sure that the method signature (the number and type of the method arguments) is appropriate for the event you want to bind to.

To bind to an existing event handler in Web Forms using Visual C#

1. In Design view, select the control whose event you want to bind to a method.
2. In the Properties window, click the **Events** button ( ).

   The Properties window displays a list of the events for the control, with boxes to the right that display the names of event handlers bound to those events.
3. From the drop-down list box, select the method to bind the event to. The methods displayed in the list are those that have the correct signature for the event you want to handle.

See Also

Creating Event Handlers in Web Forms Pages | Creating an Event Handler at Run Time in Web Forms Pages | Connecting Multiple Events to a Single Event Handler in Web Forms Pages | ASP.NET Server Control Event Model | Handles | Server Event Handling in Web Forms Pages | ASP.NET Server Controls | Event Handlers in Visual Basic and Visual C#

Creating an Event Handler at Run Time in Web Forms Pages

If you already have an event handler with the appropriate signature, you can bind a control event to it at run time.

To create an event handler at run-time in Web Forms using Visual Basic

- Include an AddHandler statement, passing it the event to bind and the address of the method to call. Be sure that the statement is executed before the event can be raised — typically, you add handlers in the page initialization. The following example shows how you can bind the **Click** event of the **Button1** control a method called `myEventHandler`:

```
AddHandler Button1.Click, AddressOf myEventHandler
```

To create an event handler at run-time in Web Forms using Visual C#

- Create an instance of the EventHandler delegate, passing it the address of the method to bind to. Then add the delegate object to the list of methods called when the event is raised. The following example shows how you can bind the **Click** event of the **Button1** control to a method called `myEventHandler`:

```
Button1.Click += new System.EventHandler(this.myEventHandler);
```

See Also

Creating Event Handlers in Web Forms Pages | Binding to an Existing Event Handler in Web Forms Pages | Connecting Multiple Events to a Single Event Handler in Web Forms Pages | ASP.NET Server Control Event Model | Handles | Server Event Handling in Web Forms Pages | ASP.NET Server Controls | Handling and Raising Events (.NET Framework documentation)

Connecting Multiple Events to a Single Event Handler in Web Forms Pages

If you already have an event handler, you can bind several control events to it. These multiple events can be from the same control or one event from several different controls (as long as the events all have the same signature). For example, you may want to bind the **Click** events of several **Button** Web server controls on a Web Forms page to a single event handler.

For both Visual Basic and Visual C#, you must add specific code to the event-handler method signature in the code-behind file.

To connect multiple events to a single event handler in Visual Basic

- Modify the Handles clause of a method by adding the names of the events that the method should handle. Separate the event names with commas.

  The following example shows how you can have the method `MyMultipleEventHandler` bound to events raised by three controls:

  ```
  Sub MyMultipleEventHandler (ByVal sender as System.Object, ByVal e as System.EventArgs) H
  andles Button1.Click, Button2.Click, Button3.Click
  ```

  Note It is up to you to make sure that the method has the correct signature for the event it is handling.

If you are using Visual C#, you associate multiple control events with a single event handler using the Web Forms Designer.

To connect multiple events to a single event handler in Visual C#

1. In Design view, select a control whose event you want to bind to a method.
2. In the Properties window, click the **Events** button ( ).

   The Properties window displays a list of the events for the control, with boxes to the right that display the names of event handlers bound to those events.
3. From the drop-down list, select the method to bind the event to. The methods displayed in the list are those that have the correct signature for the event you want to handle.

   The Code Editor opens, showing the method you selected. In addition, a line is added to the page's **InitializeComponent** method to perform the binding. The following shows an example of two button Click events that call the same handler:

   ```
   this.Button1.Click += new System.EventHandler(this.myEventHandler);
   this.Button2.Click += new System.EventHandler(this.myEventHandler);
   ```
4. Repeat Steps 1 through 3 for the next control whose event you want to bind to the method.

When your handler is called, you can determine which control caused the event.

To determine which control caused the event

1. In the event handler, declare a variable whose type matches the controls that raise the event.
2. Assign the `sender` argument of the event handler to the variable, casting it to the appropriate type.

   The following example shows the handler for a **Button**-control click event that is called by several different buttons. The handler displays the ID property of the button that caused the event.

   ```
   ' Visual Basic
   Private Sub Button1_Click(ByVal sender As System.Object, _
       ByVal e As System.EventArgs) _
       Handles Button1.Click, Button2.Click, Button3.Click
           Dim b As Button = CType(sender, Button)
           Label1.Text = b.ID
   End Sub
   ```

```
// C#
private void Button1_Click(object sender, System.EventArgs e)
    {
        Button b = (Button) sender;
        Label1.Text = b.ID;
    }
```

See Also

Creating Event Handlers in Web Forms Pages | Binding to an Existing Event Handler in Web Forms Pages | Creating an Event Handler at Run Time in Web Forms Pages | ASP.NET Server Control Event Model | Server Event Handling in Web Forms Pages | Handles | ASP.NET Server Controls | Handling and Raising Events (.NET Framework documentation)

Page and Application Context in Web Forms Applications

When a Web application runs, ASP.NET maintains information about the current application, about each user session, the current HTTP request, the requested Web Forms page, and so on. The ASP.NET page framework contains a series of classes to encapsulate this context information.

ASP.NET makes instances of these classes available as intrinsic objects that you can access from your code. The following table lists these intrinsic objects and the classes they are instances of.

> **Note** Although the classes that define these objects are new with ASP.NET, the objects are used in the same way as with previous versions of ASP.

Object Name	Description	ASP.NET Class
Response	Provides access to the output stream for the current page. You can use this class to inject text into the page, write cookies, and more. For details, see Page.Response Property.	HttpResponse
Request	Provides access to the current page request, including the request headers, cookies, client certificate, query string, and so on. You can use this class to read what the browser has sent. For details, see Page.Request Property.	HttpRequest
Context	Provides access to the entire current context (including the request object). You can use this class to share information between pages.	HttpContext
Server	Exposes utility methods that you can use to transfer control between pages, get information about the most recent error, encode and decode HTML text, and more.	HttpServerUtility
Application	Provides access to application-wide methods and events for all sessions. Also provides access to an application-wide cache you can use to store information. For details, see Application State.	HttpApplicationState
Session	Provides information to the current user session. Also provides access to a session-wide cache you can use to store information, along with the means to control how the session is managed. For details, see Session State.	HttpSessionState
Trace	Provides a way to get both system and custom trace diagnostic messages to display in the HTTP page output. For details, see ASP.NET Trace.	TraceContext

The following topics show examples of how the intrinsic objects can be used.

Object	Example topics
Application	• Code: Reading Values From Application State (Visual Basic) • Code: Saving Values in Application State (Visual Basic)
Request	• Code: Reading a Cookie (Visual Basic) • Code: Retrieving the Client's Browser Information (Visual Basic) • Code: Writing a Cookie (Visual Basic)
Server	• Code: Encoding HTML Text (Visual Basic) • Code: Handling Application-Level Errors (Visual Basic) • Code: Handling Page-Level Errors (Visual Basic) • Code: Redirecting to Another Page in the Same Application (Visual Basic)
Session	• Code: Caching a Dataset in Session State (Visual Basic) • Code: Reading Values From Session State (Visual Basic) • Code: Saving Values in Session State (Visual Basic)

See Also

Introduction to Web Forms State Management | ASP.NET State Management

Redirecting Users to Another Page

You may want to redirect users from one Web Forms page to another page. You might do this to display a page that is matched to the user's browser capabilities or is written in the language that the user speaks.

There are two ways to redirect pages:

- Using a server-side method. In this scenario, the server simply transfers the context to another page. The advantage is that you can share page context information between pages. The disadvantage is that the user's browser does not know about the transfer, so the browser's history is not updated. If the user refreshes the page, unexpected results can occur.
- Using the browser. In this scenario, you send a command to the user's browser that causes the browser to fetch a different page. The advantage is that the browser's history is updated. The disadvantage is that this scenario performs an extra round trip, which can affect performance.

To redirect users to another page using a server-side method

- Call the Server.Transfer, passing it the name of the page to redirect to.
- The following example shows how to redirect to another page.

```
' Visual Basic
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Han
dles Button1.Click
        Server.Transfer("Webform2.aspx")
    End Sub

// C#
private void Button1_Click(object sender, System.EventArgs e)
{
   Server.Transfer("Webform2.aspx");
}
```

For details on sharing page context during a server transfer, see Passing Values Between Web Forms Pages.

To redirect users to another page from the browser

1. Set the **Response** object's BufferOutput property to **true**.
2. Call the **Response** object's Redirect method, passing it the URL of the page to redirect to.

   The following shows how to redirect based on the contents of a local variable, `UserLanguage`, that is set elsewhere.

```
' Visual Basic
Response.BufferOutput = True
If UserLanguage = "English" Then
   Response.Redirect("http://www.microsoft.com/gohere/look.htm")
ElseIf UserLanguage = "Deutsch" Then
   Response.Redirect("http://www.microsoft.com/gohere/look_deu.htm")
ElseIf UserLanguage = "Español" Then
   Response.Redirect("http://www.microsoft.com/gohere/look_esp.htm")
End If

// C#
Response.BufferOutput = true;
if (UserLanguage == "English")
{
   Response.Redirect("http://www.microsoft.com/gohere/look.htm");
}
else if (UserLanguage == "Deutsch")
{
   Response.Redirect("http://www.microsoft.com/gohere/look_deu.htm");
}
```

```
else if (UserLanguage == "Español")
{
   Response.Redirect("http://www.microsoft.com/gohere/look_esp.htm");
}
```

See Also

Passing Values Between Web Forms Pages | HttpResponse Class | HttpResponse.Redirect Method | Programming Web Forms

Detecting Browser Types in Web Forms

Different browsers, and different versions of the same browsers, support different features. In your application, you might need to determine what type of browser a user is viewing pages with, and perhaps even determine if the browser supports certain features.

> **Note** The ASP.NET page framework can automatically determine browser capabilities and use this information to render appropriate HTML elements and attributes for Web controls. For details, see ASP.NET Server Controls and Browser Capabilities.

Browsers and client devices are split into two distinctive groups: uplevel and downlevel. These groups define the type of native support a browser or client device offers, and they generally determine the presentation and behavior of a loading page from a Web server.

Browsers and client devices that are considered uplevel usually support at least the following:

- ECMAScript (JScript, JavaScript) version 1.2.
- HTML version 4.0
- The Microsoft Document Object Model (MSDOM)
- Cascading style sheets (CSS)

Downlevel browsers and client devices support the following only:

- HTML version 3.2

To detect browser types in Web Forms pages

- Query the HttpRequest.Browser property, which contains an HttpBrowserCapabilities object. This object gets information from the browser or client device during an HTTP request, telling your application the type and level of support the browser or client device offers.

  The following example shows how you can displays browser information the information in a text box on the page.

```
' Visual Basic
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Han
dles Button1.Click
   Dim s As String = ""
   With Request.Browser
      s &= "Browser Capabilities" & vbCrLf
      s &= "Type = " & .Type & vbCrLf
      s &= "Name = " & .Browser & vbCrLf
      s &= "Version = " & .Version & vbCrLf
      s &= "Major Version = " & .MajorVersion & vbCrLf
      s &= "Minor Version = " & .MinorVersion & vbCrLf
      s &= "Platform = " & .Platform & vbCrLf
      s &= "Is Beta = " & .Beta & vbCrLf
      s &= "Is Crawler = " & .Crawler & vbCrLf
      s &= "Is AOL = " & .AOL & vbCrLf
      s &= "Is Win16 = " & .Win16 & vbCrLf
      s &= "Is Win32 = " & .Win32 & vbCrLf
      s &= "Supports Frames = " & .Frames & vbCrLf
      s &= "Supports Tables = " & .Tables & vbCrLf
      s &= "Supports Cookies = " & .Cookies & vbCrLf
      s &= "Supports VBScript = " & .VBScript & vbCrLf
      s &= "Supports JavaScript = " & .JavaScript & vbCrLf
      s &= "Supports Java Applets = " & .JavaApplets & vbCrLf
      s &= "Supports ActiveX Controls = " & .ActiveXControls & vbCrLf
   End With
   TextBox1.Text = s
End Sub
```

```
// C#
private void Button1_Click(object sender, System.EventArgs e)
{
   System.Web.HttpBrowserCapabilities browser = Request.Browser;
   string s = "Browser Capabilities\n"
      + "Type = "                     + browser.Type + "\n"
      + "Name = "                     + browser.Browser + "\n"
      + "Version = "                  + browser.Version + "\n"
      + "Major Version = "            + browser.MajorVersion + "\n"
      + "Minor Version = "            + browser.MinorVersion + "\n"
      + "Platform = "                 + browser.Platform + "\n"
      + "Is Beta = "                  + browser.Beta + "\n"
      + "Is Crawler = "               + browser.Crawler + "\n"
      + "Is AOL = "                   + browser.AOL + "\n"
      + "Is Win16 = "                 + browser.Win16 + "\n"
      + "Is Win32 = "                 + browser.Win32 + "\n"
      + "Supports Frames = "          + browser.Frames + "\n"
      + "Supports Tables = "          + browser.Tables + "\n"
      + "Supports Cookies = "         + browser.Cookies + "\n"
      + "Supports VBScript = "        + browser.VBScript + "\n"
      + "Supports JavaScript = "      + browser.JavaScript + "\n"
      + "Supports Java Applets = "    + browser.JavaApplets + "\n"
      + "Supports ActiveX Controls = " + browser.ActiveXControls + "\n";
   TextBox1.Text = s;
}
```

See Also

HttpRequest Class | HttpRequest.Browser Property | HttpBrowserCapabilities.Type Property | navigator Object | Programming Web Forms

Web Forms State Management

Web Forms pages and the ASP.NET page framework use HTTP to communicate between browser or client device and the Web server. HTTP is a stateless protocol, meaning that it does not automatically indicate whether the requests in a sequence are all from the same client or even whether a single browser instance is still actively viewing a page or site. In addition, each time a page is processed in a round trip, it is destroyed and rendered anew, and all information about the state of the Web Forms page and the controls on that page are lost.

As a result, building Web applications that need to maintain cross-request state information — for example, showing a shopping cart or paging through data — can be extremely challenging without additional infrastructure help.

The following topics provide a general background and how-to information on how you can manage state in your Web Forms pages.

In This Section

Introduction to Web Forms State Management
Provides an overview of how values are preserved in a Web Forms page during round trips to the server.
State Management Recommendations
Provides an overview of the advantages and disadvantages of state management techniques available in ASP.NET.
Saving Web Forms Page Values Using View State
Provides information on how to store information in view state (in the page) between round trips.
Passing Values Between Web Forms Pages
Describes how to redirect one Web Forms page to another and pass information to the target page.

Related Sections

ASP.NET Applications
Provides information on additional topics of interest concerning Web Forms and pages ASP.NET applications.
ASP.NET State Management
Provides information on ASP.NET state management.
Programming Web Forms
Provides information on programming in Web Forms pages.
ASP.NET Server Controls
Provides information on server controls provided by ASP.NET.
Web Forms Pages
Gives an overview of Web Forms technology in ASP.NET.
ASP.NET Server Control Event Model
Provides information on the Web Forms control event model.

Introduction to Web Forms State Management

Web pages are recreated each time the page is posted to the server. In traditional Web programming, this would ordinarily mean that all information associated with the page and the controls on the page would be lost with each round trip. For example, if a user enters information into a text box, that information would be lost in the round trip from the browser or client device to the server. For more information, see Web Forms Page Processing.

To overcome this inherent limitation of traditional Web programming, the ASP.NET page framework includes various options to help you preserve changes — that is, for managing state. The page framework includes a facility called *view state* that automatically preserves property values of the page and all the controls on it between round trips. However, you will probably also have application-specific values that you want to preserve. To do so, you can use one of the state management options.

Some of the options involve keeping information on the client — for example, directly in the page or in a cookie — and others involve storing information on the server between round trips. Each option has distinct advantages and disadvantages.

Client-Based State Management Options

The following sections describe options for state management that involve storing information either in the page or on the client computer. For these options, no information is maintained on the server between round trips.

View State

The Control.ViewState property provides a dictionary object for retaining values between multiple requests for the same page. This is the method that the page uses to preserve page and control property values between round trips.

When the page is processed, the current state of the page and controls is hashed into a string and saved in the page as a hidden field. When the page is posted back to the server, the page parses the view state string at page initialization and restores property information in the page.

You can store values in view state as well. For details, see Saving Web Forms Page Values Using View State. For recommendations on when to use view state, see State Management Recommendations.

Hidden Form Fields

ASP.NET allows you to use HTML-standard hidden fields in a form. A hidden field does not render visibly in the browser, but you can set its properties just as you can with a standard control. When a page is submitted to the server, the content of a hidden field is sent in the HTTP Form collection along with the values of other controls. A hidden field acts as a repository for any page-specific information that you would like to store directly in the page.

> **Security Note** It is easy for a malicious user to see and modify the contents of a hidden field. Do not store any information in a hidden field that is sensitive or that your application relies on to work properly.

A hidden field stores a single variable in its value property and must be explicitly added to the page. Then you insert your value into the hidden field. ASP.NET provides the **HtmlInputHidden** control that offers hidden field functionality. For more information see ASP.NET Server Controls by Function.

In order for hidden field values to be available during page processing, you must submit the page using an HTTP post method. That is, you cannot take advantage of hidden fields if a page is processed in response to a link or HTTP GET method. For usage recommendations, see State Management Recommendations.

Cookies

A cookie is a small amount of data stored either in a text file on the client's file system or in-memory in the client browser session. It contains page-specific information the server sends to the client along with page output. Cookies can be temporary (with specific expiration times and dates) or persistent.

You can use cookies to store information about a particular client, session, or application. The cookies are saved on the client device, and when the browser requests a page, it sends the information in the cookie along with the request information. The server can read the cookie and extract its value. A typical use is to store a token (perhaps encrypted) indicating that the user has already been authenticated in your application.

> **Security Note** The browser can only send the data back to the server that originally created the cookie. However, malicious users have ways to "steal" cookies and read their contents. It is recommended that you do not store sensitive information in a cookie, such as a user name or password. Instead, store a token that you can use to look up the sensitive information on the server.

For more information on using cookies, see HttpResponse.Cookies Property and State Management Recommendations.

Query Strings

A query string is information appended to the end of a page's URL. A typical example might look like the following:

```
http://www.contoso.com/listwidgets.aspx?category=basic&price=100
```

In the URL path above, the query string starts with the question mark (?) and includes two attribute-value pairs, one called "category" and the other called "price."

Query strings provide a simple but limited way of maintaining some state information. For example, they are an easy way to pass information from one page to another, such as passing a product number from one page to another page where it will be processed. However, most browsers and client devices impose a 255-character limit on the length of the URL.

> **Security Note** Information passed in a query string can be tampered with by a malicious user. Do not rely on query strings to convey important or sensitive data. For more information, see Scripting Exploits.

In order for query string values to be available during page processing, you must submit the page using an HTTP get method. That is, you cannot take advantage of a query string if a page is processed in response to an HTTP post method. For usage recommendations, see State Management Recommendations.

Server-Based State Management Options

ASP.NET offers you a variety of ways to maintain state information on the server, as described in the following sections.

Application State

ASP.NET allows you to save values using application state (an instance of the HttpApplicationState class) for each active Web application. Application state is a global storage mechanism accessible from all pages in the Web application and is thus useful for storing information that needs to be maintained between server round trips and between pages. For more information, see Application State.

Application state is a key-value dictionary structure created during each request to a specific URL. You can add your application-specific information to this structure to store it between page requests.

Once you add your application-specific information to application state, the server manages it. For usage recommendations, see State Management Recommendations.

Session State

ASP.NET allows you to save values using session state, which is an instance of the HttpSessionState class for each active Web application session. (For an overview, see Session State.)

Session state is similar to application state, except that it is scoped to the current browser session. If different users are using your application, each will have a different session state. In addition, if the same user leaves your application and then returns later, that user will also have a different session state.

Session state is structured as a key-value dictionary structure for storing session-specific information that needs to be maintained between server round trips and between requests for pages. For more information, see Session State.

Session state enables you to:

- Uniquely identify browser or client-device requests and map them to an individual session instance on the server.
- Store session-specific data on the server for use across multiple browser or client-device requests within the same session.
- Raise appropriate session management events. In addition, you can write application code leveraging these events.

Once you add your application-specific information to session state , the server manages this object. Depending on what options you specify, session information can be stored in cookies, an out-of-process server, or a SQL Server. For usage recommendations, see State Management Recommendations.

Database Support

Maintaining state using database technology is a common practice when storing user-specific information where the information store is large. Database storage is particularly useful for maintaining long-term state or state that must be preserved even if the server must be restarted.

The database approach is often used in conjunction with cookies. For example, when a user first accesses your application, you might have the user log in. You can look up the user in your database and then pass a cookie to the user. The cookie might contain only the ID of the user in your database (for example, a customer number). You can then use the cookie in subsequent requests to find the user information in the database as needed.

> **Security Note** Because cookies can be tampered with, you should use caution in using them to store information that your application relies on. For example, never store sensitive information in a cookie. If you use cookies to store tokens (such as a user ID), consider encrypting the cookie value or adding logic to be sure that the cookie's value has not been tampered with.

The database support approach enables you to:

- Uniquely identify browser or client-device requests and map them to a unique ID.
- Maintain state by relating stored information to the unique ID. You can use the unique ID to query the database for information relating to that ID. You can then modify the information and save it back to the database for use across multiple requests for the same — or different — pages in your site.
- Raise appropriate events. A condition in the database may determine site action. For example, if the user of a commerce site attempts to purchase something that is not in stock, a database query may signal that the Web site should prompt the user to make another selection.

For more information on using a database to maintain state, see State Management Recommendations.

See Also

Web Forms State Management | Saving Web Forms Page Values Using View State | State Management Recommendations | Page Members | ASP.NET State Management | Application State | Session State | HttpCookie Class

State Management Recommendations

State management is the process by which you maintain state and page information over multiple requests for the same or different pages. As is true for any HTTP-based technology, Web Forms pages are stateless, which means that they do not automatically indicate whether the requests in a sequence are all from the same client or even whether a single browser instance is still actively viewing a page or site. Furthermore, pages are destroyed and recreated with each round trip to the server; therefore page information will not exist beyond the life cycle of a single page. For more information on server round trips and the Web Forms page life cycle, see Web Forms Page Processing.

ASP.NET provides multiple ways to maintain state between server round trips. Choosing among the options for state management available in ASP.NET will depend heavily upon your application, and it should be based on the following criteria:

- How much information do you need to store?
- Does the client accept persistent or in-memory cookies?
- Do you want to store the information on the client or server?
- Is the information sensitive?
- What sorts of performance criteria do you have for your application?

ASP.NET supports various client-side and server-side options for state management.

Client-side options are:

- The **ViewState** property
- Hidden fields
- Cookies
- Query strings

Server-side options are:

- Application state
- Session state
- Database

Client-Side State Management Options

Storing page information using client-side options doesn't use server resources. These options tend to have minimal security but fast server performance because the demand on server resources is modest. However, because you must send information to the client for it to be stored, there is a practical limit on how much information you can store this way.

View State

Web Forms pages provide the **ViewState** property as a built-in structure for automatically retaining values between multiple requests for the same page. View state is maintained as a hidden field in the page. For more information see Introduction to Web Forms State Management.

You can use view state to store your own page-specific values across round trips when the page posts back to itself. For example, if your application is maintaining user-specific information — that is, information used in the page but not necessarily part of any control — you can store it in view state.

The advantages of using view state are:

- No server resources required. The view state is contained in a structure within the page code.
- Simple implementation.
- Automatic retention of page and control state.
- Enhanced security features. The values in view state are hashed, compressed, and encoded for Unicode implementations, thus representing a higher state of security than hidden fields have.

The disadvantages of using the view state are:

- Performance. Because the view state is stored in the page itself, storing large values can cause the page to slow down when users display it and when they post it.

- Security. The view state is stored in a hidden field on the page. Although view state stores data in a hashed format, it can be tampered with. The information in the hidden field can be seen if the page output source is viewed directly, creating a potential security issue. For more information see Introduction to Web Application Security.

For more information on using view state, see Saving Web Forms Page Values Using View State.

Hidden Fields

You can store page-specific information in a hidden field on your page as a way of maintaining the state of your page. For more information on hidden fields see Introduction to Web Forms State Management.

If you use hidden fields, it is best to store only small amounts of frequently changed data on the client. ASP.NET provides the **HtmlInputHidden** control, which offers hidden field functionality. For more information on **HtmlInputHidden** see ASP.NET Server Controls by Function.

> **Note** If you use hidden fields you must submit your pages to the server using the HTTP POST method rather than requesting the page via the page URL (the HTTP GET method).

The advantages of using hidden fields are:

- No server resources are required. The hidden field is stored and read from the page.
- Broad support. Almost all browsers and client devices support forms with hidden fields.
- Simple implementation.

The disadvantages of using hidden fields are:

- Security. The hidden field can be tampered with. The information in the hidden field can be seen if the page output source is viewed directly, creating a potential security issue. For more information see Introduction to Web Application Security.
- Limited storage structure. The hidden field does not support rich structures. Hidden fields offer a single value field in which to place information. To store multiple values, you must implement delimited strings and the code to parse those strings.
- Performance. Because hidden fields are stored in the page itself, storing large values can cause the page to slow down when users display it and when they post it.

Cookies

Cookies are useful for storing small amounts of frequently changed information on the client. The information is sent with the request to the server.

The advantages of using cookies are:

- No server resources are required. The cookie is stored on the client and read by the server after a post.
- Simplicity. The cookie is a lightweight, text-based structure with simple key-value pairs.
- Configurable expiration. The cookie can expire when the browser session ends, or it can exist indefinitely on the client computer, subject to the expiration rules on the client.

The disadvantages of using cookies are:

- Limited size. Most browsers place a 4096-byte limit on the size of a cookie, although the support for 8192-byte cookie size is becoming common in the new browser and client-device versions available today.
- User-configured refusal. Some users disable their browser or client device's ability to receive cookies, thereby limiting this functionality.
- Security. Cookies are subject to tampering. Users can manipulate cookies on their computer, which can potentially represent a security compromise or cause the application dependent on the cookie to fail. For more information see Introduction to Web Application Security.
- Durability. The durability of the cookie on a client computer is subject to cookie expiration processes on the client and user intervention.

> **Note** Cookies are often used for personalization, where content is customized for a known user. In most of these cases, identification is the issue rather than authentication, so it is enough to merely store the user name, account name, or a unique user ID (such as a GUID) in a cookie and use it to access the user personalization infrastructure of a site.

For details about creating and reading cookies, see HttpResponse.Cookies Property and HttpRequest.Cookies Property.

Query Strings

A query string is information appended to the end of a page's URL. For more information, see Introduction to Web Forms State Management.

You can use a query string to submit data back to your page or to another page through the URL. Query strings provide a simple but limited way of maintaining some state information. For example, they are an easy way to pass information from one page to another, such as passing a product number to another page where it will be processed.

> **Note** Query strings are a viable option only when a page is requested via its URL. You cannot read a query string from a page that has been submitted to the server.

The advantages of using query strings are:

- No server resources required. The query string is contained in the HTTP request for a specific URL.
- Broad support. Almost all browsers and client devices support passing values in a query string.
- Simple implementation. ASP.NET provides full support for the query string method, including methods of reading query strings using the HttpRequest.Params property.

The disadvantages of using query strings are:

- Security. The information in the query string is directly visible to the user via the browser user interface. The query values are exposed to the Internet via the URL so in some cases security may be an issue. For more information on see Introduction to Web Application Security.
- Limited capacity. Most browsers and client devices impose a 255-character limit on URL length.

Client-Side Method State Management Summary

The following table summarizes client-side state management options and when you should consider using them.

Method	Use when
View state	You need to store small amounts of information for a page that will post back to itself. Use of the **ViewState** property provides functionality with basic security.
Hidden fields	You need to store small amounts of information for a page that will post back to itself or another page, and security is not an issue. **Note** You can use a hidden field only on pages that are submitted to the server.
Cookies	You need to store small amounts of information on the client and security is not an issue.
Query string	You are transferring small amounts of information from one page to another and security is not an issue. **Note** You can use query strings only if you are requesting the same page, or another page via a link.

Server-Side State Management Options

Server-side options for storing page information tend to have higher security than client-side options, but they can use more Web server resources, which may lead to scalability issues when the size of the information store is large. ASP.NET provides several options to implement server-side state management. For more information, see Introduction to Web Forms State Management.

Application State

ASP.NET provides application state via the HttpApplicationState class as a method of storing global application-specific information that is visible to the entire application. Application state variables are, in effect, global variables for an ASP.NET application. For more information, see Application State.

You can store your application-specific values in application state, which is then managed by the server. For more information, see Introduction to Web Forms State Management.

The ideal data to insert into application state variables is data that is shared by multiple sessions and does not change often.

> **Note** If you store a dataset in application state, you have to cast it from **Object** back to a dataset. For details, see

Web Data Access Strategy Recommendations.

The advantages of using application state are:

- Ease of implementation. Application state is easy to use, familiar to ASP developers, and consistent with other .NET Framework classes.
- Global scope. Because application state is accessible to all pages in an application, storing information in application state can mean keeping only a single copy of the information (for instance, as opposed to keeping copies of information in session state or in individual pages).

The disadvantages of using application state are:

- Global scope. The global nature of application state can also be a disadvantage. Variables stored in application state are global only to the particular process the application is running in, and each application process can have different values. Therefore, you cannot rely on application state to store unique values or update global counters in Web-garden and Web-farm configurations.
- Durability. Because global data stored in application state is volatile, it will be lost if the Web server process containing it is destroyed, most likely from a server crash, upgrade, or shutdown.
- Resource requirements. Application state requires server memory, which can affect the performance of the server as well as the scalability of the application.

Careful design and implementation of application state can increase Web application performance. For example, placing a commonly used, relatively static dataset in application state can increase site performance by reducing the overall number of data requests to a database. However, there is a performance trade-off. Application state variables containing large blocks of information reduce Web server performance as server load increases. The memory occupied by a variable stored in application state is not released until the value is either removed or replaced. Therefore, it is best to use application state variables only with small, infrequently changed datasets. For more information on optimizing ASP.NET Web applications, see ASP.NET Optimization.

Session State

ASP.NET provides a session state, available as the HttpSessionState class, as a method of storing session-specific information that is visible within the session only. For more information, see Introduction to Web Forms State Management and Session State.

You can store your session-specific values and objects in session state, which is then managed by the server and available to the browser or client device. The ideal data to store in session-state variables is short-lived, sensitive data that is specific to an individual session.

> **Note** If you store a dataset in application state, you have to cast it from **Object** back to a dataset. See Web Data Access Strategy Recommendations.

The advantages of using session state are:

- Ease of implementation. The session state facility is easy to use, familiar to ASP developers, and consistent with other .NET Framework classes.
- Session-specific events. Session management events can be raised and used by your application.
- Durability. Data placed in session-state variables can survive Internet Information Services (IIS) restarts and worker-process restarts without losing session data because the data is stored in another process space.
- Platform scalability. Session state can be used in both multi-computer and multi-process configurations, therefore optimizing scalability scenarios.
- Session state works with browsers that do not support HTTP cookies, although session state is most commonly used with cookies to provide user identification facilities to a Web application. For more information on using session state without cookies, see ASP.NET Configuration Sections.

For more information, see Session State.

The disadvantage of using session state is:

- Performance. Session state variables stay in memory until they are either removed or replaced, and therefore can degrade server performance. Session state variables containing blocks of information like large datasets can adversely affect Web server performance as server load increases.

Database Support

In some cases, you may wish to use database support to maintain state on your Web site. Typically, database support is used in conjunction with cookies or session state. For example, it is quite common for an e-commerce Web site to maintain state information using a relational database for the following reasons:

- Security
- Personalization
- Consistency
- Data mining

The following are typical features of a cookie-supported database Web site:

- **Security**. The visitor types an account name and password into a site logon page. The site infrastructure queries the database with the logon values to determine if the user has rights to utilize your site. If the database validates the user information, the Web site will distribute a valid cookie containing a unique ID for that user on that client computer. The site grants access to the user.
- **Personalization**. With security information in place, your site is able to distinguish each user on your site by reading the cookie on the client computer. Typically, sites have information in the database that describes the preferences of a user (identified by a unique ID). This relationship is known as personalization. The site can research the user's preferences using the unique ID contained in the cookie, and then place content and information in front of the user that pertains to the user's specific wishes and reacts to the user's preferences over time.
- **Consistency**. If you have created a commerce Web site, you may wish to keep transactional records of purchases made for goods and services on your site. This information can be reliably saved in your database and referenced by the user's unique ID. It can be used to determine if a purchase transaction has been completed, and also to determine the course of action should a purchase transaction fail. The information may also be used to inform the user of the status of an order placed using your site.
- **Data mining**. Information about your site usage, your visitors, or your product transactions can be reliably stored in your database. For example, your business development department may wish to use this data collected from your site to determine next year's product line or distribution policy. Your marketing department may wish to examine demographic information about users on your site. Your engineering and support department may wish to look at transactions and note areas where your purchasing process could be improved. Most enterprise-level relational databases such as Microsoft SQL Server contain an expansive toolset for most data mining projects.

By designing the Web site to repeatedly query the database by using the unique ID during each general stage in the above scenario, the site maintains state. In this way, the user perceives that the site is remembering and reacting to him or her personally.

The advantages of using a database to maintain state are:

- Security. Access to databases is typically very secure, requiring rigorous authentication and authorization.
- Capacity. You can store as much information as you like in a database.
- Persistence. Database information can be stored as long as you like, and it is not subject to the availability of the Web server.
- Robustness and data integrity. Databases include various facilities for maintaining good data, including triggers and referential integrity, transactions, and so on. By keeping information about transactions in a database (rather than in session state, for example), you can recover from errors more readily.
- Accessibility. The data stored in your database is accessible to a wide variety of information-processing tools.
- Wide support. There is a large range of database tools available, and many custom configurations are available.

The disadvantages of using a database to maintain state are:

- Complexity. Using a database to support state management implies more complex hardware and software configurations.
- Performance. Poor construction of the relational data model can lead to scaling issues. Also, leveraging too many queries to the database can adversely affect server performance.

Server-Side Method State Management Summary

The following table summarizes server-side state management options and when you should consider using them.

Method	Use when
Application state	You are storing infrequently changed, global information that is used by many users, and security is not an issue. Do not store large quantities of information in application state.
Session state	You are storing short-lived information that is specific to an individual session, and security is an issue. Do not store large quantities of information in session state. Be aware that a session state object will be created and maintained for the lifetime of every session in your application. In applications hosting many users, this can occupy significant server resources and affect scalability.
Database support	You are storing large amounts of information, managing transactions, or the information must survive application and session restarts. Data mining is a concern, and security is an issue.

See Also

Web Forms State Management | Introduction to Web Forms State Management | Programming Web Forms | State Management Recommendations | Saving Web Forms Page Values Using View State | Developing High-Performance ASP.NET Applications

Saving Web Forms Page Values Using View State

Each control on a Web Forms page, including the page itself, has a **ViewState** property that it inherits from the base **Control** class. View state is used by the ASP.NET page framework to automatically save the values of the page and of each control just prior to rendering to the page. When the page is posted, one of the first tasks performed by page processing is to restore view state.

You can use the **ViewState** property to save your values independent of control state between round trips to the server. The **ViewState** property is stored in the page in a hidden form field.

> **Note** In order to use the **ViewState** property, your form must have a server form element (<form runat="server">). For usage recommendations, see State Management Recommendations.

To store information in view state

- Create and store a new element in the ViewState property.

  > **Note** The data must be in a format compatible with view state. For example, to store a dataset in view state, you should first convert it to a string representation.

```
' Visual Basic
ViewState("color") = "yellow"

// C#
ViewState["color"] = "red";
```

To retrieve information from view state

- Get the value of an element by specifying its name. Cast the object in view state to the type that you need it to be.

  The following example shows how you can retrieve a value stored earlier under the **ViewState** name "color." Note that the value is cast to a string.

```
' Visual Basic
Dim strColor as String
strColor = CStr(ViewState("color"))

// C#
string strColor;
strColor =(string)ViewState["color"];
```

The following example shows you can save a dataset in view state and then retrieve it. To save it, you create a string representation of it using the dataset's DataSet.WriteXml method. When retrieving the dataset, you cast it to a StringReader class and then call the dataset's DataSet.ReadXml method.

```
' Visual Basic
If Page.IsPostBack Then
   Dim sr as New System.IO.StringReader(CStr(ViewState("dsCustomers")))
   DsCustomers1.ReadXml(sr)
Else
   SqlDataAdapter1.Fill(DsCustomers1)
   Dim sw as New System.IO.StringWriter()
   DsCustomers1.WriteXml(sw)
   ViewState("dsCustomers") = sw.ToString()
End If

// C#
if (Page.IsPostBack)
{
   System.IO.StringReader sr = new System.IO.StringReader((string)(ViewState["dsCustomers"]))
```

```
;
   dsCustomers1.ReadXml(sr);
}
else
{
   sqlDataAdapter1.Fill(dsCustomers1);
   System.IO.StringWriter sw = new System.IO.StringWriter();
   dsCustomers1.WriteXml(sw);
   ViewState["dsCustomers"] = sw.ToString();
}
```

See Also

Web Forms State Management | Introduction to Web Forms State Management | Maintaining State in a Control | Programming Web Forms | State Management Recommendations | Developing High-Performance ASP.NET Applications

Passing Values Between Web Forms Pages

There are many times when you will want to navigate from one page to another and pass values from the first page to the second. For example, you might have a page that prompts users for a name and password. When users submit the form, you want to call another page that authenticates the user.

There are a variety of ways to share information between pages:

- Use a query string, which appends information onto the URL and passes it to the next page. This has the disadvantage of making the information visible. For details, see WebClient.QueryString Property and HttpRequest.QueryString Property.
- Use Session state to store information that is then accessible globally to all pages in the user's current session. However, this takes server memory and the information is stored until the session expires, which can represent more overhead than you want when you are simply passing information to the next page. For details, see Web Forms State Management.
- Use custom page properties and then redirect from the original page to the target page in a way that you can still read values in the old page. This strategy is described in this topic.

Web Forms pages are classes in your application, and as such, you can create properties for them as you would any class. However, Web Forms pages have a very short lifetime, since they effectively exist only long enough to perform page processing. Properties on a Web Forms page therefore have limited use, since they exist only while the page is being processed. But if you transfer control from one page to another using specific techniques, the second page can still access properties on the first page.

To create shareable values on the source page

1. In code, declare one or more read-only properties on the page using standard syntax for properties. Return the property value you want to pass to the next page.

   The following example shows how you can declare a property called `Property1` and sets its value to the value of text box on the page:

   ```
   ' Visual Basic
   Public ReadOnly Property Property1() As String
      Get
         Return TextBox1.Text
      End Get
   End Property

   // C#
   public string Property1
   {
      get
      {
         return TextBox1.Text;
      }
   }
   ```

2. Call the next page by calling the Transfer method of the `Server` object (the HttpServerUtility class), passing it the URL of the page you want to pass information to.

   The following example shows how you might call a page called `WebForm2` (in the same project) from an event handler:

   ```
   ' Visual Basic
   Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Han
   dles Button1.Click
           Server.Transfer("Webform2.aspx")
       End Sub

   // C#
   private void Button1_Click(object sender, System.EventArgs e)
   {
      Server.Transfer("Webform2.aspx");
   ```

```
}
```

To get the property values of the first page from the called page, create an instance variable of the source page class. You then assign to it the HTTP object (an instance of the IHttpHandler class), the object that received the original request.

To read property values from the source page in the called page

1. Create a global instance variable that is typed to the class of the source page.

   The following example shows how to declare a variable called `sourcepage` that is of the type `WebForm1`:

```
' Visual Basic
' Put immediately after the Inherits statements
' at the top of the file
Public sourcepage As WebForm1

// C#
// Put immediately after the opening brace of the class
public class WebForm3 : System.Web.UI.Page
    {
        public WebForm1 sourcepage;
        // etc.
```

2. In the **Page_Load** handler, get the source page from the **Context.Handler** object (IHttpHandler interface) and assign it to the variable that you created in step 1. You must cast the handler object to the type of the source page class.

   Note You should only perform this logic the first time the page runs (that is, when the page is first called from the source page).

3. Get the property values from the source page and use them as you would any object properties.

   Note Be sure to save the property values (for example, in view state) if you want to use them in page processing other than the first page initialization. For details, see Introduction to Web Forms State Management.

   The complete **Page_Load** handler might look like this:

```
' Visual Basic
Private Sub Page_Load(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles
MyBase.Load
   If Not Page.IsPostBack Then
      sourcepage = CType(Context.Handler, WebForm1)
      Label1.Text = sourcepage.Property1
   End If
End Sub

// C#
private void Page_Load(object sender, System.EventArgs e)
{
   if (!IsPostBack){
      WebForm1 sourcepage = (WebForm1) Context.Handler;
      Label1.Text = sourcepage.Property1;
   }
}
```

See Also

Web Forms State Management

Data Access in Web Forms Pages

Many Web Forms pages involve data access — displaying data and, in some cases, allowing users to edit and update data. Knowledge of data access technology in Web Forms pages helps you create efficient Web applications.

The following sections contain introductory information on data access in Web Forms pages and on how to bind controls to data in a page.

> **Note** When you deploy an application that includes Visual Studio data-access components, you must make sure that the user installing the application has version 2.6 or higher of the Microsoft Data Access Components (MDAC). For more information, see Adding a Launch Condition for Microsoft Data Access Components.

In This Section

Introduction to Data Access in Web Forms Pages
Provides background on how data access works in Web Forms pages.
Data Sources for Web Forms Pages
Discusses what types of data you can work with in a Web Forms page.
Web Data Access Strategy Recommendations
Provides guidelines for choosing whether to use direct access to a data source or a disconnected cache (dataset); and if you use a dataset, whether to store it between round trips.
Data Walkthroughs for Web Forms Pages
Provides a list of Help topics that demonstrate various data access strategies for Web applications.
Web Forms Data Binding
Provides links to topics that discuss how data binding works in Web Forms pages and how to specify data binding for various types of controls.

Related Sections

Dataset Updates in Visual Studio .NET
Lists topics that describe how to edit, insert, and delete records in a dataset.
Database Updates from Datasets
Lists topics that describe how to save changes from a dataset to a data source.
Performing Database Operations Directly
Lists topics that describe how to read and write information directly from and to a database.
Introduction to Data Binding in Web Forms Pages
Provides a detailed introduction to data binding in Web Forms.
Data Form Wizard
Creates data-bound Windows Forms or Web Forms pages. The topic describes how to run the wizard, what options you can choose, and what the wizard generates.
Walkthrough: Creating Paged Data Access Using a Web Forms Page
Describes how to create a Web Forms page that allows users to view data records a few records at a time, paging back and forth to view earlier and later records.
Walkthrough: Creating Read-Only Data Access in a Web Forms Page
Describes how to create a Web Forms page with an ASP.NET **DataGrid** server control on it that uses a data reader to display read-only data.
Walkthrough: Displaying an XML Document in a Web Forms Page using Transformations
Gives step-by-step details on how to read an XML file and display it in different ways on a Web Forms page.
Walkthrough: Using a DataGrid Web Control to Read and Write Data
Gives step-by-step details on using a **DataGrid** control to display data, allow users to edit it, and then send the changed data back to the source.
Web Walkthroughs
Lists topics that illustrate various ways to create Web applications, including some that show examples of data binding.

Introduction to Data Access in Web Forms Pages

Web Forms pages often need to display information that derives from a database, an XML document or stream, or some other data source. The architecture of Web Forms pages provides you with ways to incorporate data sources (or references to them) in the page, to bind controls to data, and to manipulate data in various ways.

However, due to the nature of not just Web Forms pages, but of Web programming itself, data access in Web Forms pages differs in several ways from data access in Windows Forms or in forms technology of previous Microsoft products. To some extent, these are the same differences that make any kind of Web programming different from traditional forms programming: state management, separation of server and client, designing for scalability, and so on. In addition, because you are working with other resources such as databases, there are important points to understand about how to manage data in Web Forms pages.

> **Note** When you deploy an application that includes Visual Studio data-access components, you must make sure that the user installing the application has version 2.6 or higher of the Microsoft Data Access Components (MDAC). For more information, see Adding a Launch Condition for Microsoft Data Access Components.

Fundamentals of Data Access in Web Forms Pages

Data access in Web Forms pages is built around the following fundamental principles:

- Using a disconnected model
- Reading data more often than updating it
- Minimizing server resource requirements
- Accessing data using remote processes (distributing data access)

Disconnected Model

Web Forms pages are disconnected. Each time a Web Forms page is requested, it is built, processed, sent to the browser, and discarded from server memory. By extension, the same applies to data access in a Web Forms page. Data is read or updated while the page is being processed on the server; when the page is finished and sent to the browser, data is discarded along with other page elements.

This model has various implications for how you work with data in the Web Forms page:

- The data you are working with is not automatically available for each round trip — that is, when users click a button in the page. If you want to access the data again each time the page is posted to the server, you must either re-read the data from the source with each round trip or you must include code in the page to explicitly save it and restore it as part of page processing.
- It is impractical to maintain open connections to data sources. Generally, during page processing you open a connection to the source, read or write data, and then close the connection.
- During each round trip, you typically perform a single data access operation. For example, the first time the page is called, you might read data from the source and bind controls on the page to it. When users click a button, you might write data from a control back to a data source. If you are implementing operations such as paging in a grid, each round trip typically fetches one set of records.

Reading and Updating Data

The Web Forms data model presumes that most data access by Web pages is read-only. Typical examples are catalog or search listings, which display data items. (The items are often links to other pages.) In most cases, the user does not enter data that is written back to the data source.

Because most data access is read-only, the Web Forms data-binding architecture is one-way. That is, data binding displays data in controls, but does not write data from the controls to a data source.

One-way data binding makes Web Forms pages more efficient. Updating requires substantially more overhead in a page: the page must have available copies of the records to update and must include logic to update, insert, and delete records. Because this overhead can add significantly to the page processing time and to server memory requirements, and because in most pages no updates are required, the default for Web Forms pages is not to include a way to write data from a control to a data source.

If you are creating a page that updates a data source, you can include the logic yourself to perform the update operations. For example, a portal page might allow users to specify custom display options. You can use a Web Forms page to present available options and then write the user's options to a database. For details about updating, see Dataset Updates in Visual Studio .NET. For an example of how to perform updates, see Walkthrough: Updating Data Using a Database Update Query in Web Forms.

Minimizing Use of Server Resources

Because Web Forms pages are processed on the server before being sent to the browser, any data access in the page adds to the server load in both processing time and memory usage. If you choose to save data between round trips and you keep it on the server, you use server resources even when the page is not being processed.

Data access in Web Forms pages therefore requires careful attention to how you are using resources. A data–access design that uses server memory or processing time unnecessarily might work for a few users of your Web application, but can become unwieldy as your application scales up to many users. Examples of some of the design decisions you might make include:

- Get only as much data from a data source as you need in any given page.
- Use client-side state management options (for example, view state) to store data if possible.

For more details about designing data access for Web Forms pages, see Web Data Access Strategy Recommendations.

Data Access in Remote Processes

Web Forms pages are the presentation tier of your Web application. You can build data access into your pages, but it is also common to separate data access logic from the UI by building it into another component (for example, an XML Web service) that interacts with the data source. For more information, see Introduction to ASP.NET Web Applications in Visual Studio.

As .NET components, Web Forms pages exchange data with other processes via an XML stream. You generally do not need to work directly with the XML being passed between components. The data framework of your page (usually a dataset) performs this conversion automatically. However, if it makes sense in your application to do so, you can access data in XML format directly.

Access Permissions for Web Applications

Regardless of the specific data-access strategy you select for your application, you need to be aware of the security implications of data access. In most cases, data resources are protected by Windows or database-specific security. When you create data access in your application, you need to be sure that the application will have sufficient permissions at run time to get access to the data it needs. (The permissions that your application has at run time are often different from the access you have at design time while creating and testing your application.)

For details, see Access Permissions for Web Applications. For details about accessing SQL Server, see Accessing SQL Server from a Web Application.

See Also

Data Sources for Web Forms Pages | Web Data Access Strategy Recommendations | Web Forms Data Binding | Data Form Wizard | Performing Database Operations Directly | XML and SOAP Serialization

Data Sources for Web Forms Pages

The Web Forms page architecture provides a very flexible notion of data. This includes everything from traditional database access, to using XML documents as a data source, to generating data at run time and storing it in an array.

Database Access

To read and write database data, you use ADO.NET. ADO.NET includes managed data providers (connection and command objects) to communicate with SQL Server or OLE DB-compatible databases. ADO.NET also includes support for disconnected data access via a dataset, which is an in-memory cache into which you can read records to work with. For details, see Introduction to Datasets.

> **Note** You need to be sure that when your application runs, it has sufficient permissions to establish a data connection to your data source. For details, see Access Permissions for Web Applications.

Alternatively, you can use ADO.NET objects to execute SQL commands or stored procedures directly. This allows you to read data straight from the database and send updates back. For details, see Introduction to DataCommand Objects in Visual Studio.

XML Data

Another possible source of data in a Web Forms page is an XML document or stream. You can work with XML data in two ways:

- If the XML data is structured — that is, if it can be represented as relational data (even if it is not stored that way) — then you can convert the XML data into a dataset and use ADO.NET data functions to read and update the data. This allows you to take advantage of the comparatively sophisticated and simple data-processing functionality of datasets. You can then convert the data back to XML to share with other processes. For details, see Synchronizing a DataSet with an XmlDataDocument.
- If the XML data cannot be represented as relational data, you can use XML parsing and processing functions from the System.Xml namespace to manipulate the data. In Web Forms pages, you can do this using the XML Web Server Control. Alternatively, you can work directly with XML documents in code. For details, see Employing XML in the .NET Framework.

> **Note** You need to be sure that when your application runs, it has sufficient permissions to read (and if necessary, write to) XML files. For details, see Access Permissions for Web Applications.

Other Data Sources

Web Forms pages allow you to work with virtually any other type of data as well. The data-binding architecture of Web Forms pages allows you to bind a control to any structure. In practice, this means you can bind to any arrays or collections available in the page, as well as to properties of the page or of other controls.

For ultimate flexibility, you can perform data binding at run time. Each control, as well as the page itself, supports a data-binding event. By writing handlers for this event, you can bind controls on the page to any data you fetch or calculate during page processing.

For details, see Data Binding Single-Value Web Server Controls at Design Time.

See Also

Recommendations for Data Access Strategies | Web Data Access Strategy Recommendations | XML Web Server Control | Employing XML in the .NET Framework

Web Data Access Strategy Recommendations

When designing data access in your Web application, you have a variety of choices to make about how you communicate with a data source, whether you store data between round trips of the page, and if you do store data, where. The choices you make can determine how efficiently your application runs and how well it scales. There is no one data access strategy that is suitable for all situations. In fact, each choice has its own advantages and disadvantages, which you will need to understand.

The following sections detail the basic design choices you must make for Web Forms data access. The choices are presented here in a sequence; each choice is based on a choice you made previously.

> **Security Note** Accessing a database from a Web application introduces security issues that you should be aware of. For details, see Access Permissions for Web Applications. For details about accessing SQL Server, see Accessing SQL Server from a Web Application.

Dataset or Direct Access and Data Reader?

A primary choice is whether you want to cache records in a dataset or whether you want to access the database directly and read records using a data reader. For some database operations, such as creating and editing database structures, you cannot use a dataset. For example, if you want to create a new database table from within your application, you cannot do so using a dataset; instead, you would execute a data command. For general data-access scenarios, however, you can often choose between storing records in a disconnected dataset and working directly with the database using data commands.

> **Note** For details about datasets, see Introduction to Datasets. For details about accessing a database directly, see Introduction to DataCommand Objects in Visual Studio.

Each strategy has inherent advantages that apply to any data access scenario (not just Web Forms pages). For example, datasets make it easier to work with related tables and to work with data from disparate sources. Conversely, using a data reader often results in slightly better performance and memory usage, it eliminates the extra step (and memory requirement) of filling a dataset, and it allows you more direct control over the statements or stored procedures you use. For details about the comparative advantages of each of these approaches in general, see Recommendations for Data Access Strategies.

Datasets and Data Commands in Web Forms Pages

When you are working with Web Forms pages, additional factors come into play when deciding whether to use a dataset or a data reader. A significant factor is the page life cycle — Web Forms pages are initialized, processed, and then discarded with each round trip. If you simply want to display data on the page, then creating a dataset, filling it, and then binding controls to it can represent unnecessary overhead, since the dataset will be discarded immediately. In many cases, it is more efficient to use data reader to fetch the data and then to bind controls to it at run time.

> **Tip** No matter how you decide to get data from the data source, you should always attempt to minimize the amount of data you use in a page. The more data a query or stored procedure brings back, the more database and server resources you use.

In general, therefore, you might want to assume that in Web Forms pages it is usually better to use data commands to execute SQL statement or stored procedures, using a data reader to fetch data. For example, to display information in a **DataList** command, you can execute a SQL statement and then bind the control to a data reader. Exceptions to this general rule might be the following:

- **Working with related tables** Datasets allow you to maintain multiple related tables and include support for relations and referential integrity. Working with related records — for example, reading parent and corresponding child records — can be substantially simpler in a dataset than fetching records independently using when executing commands against a database.
- **Exchanging data with other processes** If your Web Forms page is getting its data from another component, such as an XML Web service, you will almost always use a dataset to hold the local copy of the data. Datasets automatically read and write the XML stream used to communicate between components in the .NET Framework.
- **Working with a static set of records** If you need to use the same set of records repeatedly — for example, when users are paging in a grid — it can be efficient to fill those records into a dataset rather than return to the data source with each round trip. This is particularly true if you want to set aside a particular set of records to work with from a database that is changing frequently.

A more general advantage of using a dataset is that it is easier to program than working directly with data commands. However, you must weigh this advantage carefully against other application requirements.

For details about using data commands, see Introduction to DataCommand Objects in Visual Studio.

Save Dataset or Recreate Each Time?

If you do use a dataset, your next choice is whether you want to recreate it with each round trip. You have two options:

- Each time the page is processed, create an instance of the dataset and fill it. When page processing is finished and the page is sent to the browser, the dataset is discarded.
- Create and fill the dataset once (typically, the first time the page runs). Then save the dataset in a way that you can retrieve it with each subsequent round trip.

Creating the dataset each time means that with each round trip — effectively, each time the user clicks a button on your page — you execute a query or stored procedure against the data source. For example, you might have a Web Forms page where the user wants to page through data. If you create the dataset each time, the Web Forms page executes a query against the data source to get the next set of records to display.

> **Tip** Remember to always minimize data transfer. Whenever practical, use selection criteria to reduce the number of records you need on a page.

If you save and restore the dataset, on the other hand, you do not need to go back to the source just to get a few more records. However, saving the dataset has a number of drawbacks. An important one is that the dataset consumes memory between round trips. If the dataset is very large, for example, it can take considerable server memory to store it. If multiple users create large datasets, you can quickly consume available server memory. (An option is to store data in the page; for details, see the next section.)

Another potential drawback is that the dataset can get out of sync with the data source, since you are not refreshing the dataset each time the user clicks a button. If you are working with very volatile data (inventory data, for instance), you might find it better for your application to recreate the dataset with each round trip.

Cache on the Server or on the Client?

If you decide to save a dataset between round trips, you must decide where to keep it. This issue is the standard one of state maintenance in Web Forms pages — where do you store information you want to preserve between round trips? For information about saving values, see Web Forms State Management.

You have two options:

- On the server, save the dataset in Session state, Application state, or using a cache.
- On the client — that is, in the page — save the dataset using view state or by putting the data into your own hidden field. (View state is also implemented using a hidden field.)

Storing the dataset on the server uses server resources. If you store too much data (a large dataset, or many users storing small datasets), it can affect server performance and scalability. Using a cache can partly offset this problem, because the cache manager will discard the dataset if the server needs memory or if cached data expires. But because the dataset is not guaranteed to be in the cache, you must add logic to your page to check that the dataset is available in the cache; if not, you must recreate it and put a copy back in the cache.

Storing data in the page means that you do not need server resources to save the data. However, the data becomes part of the HTML stream of the page. If the dataset is large, it can substantially affect the time it takes for the page to load into the user's browser and to post the page back to the server. For details about saving data in view state, see Saving Web Forms Page Values using View State.

> **Tip** Always try to keep the size of a dataset to a minimum by filling it with only the records you need.

No matter where you decide to store the dataset, you must add logic to your page to save it and restore it at the appropriate time. The following example shows a typical way to store and restore a dataset in Session state. The dataset `dsCustomers1` is an instance of the dataset class `dsCustomers`. Note that the dataset is stored in Session state as type **Object**. When restoring the dataset from Session state, you must cast it from **Object** back to a dataset class.

```
'Visual Basic
Private Sub Page_Load(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles MyB
ase.Load
   If Page.IsPostBack Then
      dsCustomers1 = CType(Session("myDsCustomers"), dsCustomers)
   Else
      If Session("myDsCustomers") Is Nothing Then
         OleDbDataAdapter1.Fill(dsCustomers1)
         Session("myDsCustomers") = dsCustomers1
```

```
        End If
    End If
End Sub

//C#
private void Page_Load(object sender, System.EventArgs e)
{
   // Put user code to initialize the page here
    if (Page.IsPostBack)
    {
        dsCustomers1 = (dsCustomers) Session["myDsCustomers"];
    }
    else
    {
        if (Session["myDsCustomers"] == null)
        {
            oleDbDataAdapter1.Fill(dsCustomers1);
            Session["myDsCustomers"] = dsCustomers1;
        }
    }
}
```

See Also

Web Forms State Management | Caching Application Data | Introduction to Distributed Applications and Data Integration | Introduction to Data Access in Web Forms Pages | ASP.NET Web Application Security

Data Walkthroughs for Web Forms Pages

The topics in this section demonstrate various data access strategies for Web applications.

In This Section

Walkthrough: Displaying Data in a Web Forms Page
Gives step-by-step details on displaying data in a Web Forms page using a dataset and common controls provided by ASP.NET.

Walkthrough: Creating Read-Only Data Access in a Web Forms Page
Gives step-by-step details on creating optimized data access in a Web Forms page intended for read-only display.

Walkthrough: Updating Data Using a Database Update Query in Web Forms
Gives step-by-step details on updating data using a database query in a Web Forms page.

Walkthrough: Using a DataGrid Web Control to Read and Write Data
Gives step-by-step details on creating a Web Forms page that updates data rather than merely displaying it.

Walkthrough: Creating Paged Data Access Using a Web Forms Page
Illustrates how to create a Web Forms page that lists data one page at a time.

Related Sections

Web Forms Data Binding
Provides links to topics that discuss how data binding works in Web Forms pages and how to specify data binding for various types of controls.

Visual Basic Code Example Topics
Shows quickstart-like code examples that help you perform common tasks in Visual Basic .NET.

Walkthrough: Displaying Data in a Web Forms Page

Web Forms pages provide you with a variety of ways to work with data. This walkthrough illustrates the simplest way.

In the walkthrough, you will display information from the Northwind Categories table in a data grid on the page. You will create a dataset — an in-memory cache of data — into which you will read information from the database. You will then bind a DataGrid Web server control to the dataset in order to display data.

> **Note** Using a dataset is only one option for data access in Web Forms, and it is not the optimal choice in some scenarios. You will use a dataset in this walkthrough for the sake of simplicity. For more information, see Web Data Access Strategy Recommendations.

In order to complete this walkthrough, you will need:

- Access to a server with the Northwind SQL Server sample database. The SQL Server can be located on the same computer as the Web server or on another server. In either case, you must configure access permissions. For details, see Accessing SQL Server from a Web Application.
- Sufficient permissions to be able to create ASP.NET Web application projects on the computer where your Web server is. For details, see Web Application Security at Design Time in Visual Studio.

The walkthrough is split into a number of smaller pieces:

- Creating the Web application project and a Web Forms page.
- Creating and configuring the dataset you will bind the grid to. This includes creating a query that populates the dataset from the database.
- Adding the **DataGrid** control to the form and binding it to data.
- Add code to fill the dataset.

Creating the Project and Form

The first step is to create a Web application and a Web Forms page.

To create the project and form

1. On the **File** menu, point to **New**, then click **Project**.
2. In the **New Project** dialog box, do the following:
   a. In the **Project Types** pane, choose either **Visual Basic Projects** or **Visual C# Projects**.
   b. In the **Templates** pane, choose **ASP.NET Web Application**.
   c. In the **Location** box, enter the complete URL for your application, including http://, the name of the server, and the name of your project. The Web server must have IIS version 5 or later and the .NET Framework installed on it. If you have IIS on your computer, you can specify `http://localhost` for the server. (If you normally use a proxy server to access the Internet, you might need to configure Internet Explorer to bypass the proxy server in order to use localhost.)

      When you click **OK**, a new Web Forms project is created at the root of the Web server you specified. In addition, a new Web Forms page called WebForm1.aspx is displayed in the Web Forms Designer in Design view.

      > **Tip** If you have trouble creating a Web application project, see Web Access Failed Dialog Box.

Creating and Configuring a Dataset

In this walkthrough, you will add a dataset to the page. The dataset will contain a single table, the Categories table. (A dataset can contain multiple tables, but in this walkthrough it will contain just one table.)

The dataset does not already exist. Instead of adding it manually to the form, you will perform the following set of steps:

- Create a data adapter using a wizard. The adapter contains SQL statements used to read and write database information. The wizard helps you define the SQL statements you need. If necessary, the wizard also creates a connection to the database.
- Generate the dataset schema. In this process, you will have Visual Studio create a new dataset class based on the tables and columns you are accessing. When you generate the dataset class, you will also add an instance of it to the form.

It is important that you follow all the procedures in this section. Otherwise your page will not have the dataset that you will be using in subsequent parts of the walkthrough.

For details on data adapters, see Introduction to Data Adapters. For details about datasets, see Introduction to Datasets.

Configuring a Data Connection and Data Adapter

To begin, you create a data adapter that contains the SQL statement used to populate the dataset later. As part of this process, you define a connection to access a database. You configure the data adapter using a wizard, which makes it easy to create the SQL statements you need for data access.

> **Note** When the wizard is done, you must continue to the next section in order to generate a dataset and complete the data access portion of your form.

To create the data connection and data adapter

1. From the **Data** tab of the Toolbox, drag an **OleDbDataAdapter** object onto the form.

   > **Note** You could also use the **SqlDataAdapter**, which is optimized for working with SQL Server 7.0 or later. In this walkthrough, you use the **OleDbDataAdapter** because it is more generic, providing ADO.NET access to any OLE DB-compatible data source.

   The **Data Adapter Configuration Wizard** starts, which will help you create both the connection and the adapter.

2. In the wizard, do the following:
   a. In the second pane, create or choose a connection pointing to the SQL Server Northwind database.

      > **Note** You need appropriate read/write permissions on the SQL Server you are using. If SQL Server is on the same computer as IIS, it is recommended that you specify Windows integrated security when creating the connection. Otherwise, you can specify a user name and password and save that information with the connection, but doing so can compromise security. For more information, see Accessing SQL Server from a Web Application.

   b. In the third pane, specify that you want to use a SQL statement to access the database.
   c. In the fourth pane, create the following SQL statement:

      ```
      SELECT CategoryID, CategoryName, Description
      FROM Categories
      ```

      For assistance building the SQL statement, click **Query Builder** to launch the **Query Builder** dialog box.

      > **Note** In this walkthrough, you will populate the dataset with all the rows from the categories table. In production applications, you typically optimize data access by creating a query that returns only the columns and rows you need. For an example, see Walkthrough: Displaying Data in a Windows Form Using a Parameterized Query.

   d. Click **Finish**.

      The wizard creates a connection (**OleDbConnection1**) containing information about how to access your database. You will also have a data adapter (**OleDbDataAdapter1**) that contains a query defining what table and columns in the database you want to access.

3. After the wizard is complete, generate the dataset based on the SQL query that you created during this procedure. For details, see the next section.

   > **Note** For more information, click **Help** while running the wizard or see Data Adapter Configuration Wizard. For details about data adapters, see Introduction to Data Adapters.

Creating the Dataset

After you have established the means to connect to the database and specified the information you want (via the SQL command in the data adapter), you can have Visual Studio create a dataset. Visual Studio can generate the dataset automatically based on the query you specified for the data adapter. The dataset is an instance of the **DataSet** class based on a corresponding schema (.xsd file) that describes the class's elements (table, columns, and constraints). For details about the relationship between datasets and schemas, see Introduction to Data Access with ADO.NET.

To generate a dataset

1. From the **Data** menu, choose **Generate DataSet**.

   Tip If you do not see the **Data** menu, click the form; the form must have focus for the menu to appear.

   The **Generate Dataset** dialog box appears.

2. Select the **New** option and name the dataset **dsCategories**.

   In the list under **Choose which table(s) to add to the dataset**, the categories table should be selected.

3. Check **Add this dataset to the designer**, then click **OK**.

   Visual Studio generates a typed dataset class (**dsCategories**) and a schema that defines the dataset. You will see the new schema (**dsCategories.xsd**) in Solution Explorer.

   Tip In Solution Explorer, click the **Show All Files** toolbar button to see the schema file's dependent .vb or .cs file, which contains the code the defines your new dataset class.

   Finally, Visual Studio adds an instance of the new dataset class (**DsCategories1**) to the form.

At this point you have set up everything you need in order to get information out of a database and into a dataset.

Adding a DataGrid to Display Data

In this walkthrough you will use a **DataGrid** Web server control to display information from the Categories table. You could also use a **DataList** or **Repeater** control to achieve the same results. However, the **DataGrid** control is the easiest control to configure (for example, you do not need to create any templates for it), which will allow you to concentrate on the data aspects of the walkthrough.

To add and configure the control

1. From the **Web Forms** tab of the Toolbox, drag a **DataGrid** control onto the page.
2. At the bottom of the Properties window, choose the **AutoFormat** link and select a predefined format for the grid.
3. In the **DataSource** property, select **DsCategories1** as the data source. This binds the grid to the dataset as a whole.
4. In the **DataMember** property, select **Categories**. If a data source contains more than one bindable object, you can use the **DataMember** property to specify which object to bind to.

   Setting these two properties binds the categories data table in the **DsCategories1** dataset to the grid.

Filling the Dataset and Displaying Data in the DataGrid Control

Although the grid is bound to the dataset you created, the dataset itself is not automatically filled in. Instead, you must fill the dataset yourself by calling a data-adapter method. For details about filling datasets, see Introduction to Datasets.

Even after the dataset is filled in, the **DataGrid** control does not automatically display the data. You must explicitly bind the grid to its data source. For more information, see Introduction to Data Binding in Web Forms Pages.

To fill the dataset and display data in the DataGrid control

1. Double-click the page to display the page's class file in the Code Editor.
2. In the **Page_Load** event handler, call the data adapter's **Fill** method, passing it the dataset you want to populate:

   ```
   OleDbDataAdapter1.Fill(DsCategories1)
   ```

3. Call the **DataGrid** control's **DataBind** method to bind the control to the dataset.

   You do not need to refill the dataset and bind the grid with each round trip. Once the **DataGrid** control has been populated with data, its values are preserved in view state each time the page is posted. Therefore, you only need to fill the dataset and bind the grid the first time the page is called. You can test for this using the page's IsPostBack property.

   The complete handler will look like the following:

   ```
   ' Visual Basic
   Private Sub Page_Load(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles
   MyBase.Load
      'Put user code to initialize the page here
   ```

```
    If Not IsPostBack Then
       OleDbDataAdapter1.Fill(DsCategories1)
       DataGrid1.DataBind()
    End If
End Sub

// C#
private void Page_Load(object sender, System.EventArgs e)
{
   // Put user code to initialize the page here
   if ( !IsPostBack)
   {
      oleDbDataAdapter1.Fill(dsCategories1);
      DataGrid1.DataBind();
   }
}
```

Testing

After adding the code, test data access in the Web Forms page.

To test the Web Forms page

1. Save the page.
2. In Solution Explorer, right-click the page and choose **View in Browser**.

   Confirm that a list of categories is displayed in the grid.

Next Steps

- Format the grid by changing its color, font, and so on.
- Display only selected information in the grid. In many instances, you will base the display on information that the user supplies at run time (for example, you might display authors only from a particular city). To do this, you create a parameterized query. For details, see Walkthrough: Displaying Data in a Windows Form Using a Parameterized Query.
- Use a data reader instead of a dataset as the source of data for the grid. In a page such as this one, where the data is read-only, it is often more efficient to get the data directly from the database rather than first creating and filling a dataset. For an example, see Walkthrough: Creating Read-Only Data Access in a Web Forms Page.
- Separate data access from the user interface. In this walkthrough, you have created a form that accesses the data more-or-less directly (via the dataset). A more flexible and maintainable design is to create a data-access component that handles data access. The form (that is, the user interface) could then interact with the component as needed. The same component could be used by multiple forms (and by other components), which eliminates the overhead and redundancy of redesigning data access for every form you create. For details on creating component-based data access, see Walkthrough: Creating a Distributed Application.
- Learn about the security issues associated with Web Forms. For details, see Overview of Web Application Security Threats.

See Also

Data Access in Web Forms Pages | Introduction to Datasets

Walkthrough: Creating Read-Only Data Access in a Web Forms Page

In Web applications, it is very common for a page to simply display data — that is, for the data on the page to be read-only. Typical examples include catalog listings, search results, and so on. Users might act on the data by clicking a button to add an item to a shopping cart or clicking a link on the page, but those actions do not directly affect the data on the page.

In this kind of scenario, data access performance can be optimized in a couple of ways. One is that the Web Forms page does not need to create and populate a dataset. If the page is going to simply display the data directly, there is no particular need to cache it in a dataset. Because of this, a second optimization is that the data components of the page can be reduced to a minimum: because you do not need a dataset, you also do not need a data adapter to populate it. In fact, the only data access components you need are a connection and a data command containing a SQL statement or stored procedure that can be executed to fetch the data.

The only remaining piece is a special data component, the data reader, which can fetch records from the result set and pass them to another component. The data reader is specifically designed to be a read-only, forward-only reader, so it is optimized for extremely fast data access. The data reader returns a structure to which you can bind list controls (**Repeater**, **DataList**, **DataGrid** Web server controls).

This walkthrough illustrates how to create a Web Forms page with an ASP.NET **DataGrid** server control on it that uses a data reader to display read-only data. When you are through, you will have a page that looks like the following:

Start Page | WebForm1.aspx | WebForm1.aspx.vb | **Browse - WebForm1**

au_id	au_fname	au_lname	phone
172-32-1176	Johnson	White	408 496-7223
213-46-8915	Margie	Green	415 986-7020
238-95-7766	Cheryl	Carson	415 548-7723
267-41-2394	Anne	Hunter	408 286-2428
274-80-9391	Dean	Straight	415 834-2919
341-22-1782	Jim	Smith	913 843-0462
409-56-7008	Abraham	Bennet	415 658-9932
472-27-2349	Burt	Gringlesby	707 938-6445
486-29-1786	Charlene	Locksley	415 585-4620
648-92-1872	Reginald	Blotchet-Halls	503 745-6402
672-71-3249	Akiko	Yokomoto	415 935-4228

In order to complete this walkthrough, you will need:

- Access to a server with the Pubs SQL Server sample database. The SQL Server can be located on the same computer as the Web server or on another server. In either case, you must configure access permissions. For details, see Accessing SQL Server from a Web Application.
- Sufficient permissions to be able to create ASP.NET Web application projects on the computer where your Web server is. For details, see Web Application Security at Design Time in Visual Studio.

The walkthrough is split into a number of smaller pieces:

- Creating the Web Forms page.
- Adding the necessary data components.
- Adding the **DataGrid** control to display the data and setting a few of its properties.
- Adding the few lines of code needed to fetch the data and display it in the control.

Creating the Project and Form

The first step is to create a Web application and a Web Forms page.

To create the project and form

1. On the **File** menu, point to **New**, then click **Project**.
2. In the **New Project** dialog box, do the following:
   a. In the **Project Types** pane, choose either **Visual Basic Projects** or **Visual C# Projects**.
   b. In the **Templates** pane, choose **ASP.NET Web Application**.
   c. In the **Location** box, enter the complete URL for your application, including http://, the name of the server, and the name of your project. The Web server must have IIS version 5 or later and the .NET Framework installed on it. If you

have IIS on your computer, you can specify `http://localhost` for the server. (If you normally use a proxy server to access the Internet, you might need to configure Internet Explorer to bypass the proxy server in order to use localhost.)

When you click **OK**, a new Web Forms project is created at the root of the Web server you specified. In addition, a new Web Forms page called WebForm1.aspx is displayed in the Web Forms Designer in Design view.

> **Tip** If you have trouble creating a Web application project, see Web Access Failed Dialog Box.

Adding the Data Components

To create read-only access to the data, you will need a connection to the database you want to read from. You will also need a data command. The data command includes a SQL statement you can execute to fetch the data. (The data command can reference a stored procedure instead, but in this walkthrough you will create a simple SQL SELECT statement.)

Creating and Configuring the Connection

To begin, add a connection that points to the database you want to read from.

To create the data connection

1. From the **Data** tab of the Toolbox, drag an **OleDbConnection** object onto the page.

   > **Note** You could also use the **SqlConnection** object, which is optimized for working with SQL Server 7.0 or later. In this walkthrough, you use the **OleDbConnection** because it is more generic, providing ADO.NET access to any OLE DB-compatible data source.

2. Select the connection and in the Properties window, click the button in the **ConnectionString** property box.

   The list displays any existing connections and the **<New Connection...>** option.

3. Unless you already have a connection to the SQL Server Pubs database, choose **<New Connection...>**.

   The **Data Link Properties** dialog box is displayed.

4. Create a connection pointing to your SQL Server and the Pubs database, and then click **OK**.

   > **Tip** For assistance in creating a connection, press F1 in the **Data Link Properties** dialog box.

   > **Note** You need appropriate read/write permissions on the SQL Server you are using. If SQL Server is on the same computer as IIS, it is recommended that you specify Windows integrated security when creating the connection. Otherwise, you can specify a user name and password and save that information with the connection, but doing so can compromise security. For more information, see Accessing SQL Server from a Web Application.

   The connection information you specified is saved as the connection string for your connection.

That is all that you need to do for the connection. Now you need to add a command with a SQL statement to fetch the data.

To create the data command and SQL statement

1. From the **Data** tab of the Toolbox, drag an **OleDbCommand** object onto the page.

   > **Note** You could also use the **SqlCommand**, which is optimized for working with SQL Server 7.0 or later. In this walkthrough, you use the **OleDbCommand** because it is more generic, providing ADO.NET access to any OLE DB-compatible data source.

2. Select the command and in the Properties window, click the button in the **Connection** property, and select the connection you made earlier. (You will need to open the **Existing** node.)
3. In the **CommandText** property, click the ellipsis button to open the **Query Builder** dialog box.
4. Use the **Query Builder** dialog box to construct the following SQL statement:

```
Select au_id, au_fname, au_lname, phone
From authors
```

   > **Note** In this walkthrough, you will populate the dataset with all the rows from the authors table. In production applications, you typically optimize data access by creating a query that returns only the columns and rows you

need. For an example, see Walkthrough: Displaying Data in a Windows Form Using a Parameterized Query.

5. When you have finished, close the **Query Builder** dialog box.

   The SQL statement is filled into the **CommandText** property.

You have finished adding data components to the page. You can now add a control to display the data.

Adding a Control to Display Data

In this walkthrough you will use an **DataGrid** Web server control to display information from the authors table. You could also use a **DataList** or **Repeater** control to achieve the same results. However, the **DataGrid** control is the easiest control to configure (for example, you do not need to create any templates for it), which will allow you to concentrate on the data aspects of the walkthrough.

To add and configure the control

1. From the **Web Forms** tab of the Toolbox, drag a **DataGrid** control onto the page.
2. At the bottom of the Properties window, choose the **AutoFormat** link and select a predefined format for the grid.

If you have worked with a data-bound **DataGrid** control before, you might expect to set the control's **DataSource** property. In this example, however, you will not set the data source here; instead, you will set it using code to point to the data reader.

You will also note that you are not explicitly setting the grid columns. In this example, data binding is established at run time for the grid, and the grid will dynamically create and display columns based on the data it is bound to.

> **Note** If you are familiar with the **DataGrid** control, you can pre-format columns for the four columns (ID, first name, last name, and phone). Because this is not a required step (the grid will automatically create columns at run time when it is bound), the walkthrough does not guide you through this procedure.

Adding Code to Fetch and Display the Data

To create optimized read-only data access, you will execute the SQL statement you created with the data command. The command returns a data reader that you can then bind to. You can accomplish all of this with a few lines of code.

To add code to fetch data

1. Double-click the Web Forms page to open the Code Editor and find the **Page_Load** event handler.
2. Create code to perform the following functions:
   a. Create an instance variable of type **OleDbDataReader**.
   b. Open the connection.
   c. Call the data command's **ExecuteReader** method, which executes the SQL statement in the data command and returns its result set into the data reader object.
   d. Set the **DataGrid** control's data source to point to the data reader. You could not do this earlier in the Properties window because the data reader did not exist then.
   e. Bind the **DataGrid** control to the data reader.
   f. Close the data reader.
   g. Close the connection. This is an important step; the connection remains open while the data is being fetched from the database. Therefore, when you no longer need the connection, you want to close it and free the resources it uses.

   The following example shows the code required to perform the above steps.

   > **Note** For clarity, the code explicitly creates a variable to hold the data reader.

```
' Visual Basic
Private Sub Page_Load(ByVal sender As System.Object, ByVal e As System.EventArgs) Ha
ndles MyBase.Load
   If Not IsPostBack Then
      oleDbConnection1.Open()
      Dim Reader As System.Data.OleDb.OleDbDataReader
      Reader = oleDbCommand1.ExecuteReader()
      DataGrid1.DataSource = Reader
      DataGrid1.DataBind()
```

```
        Reader.Close()
        oleDbConnection1.Close()
    End If
End Sub

// C#
private void Page_Load(object sender, System.EventArgs e)
{
    if (!IsPostBack) {
        oleDbConnection1.Open();
        System.Data.OleDb.OleDbDataReader Reader;
        Reader = oleDbCommand1.ExecuteReader();
        DataGrid1.DataSource = Reader;
        DataGrid1.DataBind();
        Reader.Close();
        oleDbConnection1.Close();
    }
}
```

Testing

After adding the code, test data access in the Web Forms page.

To test the Web Forms page

1. Save the page.
2. In Solution Explorer, right-click the page and choose **View in Browser**.

   Confirm that a list of authors is displayed in the grid.

Next Steps

This walkthrough has illustrated the basic steps involved in using a data reader to display read-only data in the page. Some enhancements you could make to the form in this walkthrough include:

- Format columns in the grid to display the information that will be available at run time.
- Format the grid by changing its color, font, and so on.
- Display data in a different control, such as a **DataList** or **Repeater** control. The process is largely the same. One difference, however, is that you need to create templates, drag controls into the templates, and bind the controls. Because the data binding of the container control is done at run time, you have to create custom binding expressions for the controls in the templates. A typical custom binding expression looks like this:

  ```
  DataBinder.Eval(Container, "DataItem.au_lname")
  ```

  The expression would vary only in the last part (`au_lname`) for each control in the template.

  For information about templates, see Web Server Controls Templates.

- Use a parameterized query to fetch data, which is a very common real-world scenario. In that case, you would create a SQL statement that accepts parameters, and then pass those parameters by filling in the **Parameters** collection of a data command before executing it. For details, see Setting and Getting Data Command Parameters.
- Add paging. This is a variation of a parameterized query. You need to maintain in the page some indication of what records you have displayed. When the user navigates to the next page, you need to pass parameter information with the query to get the next page worth of data. For details on creating paged output, see Paging Behavior in DataGrid Web Server Controls.
- Learn about the security issues associated with Web Forms. For details, see Overview of Web Application Security Threats.

See Also

Introduction to DataCommand Objects in Visual Studio | Introduction to the DataGrid Web Server Control | OleDbDataReader Class | SqlDataReader Class

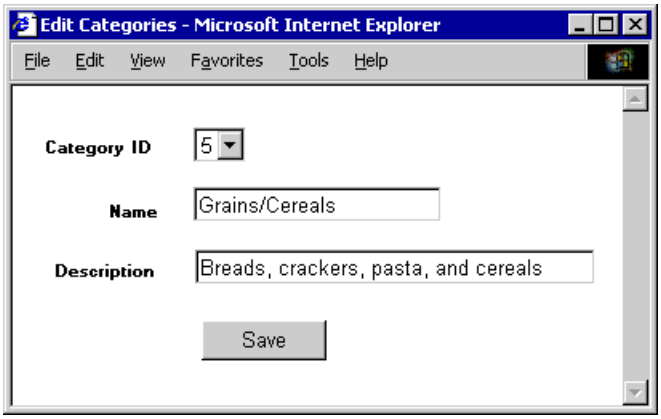
Walkthrough: Updating Data Using a Database Update Query in Web Forms

The Web Forms data-binding architecture allows you to bind control properties to a data source, making it easy to display data on the page. However, the data-binding architecture does not support updates — that is, data binding is one-way in Web Forms pages. Because a majority of Web applications involve reading data and not updating it, one-way data binding keeps the size and complexity of the page architecture to a minimum for the most common scenarios.

However, there are times when you do want to update data. There are several ways to do so. If you are using a dataset, you can write changes to the dataset and then call the data adapter to send changes from the dataset to the database. (For an example of this scenario, see Walkthrough: Using a DataGrid Web Control to Read and Write Data.) Alternatively, you can update the database directly, using SQL statements or stored procedures, which is what this walkthrough illustrates.

> **Note** For details about choosing between these approaches, see Web Data Access Strategy Recommendations.

This walkthrough illustrates how to create a Web Forms page that reads and writes data to a database using SQL statements. For the walkthrough, you will work with the Categories table of the SQL Server Northwind database. In the page, you can select a record, edit it, and then save it to the database. When you are done, your page will look something like the following:



The walkthrough will illustrate a number of concepts:

- Creating command objects that incorporate SQL statements
- Using a SQL statement and a data reader to fetch data from the database
- Creating parameterized queries
- Setting query parameters and executing queries

In order to complete this walkthrough, you will need:

- Access to a server with the Northwind SQL Server sample database. The SQL Server can be located on the same computer as the Web server or on another server. In either case, you must configure access permissions. For details, see Accessing SQL Server from a Web Application.
- Sufficient permissions to be able to create ASP.NET Web application projects on the computer where your Web server is. For details, see Web Application Security at Design Time in Visual Studio.

The walkthrough is split into a number of smaller pieces:

- Creating the Web Forms page.
- Adding the necessary data components.
- Adding controls to display the data.
- Adding code to read and write data.

Creating the Project and Form

The first step is to create a Web application and a Web Forms page.

To create the project and form

1. On the **File** menu, point to **New**, then click **Project**.
2. In the **New Project** dialog box, do the following:

a. In the **Project Types** pane, choose either **Visual Basic Projects** or **Visual C# Projects**.
b. In the **Templates** pane, choose **ASP.NET Web Application**.
c. In the **Location** box, enter the complete URL for your application, including http://, the name of the server, and the name of your project. The Web server must have IIS version 5 or later and the .NET Framework installed on it. If you have IIS on your computer, you can specify `http://localhost` for the server. (If you normally use a proxy server to access the Internet, you might need to configure Internet Explorer to bypass the proxy server in order to use localhost.)

   When you click **OK**, a new Web Forms project is created at the root of the Web server you specified. In addition, a new Web Forms page called WebForm1.aspx is displayed in the Web Forms Designer in Design view.

   Tip If you have trouble creating a Web application project, see Web Access Failed Dialog Box.

Adding the Data Components

To access the data, you will need a connection to the database you want to read from. You will also need several command objects — two to read data from the database using SQL Select statements, and a third to update the database with a SQL Update statement. Two of these commands will require parameters, so that you can pass run-time values to the statements before executing them. (The data commands can reference a stored procedure instead of SQL statements, but in this walkthrough you will use statements.)

Creating and Configuring the Connection

To begin, add a connection that points to the database you want to read from.

To create the data connection

1. From the **Data** tab of the Toolbox, drag a **SqlConnection** object onto the page.

   Note Because you are using SQL Server, you use the **SqlConnection** object, which is optimized for working with SQL Server 7.0 or later. If you are using a different database, you use the **OleDbConnection**, which provides ADO.NET access to any OLE DB-compatible data source.

2. Select the connection and in the Properties window, click the button in the **ConnectionString** property box.

   The list displays any existing connections and the **<New Connection...>** option.

3. Unless you already have a connection to the SQL Server Northwind database, choose **<New Connection...>**.

   The **Data Link Properties** dialog box is displayed.

4. Create a connection pointing to your SQL Server and the Northwind database, and then click **OK**.

   Note You need appropriate read/write permissions on the SQL Server you are using. If SQL Server is on the same computer as IIS, it is recommended that you specify Windows integrated security when creating the connection. Otherwise, you can specify a user name and password and save that information with the connection, but doing so can compromise security. For more information, see Accessing SQL Server from a Web Application.

   The connection information you specified is saved as the connection string for your connection.

Now you need to add commands. You will need three in all, each with a different query:

- One to get all the category ID values.
- One to get a selected category based on an ID that you provide.
- One to update a category with new values.

To create the data command for getting all category ids

1. From the **Data** tab of the Toolbox, drag an **SqlCommand** object onto the page.
2. Rename the command **cmdCategoriesAll**.
3. Select the command and in the Properties window, click the button in the **Connection** property, and select the connection you made earlier. (You will need to open the **Existing** node.)
4. In the **CommandText** property, click the ellipsis button to open the **Query Builder** dialog box.
5. Use the **Query Builder** dialog box to construct the following SQL statement:

```
SELECT CategoryID, CategoryName, Description FROM Categories
```

6. When you have finished, close the **Query Builder** dialog box.

   The SQL statement is filled into the **CommandText** property.

The remaining commands are slightly more complex, in that they require parameters.

To create the data command for getting a single category record

1. Drag a second **SqlCommand** object onto the page and set its **Connection** property to **SqlConnection1**.
2. Name the command **cmdCategoriesById**.
3. In the **CommandText** property, click the ellipsis button to open the **Query Builder** dialog box.
4. Use the **Query Builder** dialog box to construct the following SQL statement:

```
SELECT CategoryID, CategoryName, Description
FROM Categories
WHERE (CategoryID = @categoryid)
```

> **Note** The variable @categoryid is a "named variable." The exact name of the variable you use in the Where clause is not important, but you will need to know it later in the walkthrough. The "@" prefix is required for SQL Server named parameters.

The final command includes an Update statement for sending changes to the database.

To create the data command for updating a category record

1. Drag a third **SqlCommand** object onto the page and set its **Connection** property to **SqlConnection1**.
2. Name the command **cmdCategoriesUpdate**.
3. Set the command's **CommandText** property to the following statement:

```
UPDATE Categories
SET CategoryName = @categoryname, Description = @categorydescription WHERE (CategoryID =
@categoryid)
```

You have finished adding data components to the page.

Adding Controls

In the page, you will need a drop-down list to allow users to select a category ID, some text boxes to display and edit data with, and a button to save changes.

To add controls to the form

1. From the **Web Forms** tab of the Toolbox, add the following controls to the form and set the specified properties as indicated:

Control	Purpose	Properties
DropDownList	Allows user to select a category list that you will provide at run time.	ID: **ddlCategoryID** AutoPostBack: **true**
TextBox	Allows users to edit the category name.	ID: **txtCategoryName**
TextBox	Allows users to edit the category description.	ID: **txtCategoryDescription**
Button	Updates the database with changes.	ID: **btnSave** Text: **Save**

2. Add labels in front of the controls to indicate their function.

> **Tip** If you are working in grid layout mode, you can add literal text using the Flow Layout Panel (HTML Div Controls) from the **HTML** tab of the Toolbox.

Adding Code to Display and Update the Data

To finish the page, you will need to add code to accomplish three tasks:

- When the page is initialized, you need to populate the drop-down list with the category IDs. Since the first category ID will be automatically selected, you can also display the first category name and description in the text boxes.
- When users select a category ID from the drop-down list, you need to determine which ID they selected. You then use that value to execute the SQL statement that brings back a single record.
- When users click the Save button, you need to get their changes and use them to execute an Update statement with the new values.

Populating the Drop-Down List at Page Initialization

To populate the drop-down list, you will execute a query that returns all the records in the Categories table. You only need to do this the first time the page runs; thereafter, the values in the list are preserved in the control's view state.

After executing the query, you will use a data reader (SqlDataReader object) to fetch the records. The data reader is specifically designed to be a read-only, forward-only reader, so it is optimized for extremely fast data access.

To add code to populate the drop-down list

1. Double-click the Web Forms page to open the Code Editor and find the **Page_Load** event handler.
2. Create code to perform the following functions:
   a. Test the page's **IsPostBack** property to determine if this is the first time the page is running. If so, perform the rest of these steps.
   b. Create an instance variable of type **SqlDataReader**.
   c. Open the connection.
   d. Call the **ExecuteReader** method of the **cmdCategoriesAll** object, which executes the SQL statement in the data command and returns its result set to the data reader object.
   e. Provide a way, such as a Boolean flag, to know when you are fetching the first record, because you want to display the information from the record only.
   f. In a loop, use the data reader to fetch each successive record.
   g. For each record, create a new item in the drop-down list and assign the category ID to it.
   h. For the first record only, get the category name and description and set them to the **Text** property of the text boxes.
   i. Close the data reader.
   j. Close the connection. This is an important step; the connection remains open while the data is being fetched from the database. Therefore, when you no longer need the connection, you want to close it and free the resources it uses.

   The following example shows the complete **Page_Load** method with the code required to perform the above steps:

```
' Visual Basic
Private Sub Page_Load(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles
MyBase.Load
   If Not IsPostBack Then
      ' Executes only the first time the page is processed. After
      ' that, the list is already in the drop-down list and is
      ' preserved in view state.
      Dim dreader As SqlClient.SqlDataReader
      SqlConnection1.Open()
      dreader = cmdCategoriesAll.ExecuteReader()
      Dim firstrow As Boolean = True
      While dreader.Read()
         ddlCategoryID.Items.Add(New ListItem(dreader(0).ToString()))
         If firstrow Then
            txtCategoryName.Text = dreader(1).ToString()
            txtCategoryDescription.Text = dreader(2).ToString()
            firstrow = False
         End If
      End While
      dreader.Close()
      SqlConnection1.Close()
   End If
```

```
End Sub

// C#
private void Page_Load(object sender, System.EventArgs e)
{
   if (!IsPostBack)
   {
      // Executes only the first time the page is processed. After
      // that, the list is already in the drop-down list and is
      // preserved in view state.
      System.Data.SqlClient.SqlDataReader dreader;
      sqlConnection1.Open();
      dreader = cmdCategoriesAll.ExecuteReader();
      bool firstrow = true;
      while (dreader.Read())
      {
         ddlCategoryID.Items.Add(new ListItem(dreader[0].ToString()));
         if (firstrow)
         {
            txtCategoryName.Text = dreader[1].ToString();
            txtCategoryDescription.Text = dreader[2].ToString();
            firstrow = false;
         }
      }
      dreader.Close();
      sqlConnection1.Close();
   }
}
```

Displaying a Record When Users Select a Category ID

The drop-down list displays category IDs. When the user selects one — that is, when the control's **SelectedIndexChanged** event is raised — you want to display the corresponding record.

When you created the drop-down list, you set its **AutoPostBack** property to **true**. The result is that the page is posted to the server as soon as the user makes a selection.

To get a category record, you will execute a SQL statement that uses a Where clause to find only the category whose ID the user selected. To pass this value to the statement, you set the value of a parameter (a **SqlParameter** object) that is part of the data command.

To display data for the selected category

1. Double-click the drop-down list to open the Code Editor and create the **SelectedIndexChanged** event handler.
2. Create code to perform the following functions:
   a. Get the text of the user's selection from the drop-down list.
   b. Set the parameter (@categoryid) of the **cmdCategoriesById** command to the selected ID.
   c. Open the connection.
   d. Create a data reader instance variable and call the **ExecuteReader** method of the **cmdCategoriesById** object to execute the command. In this case, you will pass a flag to the method indicating that you want to return only a single record.
   e. Use the data reader to fetch the returned record and then display its values in the text boxes.
   f. Close the data reader.
   g. Close the connection.

   The following example shows the complete **SelectedIndexChanged** method with the code required to perform the above steps:

```
' Visual Basic
Private Sub ddlCategoryID_SelectedIndexChanged(ByVal sender As System.Object, ByVal e As
```

```
System.EventArgs) Handles ddlCategoryID.SelectedIndexChanged
   Dim categoryid As String
   categoryid = ddlCategoryID.SelectedItem.Text
   cmdCategoriesById.Parameters("@categoryid").Value = categoryid
   SqlConnection1.Open()
   Dim dreader As SqlClient.SqlDataReader
   dreader = cmdCategoriesById.ExecuteReader(CommandBehavior.SingleRow)
   If dreader.Read() Then
      txtCategoryName.Text = dreader(1)
      txtCategoryDescription.Text = dreader(2)
   End If
   dreader.Close()
   SqlConnection1.Close()
End Sub

// C#
private void ddlCategoryID_SelectedIndexChanged(object sender, System.EventArgs e)
{
   string categoryid;
   categoryid = ddlCategoryID.SelectedItem.Text;
   cmdCategoriesById.Parameters["@categoryid"].Value = categoryid;
   sqlConnection1.Open();
   System.Data.SqlClient.SqlDataReader dreader;
   dreader = cmdCategoriesById.ExecuteReader(CommandBehavior.SingleRow);
   if (dreader.Read())
   {
      txtCategoryName.Text = dreader[1].ToString();
      txtCategoryDescription.Text = dreader[2].ToString();
   }
   dreader.Close();
   sqlConnection1.Close();
}
```

Updating a Record

Finally, you want to add code that will update the database with changes from the record displayed in the page. As with fetching a single record, the update requires that you set some parameters and then execute a statement (in this case, an Update statement). However, because the Update statement does not return any records, the code for executing the statement is relatively simple.

> **Security Note** User input in a Web Forms page can include potentially malicious client script. By default, the Web Forms page validates that user input does not include script or HTML elements. For more information, see Scripting Exploits and Protecting Against Script Exploits in a Web Application.

To update records

1. Double-click the Save button to create a **Click** handler.
2. Create code to perform the following functions:
   a. Set parameters (three in all) to the values of the controls in the page.
   b. Open the connection.
   c. Call the **ExecuteNonQuery** method of the **cmdCategoriesUpdate** command.
   d. Close the connection.

The following example shows the complete **Click** method with the code required to perform the above steps:

```
' Visual Basic
Private Sub btnSave_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Han
dles btnSave.Click
   With cmdCategoriesUpdate
      .Parameters("@categoryid").Value = ddlCategoryID.SelectedItem.Text
```

```
        .Parameters("@categoryname").Value = txtCategoryName.Text
        .Parameters("@categorydescription").Value = txtCategoryDescription.Text
    End With
    SqlConnection1.Open()
    cmdCategoriesUpdate.ExecuteNonQuery()
    SqlConnection1.Close()
End Sub

// C#
private void btnSave_Click(object sender, System.EventArgs e)
{
    cmdCategoriesUpdate.Parameters["@categoryid"].Value = ddlCategoryID.SelectedItem.Text;
    cmdCategoriesUpdate.Parameters["@categoryname"].Value = txtCategoryName.Text;
    cmdCategoriesUpdate.Parameters["@categorydescription"].Value = txtCategoryDescription.
Text;
    sqlConnection1.Open();
    cmdCategoriesUpdate.ExecuteNonQuery();
    sqlConnection1.Close();
}
```

Testing

You can now test the form to make sure it displays Categories data and that users can make updates.

To test the page

1. In Solution Explorer, right-click the page and choose **View in Browser**.
2. When the page is displayed, make sure that the drop-down list contains a list of category IDs and that the first category is displayed in the text boxes.
3. Select a different category from the list.

   The new record will be displayed.

   > **Note** If selecting a new category ID in the drop-down list does not change the record, make sure that the control's **AutoPostBack** property is set to **true**.

4. Make a change in one of the text boxes, and then click the **Save** button.
5. Move to a different category record, and then back to the one you changed.

   The changes you made are visible when you redisplay the record.

Next Steps

This walkthrough has illustrated the basic steps involved in using a data reader to fetch records directly from the database and using a data command to make updates. To enhance the Web Forms page you have created, you might want to explore some of the following ideas:

- Provide an alternative way for users to select categories. In this walkthrough, the drop-down list displayed category IDs, which ordinarily are not very useful to users. This was done to allow users to edit the name, if desired, and to maintain clarity for how information from database records was moving into and out of controls.

  A relatively easy change would be to display category names as the text of the drop-down list (the information that is visible in the list) and category IDs as the value of the list. (However, this would not allow users to edit the name.) You could then pass the value as a parameter value instead of passing the text.

- Add error checking. Typically, you want to make your code robust by enclosing all data-access statements in try-catch blocks. Data access is subject to many possible problems (many of them out of the control of your application, such as network problems). For details, see Best Practices for Throwing Exceptions from Components.
- Add concurrency control. In this walkthrough, you simply update a record in the database without determining first whether another user might have also made changes. For details, see Concurrency Control in ADO.NET.
- Learn about the security issues associated with Web Forms. For details, see Overview of Web Application Security Threats.

See Also

Walkthrough: Creating Read-Only Data Access in a Web Forms Page | Introduction to ADO.NET Connection Design Tools | Introduction to DataCommand Objects in Visual Studio | Executing a Data Command That Returns a Result Set | Executing Updates or Database Commands using a Data Command | Setting and Getting Data Command Parameters | Concurrency Control in ADO.NET

Walkthrough: Using a DataGrid Web Control to Read and Write Data

The architecture of Web Forms data binding makes it easy to display data in page controls. However, data binding is not two-way — that is, data binding reads data from a data source but does not update it. Updating is more complex than displaying data, and because the majority of Web Forms pages do not require that data be written back to the source, Web Forms data binding keeps page sizes and page processing to a minimum by not including update code.

Of course, there are times when you do want to create a Web Forms page that updates data. This walkthrough illustrates one way to do that. It shows how you can use a **DataGrid** control to display data, allow users to edit it, and then send the changed data back to the source.

> **Note** The **DataGrid** control does not inherently include a facility for creating new records. The section "Next Steps" at the end of this topic includes suggestions for adding this ability.

When the page is running, it will look something like the following:

	CategoryID	CategoryName	Description
Edit	1	Beverages	Soft drinks, coffees, teas, beers, and ales
Edit	2	Condiments	Sweet and savory sauces, relishes, spreads, and seasonings
Edit	3	Confections	Desserts, candies, and sweet breads
Edit	4	Dairy Products	Cheeses
Edit	5	Grains/Cereals	Breads, crackers, pasta, and cereal
Edit	6	Meat/Poultry	Prepared meats
Edit	7	Produce	Dried fruit and bean curd
Edit	8	Seafood	Seaweed and fish

In order to complete this walkthrough, you will need:

- Access to a server with the Northwind SQL Server sample database. The SQL Server can be located on the same computer as the Web server or on another server. In either case, you must configure access permissions. For details, see Accessing SQL Server from a Web Application.
- Sufficient permissions to be able to create ASP.NET Web application projects on the computer where your Web server is. For details, see Web Application Security at Design Time in Visual Studio.

The Overall Process

The walkthrough is split into a number of smaller pieces:

- Creating the Web Forms page.
- Adding the necessary data components.
- Adding the **DataGrid** control to display the data.
- Adding code to read data from the database and bind the grid to the data.
- Configuring the **DataGrid** control to allow users to edit data.
- Adding the code to update the data.

Creating the Project and Form

The first step is to create a Web application and a Web Forms page.

To create the project and form

1. On the **File** menu, point to **New**, then click **Project**.
2. In the **New Project** dialog box, do the following:
   a. In the **Project Types** pane, choose either **Visual Basic Projects** or **Visual C# Projects**.
   b. In the **Templates** pane, choose **ASP.NET Web Application**.
   c. In the **Location** box, enter the complete URL for your application, including http://, the name of the server, and the name of your project. The Web server must have IIS version 5 or later and the .NET Framework installed on it. If you have IIS on your computer, you can specify `http://localhost` for the server. (If you normally use a proxy server to access the Internet, you might need to configure Internet Explorer to bypass the proxy server in order to use localhost.)

When you click **OK**, a new Web Forms project is created at the root of the Web server you specified. In addition, a new Web Forms page called WebForm1.aspx is displayed in the Web Forms Designer in Design view.

> **Tip** If you have trouble creating a Web application project, see Web Access Failed Dialog Box.

Creating and Configuring a Dataset

In Web Forms pages, you have several choices for how you can access data. One is to use a dataset, which is an in-memory data cache. Alternatively, you can access a database directly, using data commands to execute SQL statements or stored procedures. As a general rule, it is easier to update data if you use a dataset, and that is what you will use in this walkthrough. For more information, see Web Data Access Strategy Recommendations.

You do not directly add a dataset to a page. Instead, you will perform the following set of steps:

1. Create a data adapter using a wizard. The adapter contains SQL statements used to read and write database information. The wizard helps you define the SQL statements you need. If necessary, the wizard also creates a connection to the database.
2. Generate the dataset schema. In this process, you will have Visual Studio create a new dataset class based on the tables and columns you are accessing. When you generate the dataset class, you will also add an instance of it to the form.

It is important that you follow all the procedures in this section. Otherwise your page will not have the dataset that you will be using in subsequent parts of the walkthrough.

For an overview of data adapters, see Introduction to Data Adapters. For an overview of datasets, see Introduction to Datasets.

Configuring a Data Connection and Data Adapter

To begin, you create a data adapter that contains the SQL statement used to populate the dataset later. As part of this process, you define a connection to access a database. You configure the data adapter using a wizard, which makes it easy to create the SQL statements you need for data access.

> **Note** When the wizard is done, you must continue to the next section in order to generate a dataset and complete the data access portion of your page.

To create the data connection and data adapter

1. Click WebForm1.aspx, and then from the **Data** tab of the Toolbox, drag a **SqlDataAdapter** object onto the page.

   > **Note** If you are not working with SQL Server, you use an adapter of type **OleDbDataAdapter**, which provides access to any OLE DB-compatible data source.

   The **Data Adapter Configuration Wizard** starts, which will help you create both the connection and the adapter.

2. In the wizard, do the following:
   a. In the second pane, create or choose a connection pointing to the SQL Server Northwind database. For information about getting access to the database, see your database administrator.

      > **Note** You need appropriate read/write permissions on the SQL Server you are using. If SQL Server is on the same computer as IIS, it is recommended that you specify Windows integrated security when creating the connection. Otherwise, you can specify a user name and password and save that information with the connection, but doing so can compromise security. For more information, see Accessing SQL Server from a Web Application.

   b. In the third pane, specify that you want to use a SQL statement to access the database.
   c. In the fourth pane, create the following SQL statement:

      ```
      SELECT CategoryID, CategoryName, Description
      FROM Categories
      ```

      For assistance building the SQL statement, click **Query Builder** to launch the **Query Builder** dialog box.

      > **Note** In this walkthrough, you will populate the dataset with all the rows from the categories table. In production applications, you typically optimize data access by creating a query that returns only the columns and rows you need. For an example, see

Walkthrough: Displaying Data in a Windows Form Using a Parameterized Query.

d. Click **Finish**.

The wizard creates a connection (**SqlConnection1** or **sqlConnection1**) containing information about how to access your database. You will also have a data adapter (**SqlDataAdapter1** or **sqlDataAdapter1**) that contains a query defining what table and columns in the database you want to access.

After the wizard is complete, you need to generate the dataset based on the SQL query that you created during this procedure. For details, see the next section.

Creating the Dataset

After you have established the means to connect to the database and specified the information you want (via the SQL command in the data adapter), you can have Visual Studio create a dataset. Visual Studio can generate the dataset automatically based on the query you specified for the data adapter. The dataset is an instance of the **DataSet** class based on a corresponding schema (.xsd file) that describes the class's elements (table, columns, and constraints). For details about the relationship between datasets and schemas, see Introduction to Data Access with ADO.NET.

To generate a dataset

1. From the **Data** menu, choose **Generate DataSet**.

   Tip If the **Generate DataSet** command is not enabled, click the page; the page must have focus for the command to appear.

   The **Generate Dataset** dialog box appears.

2. Select the **New** option and name the dataset **dsCategories**.

   In the list under **Choose which table(s) to add to the dataset**, make sure the Categories table is selected.

3. Make sure **Add this dataset to the designer** is checked, and then click **OK**.

   Visual Studio generates a typed dataset class (**dsCategories**) and a schema that defines the dataset. You will see the new schema (**dsCategories.xsd**) in Solution Explorer.

   Tip In Solution Explorer, click the **Show All Files** toolbar button to see the schema file's dependent .vb or .cs file, which contains the code that defines your new dataset class.

   Finally, Visual Studio adds an instance of the new dataset class (**dsCategories1**) to the page.

At this point you have set up everything you need in order to get information out of database and into a dataset.

Adding a DataGrid Control to Display the Data

In this walkthrough, you will add a single control — a **DataGrid** control — that can display all the records from the dataset at the same time and allows you to add the ability to edit records.

The data grid must be bound to the dataset in order to display the data.

To add a bound DataGrid control to the form

1. If you have not already done so, switch to the Web Forms Designer by clicking the tab at the top of the current window.
2. From the **Web Forms** tab of the Toolbox, drag a **DataGrid** control onto the form.
3. Select the control, press F4 to display the Properties window, and at the bottom of the window, click **Property Builder**.

   The **DataGrid1 Properties** dialog box appears.

4. In the **General** tab, make the following settings:

Property	Setting	Description
DataSource	dsCategories1	Binds the grid to the dataset.
DataMember	Categories	Specifies that the grid should display data from the categories table in the dataset
Data key field	CategoryID	Specifies that the primary key for categories record is the CategoryID column. This will allow you later to identify which record to update in the dataset.

5. Click **OK** to close the **DataGrid1 Properties** dialog box.
6. If you want to change the appearance of the grid, set the **Font**, **BackColor**, and other properties.

   Tip An easy way to set the grid's appearance is to click **Auto Format** at the bottom of the Properties window and select a predefined look.

Filling the Dataset and Displaying Data in the DataGrid Control

Although the grid is bound to the dataset you created, the dataset itself is not automatically filled in. Instead, you must fill the dataset yourself by calling a data-adapter method. For details about filling datasets, see Introduction to Datasets.

Even after the dataset is filled in, the **DataGrid** control does not automatically display the data. You must explicitly bind the grid to its data source. For more information, see Introduction to Data Binding in Web Forms Pages.

To fill the dataset and display data in the DataGrid control

1. Double-click the page to display the page's class file in the Code Editor.
2. In the **Page_Load** event handler, call the data adapter's **Fill** method, passing it the dataset you want to populate:

```
' Visual Basic
SqlDataAdapter1.Fill(DsCategories1)

//C#
sqlDataAdapter1.Fill(dsCategories1);
```

3. Call the **DataGrid** control's **DataBind** method to bind the control to the dataset. You do not want to rebind the control each time the page makes a round trip, however, because if you do, you will lose changes that the user has made in the grid. You should therefore bind the grid only at these times:

   - The first time the page is called.
   - When the dataset has changed.

   For now, you want to bind the grid the first time the page is called, which you can do by testing the page's **IsPostBack** property. Add code to the **Page_Load** event handler after the call to the adapter's **Fill** method. The complete handler will look like the following:

```
' Visual Basic
Private Sub Page_Load(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles
MyBase.Load
   'Put user code to initialize the page here
   SqlDataAdapter1.Fill(DsCategories1)
   If Not IsPostBack Then
      DataGrid1.DataBind()
   End If
End Sub

// C#
private void Page_Load(object sender, System.EventArgs e)
{
   // Put user code to initialize the page here
   sqlDataAdapter1.Fill(dsCategories1);
   if (!IsPostBack)
   {
      DataGrid1.DataBind();
   }
}
```

Adding the Ability to Edit Data

As configured, the **DataGrid** control will display information from the Categories table. But you also want users to be able to edit individual rows in the grid. To do this, you add an Edit button to each row of the grid. The user clicks the button and the grid is

redisplayed in edit mode, with text boxes where users can edit the individual columns. When the row is in edit mode, the Edit button is replaced with two other buttons, an Update button and a Cancel button, as shown in the following figure:

	CategoryID	CategoryName	Description
Update Cancel	1	Beverages	Soft drinks, coffees, teas,
Edit	2	Condiments	Sweet and savory sauces, relishes, spreads, and seasonings
Edit	3	Confections	Desserts, candies, and sweet breads
Edit	4	Dairy Products	Cheeses
Edit	5	Grains/Cereals	Breads, crackers, pasta, and cereal
Edit	6	Meat/Poultry	Prepared meats
Edit	7	Produce	Dried fruit and bean curd
Edit	8	Seafood	Seaweed and fish

You can set properties of the **DataGrid** control to display these buttons. However, the buttons are not hooked up automatically; instead, when an Edit or Cancel or Update button is clicked, it raises an event that you respond to. You will need to add short event handlers that do the following:

- Edit button – set the current row into edit mode.
- Cancel button – return the row to display mode.

The event handler for the Update button is more involved, since it performs the update that is the heart of this walkthrough. You will read about this in the next section.

To add editing capability to the DataGrid control

1. In Design view of the Web Forms Designer, select the **DataGrid** control, press F4 to display the Properties window, and then click **Property Builder** at the bottom of the window.

   The **DataGrid1 Properties** dialog box is displayed.

2. Click the **Columns** tab.
3. Under **Column list**, scroll down in the **Available columns** list and open the **Button Column** node.
4. Select **Edit, Update, Cancel** and click the add ( )button to add the buttons to the **Selected columns** box.
5. Click **OK**.

   The **DataGrid** control is redisplayed with an Edit link button in the left-hand column. (You do not initially see the Update and Cancel buttons.)

Now that you have the Edit button, you need to create the event handlers to set edit mode for the rows. To control edit mode, you set the **DataGrid** control's **EditItemIndex** property to the index (zero-based) of the row to edit. For example, to put the third row into edit mode, you set the property to 2. To return a row to display mode, you set the property to –1. After changing edit mode, you must rebind the grid so it displays the data in the row.

You can determine what row the user is in from the event-object passed to the handler. The event object for these events contains an **Item** property that represents the entire **DataGrid** row being updated. The **Item** object in turn supports various properties, including the **Item.ItemIndex** property, which returns the index value of the row you are working with.

To set edit modes

1. (In Visual Basic) Right-click the page and choose **View Code** to open the page's class file in the Code Editor.

   -or-

   (In Visual C#) In Design view, select the grid and then press F4 to open the Properties window.

2. (In Visual Basic) In the left-hand drop-down list at the top of the Code Editor, select **DataGrid1**.

   -or-

   (In Visual C#) Click the Events button ( ) at the top of the Properties window.

3. (In Visual Basic) In the right-hand drop-down list at the top of the Code Editor, select **EditCommand**.

   -or-

   (In Visual C#) Double-click **EditCommand** in the Properties window.

A **DataGrid1_EditCommand** handler is created.

4. Repeat Steps 2 and 3 for the **CancelCommand** event.

   A **DataGrid1_CancelCommand** handler is created.

5. In the **EditCommand** event handler, add the following code:

```
' Visual Basic
DataGrid1.EditItemIndex = e.Item.ItemIndex
DataGrid1.DataBind()

// C#
DataGrid1.EditItemIndex = e.Item.ItemIndex;
DataGrid1.DataBind();
```

6. In the **CancelCommand** event handler, add the following code:

```
' Visual Basic
DataGrid1.EditItemIndex = -1
DataGrid1.DataBind()

// C#
DataGrid1.EditItemIndex = -1;
DataGrid1.DataBind();
```

Testing What You Have So Far

You now have a **DataGrid** control that displays data from a dataset (after the dataset is filled from the database) and you have configured the grid with an Edit button for each row. Even though the page is not yet complete, it is useful to test it now to make sure that the edit capability is working correctly.

To test what you have so far

1. In Solution Explorer, right-click your Web Forms page and choose **View in Browser**.

   The project is compiled and the Web Forms page appears in a browser pane in the designer. If everything is working correctly, the grid will be filled with data and the first column in the grid will contain links that read "Edit."

2. Click the **Edit** link for any row in the grid.

   The row is redisplayed with these changes:

   - The Edit link is replaced with Update and Cancel links.
   - The data is redisplayed in **TextBox** controls.

3. Click **Cancel**.

   The row is redisplayed in its original form.

The next section details how you can have the Update link write changes from the grid to the dataset and database.

Updating the Dataset and Database

Up to this point, the **DataGrid** control has done most of the work involved in simply displaying data. However, as with other ASP.NET server controls, the grid does not include automatic provisions for updates — that is, for taking changes made by a user in the grid and sending them back to the data source. To perform updates, you will need to write some code.

The update will actually occur in two stages. First, you must update the dataset with changes made in the grid. Then you must write changes from the dataset back to the database. For more information, see Introduction to Dataset Updates.

When a grid row is in edit mode, the leftmost column contains an Update link. When users click the link, it raises an **UpdateCommand** event. You will write all your update code in the handler for this event.

To create the UpdateCommand handler

1. (In Visual Basic) If it is not already open, open the Code Editor again.

-or-

(In Visual C#) In Design view, select the grid and then press F4 to open the Properties window.

2. (In Visual Basic) In the left-hand drop-down list at the top of the Code Editor, select DataGrid1.

   -or-

   (In Visual C#) Click the Events button ( ) at the top of the Properties window.

3. (In Visual Basic) In the right-hand drop-down list at the top of the Code Editor, select **UpdateCommand**.

   -or-

   (In Visual C#) Double-click **UpdateCommand** in the grid.

   A **DataGrid1_UpdateCommand** event handler is created.

Steps for Updating From the DataGrid Control

The outline for what you will be doing is this:

1. Determine what row (by index) in the DataGrid control was updated. Then get the datakey out of that grid row so that you identify what row (by ID) is being updated.
2. Get the changed values out of the grid row that the user updated.
3. Use the datakey value to find the corresponding row in the dataset table and then write the changes to that row. At this point, you have updated the dataset, but not yet the database itself.
4. Send changes from the dataset to the database. This executes a SQL command or stored procedure that copies changes from the dataset to the database.
5. Refresh the contents of the **DataGrid** control.
6. The following sections explain the details of each of these steps. If you prefer, you can skip these explanations and go directly to the code that follows.

To update from the DataGrid control

1. Identify what **DataGrid** row was updated by getting the **ItemIndex** property of the row (**Item** object) passed in the event object. Then use the index value to get the corresponding value out of the grid's **DataKeys** collection:

```
' Visual Basic
Dim key As String = DataGrid1.DataKeys(e.Item.ItemIndex).ToString()

// C#
string key = DataGrid1.DataKeys[e.Item.ItemIndex].ToString();
```

2. Get the changed values out of the **DataGrid** row. To do so:
   a. From the item passed in the event object, get the appropriate cell (zero-based) out of the **Cells** collection from the item passed in the event object. For example, the left-most column in the grid is **Cells(0)**.
   b. For each cell, get its **Controls** collection, which contains all the elements that appear in the cell.
   c. Get the first (and only) control out of the collection — in this case, a **TextBox** control. To get the **TextBox**, declare a local variable of type **TextBox** and cast the object in the Controls collection to it.
   d. Get the **TextBox** control's value (its **Text** property).

   The following example shows how to perform these steps.

```
' Visual Basic
Dim categoryName, categoryDescription As String
Dim tb As TextBox
tb = CType(e.Item.Cells(2).Controls(0), TextBox)
categoryName = tb.Text
tb = CType(e.Item.Cells(3).Controls(0), TextBox)
categoryDescription = tb.Text
```

```
// C#
string categoryName;
string categoryDescription;
TextBox tb;
tb = (TextBox) e.Item.Cells[2].Controls[0];
categoryName = tb.Text;
tb = (TextBox) e.Item.Cells[3].Controls[0];
categoryDescription = tb.Text
```

3. Find the corresponding row in the data table. The typed **dsCategories** dataset contains a special **FindBy** method — in this case, the **FindByCategoryID** method — that locates a row by its primary key and returns a reference to it. Create a variable for the typed data row and call the method:

```
' Visual Basic
Dim r As dsCategories.CategoriesRow
r = DsCategories1.Categories.FindByCategoryID(key)

// C#
dsCategories.CategoriesRow r;
r = dsCategories1.Categories.FindByCategoryID(int.Parse(key));
```

4. Update the row by changing values in the row you located in Step 3, as in the following example:

```
' Visual Basic
r.CategoryName = categoryName
r.Description = categoryDescription

// C#
r.CategoryName = categoryName;
r.Description = categoryDescription;
```

5. Send changes from the dataset to the database by calling the data adapter's **Update** method:

```
' Visual Basic
SqlDataAdapter1.Update(DsCategories1)

DataGrid1.DataBind()

// C#
sqlDataAdapter1.Update(dsCategories1);
DataGrid1.DataBind();
```

6. Switch the current row in the grid out of editing mode.

```
' Visual Basic
DataGrid1.EditItemIndex = -1

// C#
DataGrid1.EditItemIndex = -1;
```

7. Data-bind the **DataGrid** control:

```
' Visual Basic
DataGrid1.DataBind()

// C#
DataGrid1.DataBind();
```

The following code shows what the completed **UpdateCommand** event handler looks like. Copy this code and paste it into the

class file for the Web Forms page.

Tip Be sure to overwrite the skeleton event handler you created earlier, otherwise you will have two methods with identical names and signatures.

```
' Visual Basic
Private Sub DataGrid1_UpdateCommand(ByVal source As Object, ByVal e As System.Web.UI.WebContr
ols.DataGridCommandEventArgs) Handles DataGrid1.UpdateCommand
    Dim categoryName, categoryDescription As String

    ' Gets the value of the key field of the row being updated
    Dim key As String = DataGrid1.DataKeys(e.Item.ItemIndex).ToString()

    ' Gets get the value of the controls (textboxes) that the user
    ' updated. The DataGrid columns are exposed as the Cells collection.
    ' Each cell has a collection of controls. In this case, there is only one
    ' control in each cell -- a TextBox control. To get its value,
    ' you copy the TextBox to a local instance (which requires casting)
    ' and extract its Text property.
    '
    ' The first column -- Cells(0) -- contains the Update and Cancel buttons.
    Dim tb As TextBox

    ' Gets the value the TextBox control in the third column
    tb = CType(e.Item.Cells(2).Controls(0), TextBox)
    categoryName = tb.Text

    ' Gets the value the TextBox control in the fourth column
    tb = CType(e.Item.Cells(3).Controls(0), TextBox)
    categoryDescription = tb.Text

    ' Finds the row in the dataset table that matches the
    ' one the user updated in the grid. This example uses a
    ' special Find method defined for the typed dataset, which
    ' returns a reference to the row.
    Dim r As dsCategories.CategoriesRow
    r = DsCategories1.Categories.FindByCategoryID(key)

    ' Updates the dataset table.
    r.CategoryName = categoryName
    r.Description = categoryDescription

    ' Calls a SQL statement to update the database from the dataset
    SqlDataAdapter1.Update(DsCategories1)

    ' Takes the DataGrid row out of editing mode
    DataGrid1.EditItemIndex = -1

    ' Refreshes the grid
    DataGrid1.DataBind()
End Sub

// C#
private void DataGrid1_UpdateCommand(object source, System.Web.UI.WebControls.DataGridCommand
EventArgs e)
{
   string categoryName, categoryDescription;

   // Gets the value of the key field of the row being updated
   string key = DataGrid1.DataKeys[e.Item.ItemIndex].ToString();

   // Gets get the value of the controls (textboxes) that the user
   // updated. The DataGrid columns are exposed as the Cells collection.
   // Each cell has a collection of controls. In this case, there is only    one
   // control in each cell -- a TextBox control. To get its value,
   // you copy the TextBox to a local instance (which requires casting)
   // and extract its Text property.
   //
```

```
    // The first column -- Cells(0) -- contains the Update and Cancel buttons.
    TextBox tb;

    // Gets the value the TextBox control in the third column
    tb = (TextBox)(e.Item.Cells[2].Controls[0]);
    categoryName = tb.Text;

    // Gets the value the TextBox control in the fourth column
    tb = (TextBox)(e.Item.Cells[3].Controls[0]);
    categoryDescription = tb.Text;

    // Finds the row in the dataset table that matches the
    // one the user updated in the grid. This example uses a
    // special Find method defined for the typed dataset, which
    // returns a reference to the row.
    dsCategories.CategoriesRow r;
    r = dsCategories1.Categories.FindByCategoryID(int.Parse(key));

    // Updates the dataset table.
    r.CategoryName = categoryName;
    r.Description = categoryDescription;

    // Calls a SQL statement to update the database from the dataset
    sqlDataAdapter1.Update(dsCategories1);

    // Takes the DataGrid row out of editing mode
    DataGrid1.EditItemIndex = -1;

    // Refreshes the grid
    DataGrid1.DataBind();
}
```

Testing

You are now finished. To illustrate how the grid works and make sure it is updating data properly, you should test the page.

To test the page

1. In Solution Explorer, right-click your Web Forms page and choose **View in Browser**.

   The project is compiled and the Web Forms page appears in a browser pane in the designer.

2. Click the **Edit** link for any row in the grid, and edit the row using the text boxes.
3. Click **Update**.

   The grid is redisplayed with your changes. If you examine the database, you will see that the change has been written there are well.

Next Steps

In this walkthrough, you used a **DataGrid** control on a Web Forms page not just to view data, but also to edit it. For the sake of clarity, the walkthrough skipped some details that you would likely want to think about in a production application. Some of the areas in which you might want to enhance what you have learned in this walkthrough are:

Enhancing the DataGrid Control

This walkthrough did not take advantage of some of the DataGrid control's more advanced features. Some of the enhancements you could make are:

- Configure individual columns in the grid by choosing specific columns to display, changing column headers and so on. For details, see Adding Bound Columns to a DataGrid Web Server Control and Specifying Grid Item Format in a DataGrid Web Server Control
- Allow users to delete rows. For details, see Allowing Users to Delete Items in a DataGrid Web Server Control. In addition to specifying a Delete button, you would need to add code to remove records from the dataset. For details, see Deleting Records in a Dataset.
- Use paging. In this instance, you only displayed a few records. However, it is common to display many more, and if so, it is

desirable to allow the grid to display information a page at a time. For details, see Paging Behavior in DataGrid Web Server Controls.

Caching the Dataset

In the walkthrough, the dataset is reinstantiated and refilled from the database each time the page makes a round trip to the server. This strategy, while straightforward to implement, can result in unnecessary traffic to your database server.

An alternative is to store the dataset after it is filled and each time it is changed. Then, instead of rereading the database information, you can restore the dataset with each round trip. There are a variety of options for storing the dataset, including in session state (on the server) or in view state (on the client, in the page). For details about what strategy to use, see State Management Recommendations.

For example, if you want to store the dataset in session state, you might include code such as the following in the **Page_Load** event handler:

```
' Visual Basic
If Session("mydatset") Is Nothing Then
   SqlDataAdapter1.Fill(DsCategories1)
   Session("mydataset") = DsCategories1
Else
   DsCategories1 = CType(Session("mydatset"), dsCategories)
End If
If Not Page.IsPostBack Then
   DataGrid1.DataBind()
End If

// C#
if (Session["mydatset"] == null)
{
   sqlDataAdapter1.Fill(dsCategories1);
   Session["mydataset"] = dsCategories1;
}
else
{
   dsCategories1 = (dsCategories)Session["mydatset"];
}
if (! Page.IsPostBack)
   {
        DataGrid1.DataBind();
   }
```

Note that if you store a dataset in Session or Application state, you need to cast it to the appropriate dataset type when you get it back out.

Be sure to refresh the copy of the dataset in session state each time it changes. For example, you might add the following line in the **UpdateCommand** event handler immediately after you update the database:

```
' Visual Basic
SqlDataAdapter1.Update(DsCategories1)
Session("mydataset") = DsCategories1

// C#
sqlDataAdapter1.Update(dsCategories1);
Session["mydataset"] = dsCategories1;
```

Using Direct Access to the Database instead of a Dataset

In the walkthrough, you created a dataset and bound the grid to it. To update the database, you first updated the dataset, then used the data adapter's **Update** method to send changes to the database.

Using a dataset makes it easy to perform updates, but at the cost of some memory and a very slight performance overhead. As an alternative, you can use data commands (OleDbCommand Class or SqlCommand Class objects) to execute SQL statements or stored procedures directly. For example, you can configure a data command with a parameterized SQL Update statement to update the Categories table in the database. After getting the changes from the grid, you set parameters for the Update statement

and then execute it.

For more information, see Introduction to DataCommand Objects in Visual Studio.

Validating Data

In a production application, you would probably include validation checks to be sure that the changes entered by a user were valid. For details, see Data Validation in Datasets.

Concurrency Control

In the walkthrough, you sent updates to the database without worrying about whether the record you were updating had been changed by another user. However, in production applications, you will usually want to include error checking to be sure that you maintain the integrity of your database. For details, see Concurrency Control in ADO.NET.

Adding Records Using the DataGrid Control

Although the **DataGrid** control allows you to display and edit data, it does not directly include the ability to add new records to a dataset or to a database table. You can add this ability indirectly by using a strategy such as the following:

- Add a new, blank record to the data source.
- Rebind the grid; this will display the new, blank record.
- Put the new grid row into edit mode, and when the user saves changes, update the new record with the user's information, as you would any existing record.

The exact steps you follow depend on the type of data source you are using (for example, whether it is a dataset or the database itself). You might also encounter limitations imposed on new records by primary key or foreign key constraints.

An example of adding a new record using the **DataGrid** control is illustrated in Top Questions about the DataGrid Web Server Control.

Security

When you create Web Forms, you should understand the types of security issues that can arise. For details, see Overview of Web Application Security Threats.

See Also

Walkthrough: Displaying Data in a Web Forms Page | Walkthrough: Creating Read-Only Data Access in a Web Forms Page | Walkthrough: Creating a Distributed Application | Walkthrough: Updating Data using a Database Update Query in Web Forms | Walkthrough: Validating User Input in a Web Forms Page

Walkthrough: Creating Paged Data Access Using a Web Forms Page

A common scenario in Web applications is to display a list — for example, a list of search results or of products from a catalog. Unless the list is short, it is typical for the list to be displayed in pages, with a way provided for users to navigate between them. This walkthrough illustrates one way to create paged output using a **DataGrid** Web server control.

There are a number of ways to create paged output, including:

- Create and fill a dataset for each page. This strategy is simple, allowing you to take advantage of the built-in paging features of the **DataGrid** control. However, it is not very efficient, since it gets the entire table from the database each time the user moves to another page. For details, see Specifying Paging Behavior in a DataGrid Web Server Control.
- Create a dataset and cache it. You can create a dataset and then store it (for example, in Session state). This saves the overhead of getting the table from the database each time, but still involves the overhead of storing the entire table in server memory. For small tables, this can be an efficient way to work.
- Fetch a page a worth of data from the database as needed. This method is efficient in two ways: it transfers only the minimum amount of data required each time, and it does not require server memory to store a dataset. However, it is the most complicated to create.

This walkthrough will illustrate the last method. When you are through, you will have a page that looks like the following:

CustomerID	CompanyName	City
ALFKI	Alfreds Futterkiste	Berlin
ANATR	Ana Trujillo Emparedados y helados	México D.F.
ANTON	Antonio Moreno Taquería	México D.F.
AROUT	Around the Horn	London
BERGS	Berglunds snabbköp	Luleå
BLAUS	Blauer See Delikatessen	Mannheim
BLONP	Blondesddsl père et fils	Strasbourg
BOLID	Bólido Comidas preparadas	Madrid
BONAP	Bon app'	Marseille
BOTTM	Bottom-Dollar Markets	Tsawassen

In order to complete this walkthrough, you will need:

- Access to a server with the Northwind SQL Server sample database. The SQL Server can be located on the same computer as the Web server or on another server. In either case, you must configure access permissions. For details, see Accessing SQL Server from a Web Application.
- Sufficient permissions to be able to create ASP.NET Web application projects on the computer where your Web server is. For details, see Web Application Security at Design Time in Visual Studio.

The walkthrough is split into a number of smaller pieces:

- Creating the Web application project and Web Forms page.
- Adding the data components needed to get data from the database.
- Adding the **DataGrid** control to the form.
- Adding code to fetch the right data when users move from page to page.

Because this walkthrough focuses on some more advanced concepts of Web Forms pages, it assumes that you are generally familiar with the following concepts:

- Data commands. For details, see Introduction to DataCommand Objects in Visual Studio.
- Setting command parameters. For details, see Setting and Getting Data Command Parameters.
- Using a data reader as a read-only, forward-only cursor. For details, see Executing a Data Command That Returns a Result Set.
- Saving page information in view state between round trips. For details, see Saving Web Forms Page Values using View State.

Creating the Project and Form

The first step is to create a Web application and a Web Forms page.

To create the project and form

1. On the **File** menu, point to **New**, then click **Project**.
2. In the **New Project** dialog box, do the following:
   a. In the **Project Types** pane, choose either **Visual Basic Projects** or **Visual C# Projects**.
   b. In the **Templates** pane, choose **ASP.NET Web Application**.
   c. In the **Location** box, enter the complete URL for your application, including http://, the name of the server, and the name of your project. The Web server must have IIS version 5 or later and the .NET Framework installed on it. If you have IIS on your computer, you can specify `http://localhost` for the server. (If you normally use a proxy server to access the Internet, you might need to configure Internet Explorer to bypass the proxy server in order to use localhost.)

   When you click **OK**, a new Web Forms project is created at the root of the Web server you specified. In addition, a new Web Forms page called WebForm1.aspx is displayed in the Web Forms Designer in Design view.

   Tip If you have trouble creating a Web application project, see Web Access Failed Dialog Box.

Adding the Data Components

In this walkthrough, you will need to execute two slightly different SQL statements to get a page of data. Both statements get 10 rows at a time based on a starting ID; they differ only in the WHERE clause.

- The statement for getting the next page of data gets 10 rows out of the table starting with the row after (that is, greater than) the last one on the current page.
- The statement for getting the previous page gets 10 records starting with (that is, equal to) the first one on the previous page.

Note As an alternative to using two statements, you can create a single one and dynamically change the WHERE clause. In this walkthrough, you will use two separate statements to shorten the amount of code you have to write.

In this walkthrough you will be using the Customers table from the SQL Server Northwind database. You can use the techniques illustrated in the walkthrough for any table, subject to the following conditions:

- The table you are accessing has a unique ID.
- The table is ordered by the ID.
- The SQL dialect you are using supports the TOP keyword in a Select statement. The TOP keyword allows you to specify how many rows to return. In this case, of course, you will use the TOP keyword to specify the number of rows per page.

Creating and Configuring the Connection

To begin, add a connection that points to the database you want to read from.

To create the data connection

1. From the **Data** tab of the Toolbox, drag a **SqlConnection** object onto the page.

   Note Because you are using SQL Server, you use the **SqlConnection** object, which is optimized for working with SQL Server 7.0 or later. If you are using a different database, you use the **OleDbConnection** object, which provides ADO.NET access to any OLE DB-compatible data source.

2. Select the connection and in the Properties window, click the button in the **ConnectionString** property box.

   The list displays any existing connections and the **<New Connection...>** option.

3. Unless you already have a connection to the SQL Server Northwind database, choose **<New Connection...>**.

   The **Data Link Properties** dialog box is displayed.

4. Create a connection pointing to your SQL Server and the Northwind database, and then click **OK**.

   Note You need appropriate read/write permissions on the SQL Server you are using. If SQL Server is on the same computer as IIS, it is recommended that you specify Windows integrated security when creating the connection. Otherwise, you can specify a user name and password and save that information with the

connection, but doing so can compromise security. For more information, see Accessing SQL Server from a Web Application.

The connection information you specified is saved as the connection string for your connection.

Now you need to add commands. You will need two in all, one for each SQL statement.

To create the data commands for getting pages of data

1. From the **Data** tab of the Toolbox, drag two **SqlCommand** objects onto the page.
2. Name one of them **cmdNext** and the other **cmdPrevious**.
3. Select the **cmdNext** command and in the Properties window, click the button in the **Connection** property, and select the connection you made earlier. (You will need to open the **Existing** node.)
4. In the **CommandText** property, click the ellipsis button to open the **Query Builder** dialog box.
5. Use the **Query Builder** dialog box to construct the following SQL statement:

```
SELECT TOP 10 CustomerID, CompanyName, City
FROM Customers
WHERE (CustomerID > @customerid)
ORDER BY CustomerID
```

6. When you have finished, close the **Query Builder** dialog box.

   The SQL statement is filled into the **CommandText** property.

7. Repeat Steps 3 through 5 for the `cmdPrevious` ecommand object, creating this query:

```
SELECT TOP 10 CustomerID, CompanyName, City
FROM Customers
WHERE (CustomerID >= @customerid)
ORDER BY CustomerID
```

You have finished adding data components to the page.

Adding the DataGrid Control

To illustrate paging, you will need a **DataGrid** control. For navigation, you will create Previous and Next buttons.

To add and configure DataGrid control

1. From the **Web Forms** tab of the Toolbox, drag a **DataGrid** control onto the page.
2. At the bottom of the Properties window, choose the **AutoFormat** link and select a predefined format for the grid.
3. In the Properties window, click the **Property Builder** link.

   The **DataGrid1 Properties** dialog box appears.

4. Click the **Paging** tab and set the following:

Property	Setting
AllowPaging	Checked
Allow custom paging	Checked
Page size	10
Show navigation buttons	Unchecked

5. Click **OK** to close the dialog box.

> **Note** If you have worked with a data-bound **DataGrid** control before, you might expect to set the control's **DataSource** property. In this example, however, you set the data source in code. Similarly, you will not be setting any columns.

To add navigation buttons

1. From the **Web Forms** tab of the Toolbox, drag two **LinkButton** controls onto the page.
2. Name one of them **btnPrevious** and the other **btnNext**.
3. Set the text of the buttons appropriately. For example, the text of the Next button might be `Next >`.

Adding Code to Fetch and Display a Page of Data

You can now add code to perform paging. The basic logic is that as users navigate, you will execute a SQL statement (using the data command objects you created) and return a data reader (SqlDataReader object) with 10 rows. You bind the **DataGrid** control to the data reader, and the rows are automatically displayed in the grid.

Next Page and Previous Page

The only complication in your page is in implementing navigation, and the issue is this: how do you determine which 10 records to get from the database?

The logic for the Next page is to get 10 rows starting with the row immediately following the last row in the grid. You will therefore get the ID of the last row in the grid and use it as a parameter for the `cmdNext` command you created earlier.

The logic for moving to the previous page is more involved. You want to get 10 rows starting (inclusively) with the first ID of the page preceding the one currently in the grid. For this, you will need to have available the ID of the previous page, and if the user continues paging backward, of the preceding pages as well.

One solution is to create a structure to store the ID of the first record of each page as that page is displayed. That way, when you return to the page, you can get its starting ID and use that as a parameter for the query to fill the page.

Because this is a Web Forms page, you cannot store page information in a variable; you must store it somewhere where it is preserved between round trips. In this walkthrough, you store information in view state, which places it into the page itself (in a hidden field). You will need to store two types of items:

- The current page number. You will use this as an index value, and it will also tell you when the user has navigated back to the first page.
- For each page, the ID of the first row in the grid. You will create a new item in view state for each page. The name of the item is the page number and the value of it is the ID.

For example, if you are on the third page of the grid, you will have added four items to view state, something like these:

```
CurrentPage=3
0=ALFKI
1=BSBEV
2=FAMIA
```

The following sections show you the code that you will need to implement the paging logic.

Displaying Data in the Grid

Filling the grid is the same no matter which way the user has navigated. You can therefore create a generic method that executes either the `cmdNext` or `cmdPrevious` command, depending on which one is passed to the method.

To display data in the grid

1. Create a private variable called `CurrentPage`. You will set the value of this variable in other methods later.
2. Create a method (perhaps named FillGrid) that accepts a data command (SqlCommand) as a parameter.
3. Instantiate a data reader.
4. Open a connection and the data command's ExecuteReader method to execute its SQL statement and fetch rows.
5. Set the **DataGrid** control's **DataSource** property to the data reader and then bind the grid.
6. Close the data reader and the connection.
7. Save two values in view state:

   - The value of the `CurrentPage` variable.
   - The value of the ID column of the first row in the grid. Store the value in a view state item whose name is the page number. The logic looks like this:

     ```
     ' Visual Basic
     ViewState(CurrentPage.ToString()) = DataGrid1.Items(0).Cells(0).Text

     // C#
     ```

```
ViewState[CurrentPage.ToString()] = DataGrid1.Items[0].Cells[0].Text;
```

8. Finally, determine whether you are at the end of the table. For simplicity, in this walkthrough you will simply test whether the query returned fewer rows than the page size (10). If you have reached the end of the table, disable the Next button so that users do not try to continue.

   Note For another suggestion about how to determine whether you are at the end of the table, see "Next Steps" later in this walkthrough.

   The following example shows how you create the `CurrentPage` variable:

```
' Visual Basic
Public Class WebForm1
    Inherits System.Web.UI.Page
    Private CurrentPage As Integer
   ' more declarations here

//C#
public class WebForm1 : System.Web.UI.Page
{
   private int CurrentPage;
   // more declarations here
```

   The following example shows what the code looks like for the complete, generic method:

```
' Visual Basic
Private Sub FillGrid(ByVal currentSqlCommand as SqlClient.SqlCommand)
   Dim dr As SqlClient.SqlDataReader
   SqlConnection1.Open()
   dr = currentSqlCommand.ExecuteReader()
   DataGrid1.DataSource = dr
   DataGrid1.DataBind()
   dr.Close()
   SqlConnection1.Close()
   ViewState("CurrentPage") = CurrentPage
   ViewState(CurrentPage.ToString()) = DataGrid1.Items(0).Cells(0).Text
   ' Determine how many rows were filled into the grid. If it is less
   ' than the number of rows per page, there are no more rows in the
   ' table, and the Next button should be disabled.
   If DataGrid1.Items.Count < DataGrid1.PageSize Then
       btnNext.Enabled = False
   End If
End Sub

//C#
private void FillGrid(System.Data.SqlClient.SqlCommand currentSqlCommand)
{
   System.Data.SqlClient.SqlDataReader dr;
   sqlConnection1.Open();
   dr = currentSqlCommand.ExecuteReader();
   DataGrid1.DataSource = dr;
   DataGrid1.DataBind();
   dr.Close();
   sqlConnection1.Close();
   ViewState["CurrentPage"] = CurrentPage;
   ViewState[CurrentPage.ToString()] = DataGrid1.Items[0].Cells[0].Text;
   // Determine how many rows were filled into the grid. If it is less
   // than the number of rows per page, there are no more rows in the
```

```
    // table, and the Next button should be disabled.
    if (DataGrid1.Items.Count < DataGrid1.PageSize)
    {
       btnNext.Enabled = false;
    }
  }
```

Initializing the Page

The first time the page runs, you want to display the first page of data. Now that you have the generic method for filling the grid, all you need to do is set a parameter and call the method. You want to execute the `cmdNext` command. Set its `@customerid` parameter to an empty string, meaning that the query will bring back the first 10 records greater than "" — that is, the first 10 records in the table. You also want to set the page number (`CurrentPage`) to zero to indicate that you are on the first page.

The code looks like this:

```
' Visual Basic
Private Sub Page_Load(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles MyB
ase.Load
   'Put user code to initialize the page here
   If Not Page.IsPostBack Then
       cmdNext.Parameters("@customerid").Value = ""
       CurrentPage = 0
       FillGrid(cmdNext)
   End If
End Sub

//C#
private void Page_Load(object sender, System.EventArgs e)
{
   // Put user code to initialize the page here
   if (!Page.IsPostBack)
   {
      cmdNext.Parameters["@customerid"].Value = "";
      CurrentPage = 0;
      FillGrid(cmdNext);
   }
}
```

Navigating to the Next Page

When users navigate to the next page, you again want to execute the `cmdNext` command using your generic method. This time, the parameter for the command is the value of the ID column of the last row in the grid.

You also need to increment the page number so that the generic `FillGrid` method can use the number to create another view state item for the ID. To increment the page number, you have to get the old one out of view state. When you do so, you have to cast it to an integer, since all items in view state are stored as objects.

The code for the Next button looks like this:

```
' Visual Basic
Private Sub btnNext_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles
btnNext.Click
   ' Get the page number of the page most recently displayed
   CurrentPage = CType(ViewState("CurrentPage"), Integer)
   CurrentPage += 1
   'Gets the id on current page
   Dim lastid As String = DataGrid1.Items(9).Cells(0).Text
   cmdNext.Parameters("@customerid").Value = lastid
   FillGrid(cmdNext)
End Sub

//C#
private void btnNext_Click(object sender, System.EventArgs e)
{
```

```
    // Get the page number of the page most recently displayed
    CurrentPage = (int)(ViewState["CurrentPage"]);
    CurrentPage++;
    // Gets the id on current page
    string lastid = DataGrid1.Items[9].Cells[0].Text;
    cmdNext.Parameters["@customerid"].Value = lastid;
    FillGrid(cmdNext);
}
```

Navigating to the Previous Page

Finally, you need to add code to move backwards. This is where you will use the information you have been storing in view state. First, you get the current page number and then decrement it. Then you look in view state, using the page number as the name, to get the ID of the first row on the page. You use that ID as the parameter for the `cmdPrevious` command, which you pass to the generic `FillGrid` method.

To prevent running off the beginning of the table, you should check that you page count is not below zero. In addition, you should enable the Next button, just in case it was disabled in previous navigation.

```
' Visual Basic
Private Sub btnPrevious_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Han
dles btnPrevious.Click
   btnNext.Enabled = True
   CurrentPage = CType(ViewState("CurrentPage"), Integer)
   CurrentPage -= 1
   If CurrentPage >= 0 Then
       Dim firstid As String
       firstid = CType(ViewState(CurrentPage.ToString()), String)
       cmdPrevious.Parameters("@customerid").Value = firstid
       FillGrid(cmdPrevious)
   End If
End Sub

//C#
private void btnPrevious_Click(object sender, System.EventArgs e)
{
   btnNext.Enabled = true;
   CurrentPage = (int)(ViewState["CurrentPage"]);
   CurrentPage--;
   if (CurrentPage >= 0)
   {
      string firstid;
      firstid = (string)(ViewState[CurrentPage.ToString()]);
      cmdPrevious.Parameters["@customerid"].Value = firstid;
      FillGrid(cmdPrevious);
   }
}
```

Testing

After adding the code, test data access in the Web Forms page.

To test the Web Forms page

1. Save the page.
2. In Solution Explorer, right-click the page and choose **View in Browser**.

   Confirm that a list of 10 customers is displayed in the grid.

3. Click the Next and Previous buttons several times to be sure you are navigating correctly. Special tests to make are:

   - On the first page, click Previous; nothing should happen.
   - Click Next till you reach the end of the table and make sure that the Next button is disabled.

Next Steps

This walkthrough has illustrated one method of implementing paging. The scenario illustrated here was simple but complete. In real applications, many additional factors can come into play, and paging can become quite complex. Some issues to think about include:

- To determine when you have reached the end of the file, the walkthrough uses an algorithm that is easy but not perfectly reliable. (It compares the number of returned rows against the number of rows in a page). In a real application, you would want to add a more sophisticated algorithm. One possibility is to do the following:

  1. Add another command object to the page that includes this SQL statement:

     ```
     Select Count(CustomerID) From Customers
     ```

  2. The first time the page runs, execute the command using the command's **ExecuteScalar** method to get the total number of rows in the table. Then calculate the total page count required for the table and store it in both a global variable and in view state. On subsequent posts, get the value back out of view state.
  3. In the `FillGrid` method, check whether the current page has reached the total page count. If so, disable the Next button.

- The walkthrough assumes that the data is not volatile. If other users are adding rows to the table, page boundaries change, and storing the first ID of a given page is not a viable solution.
- You can create a stored procedure to perform the work done here with SQL statements.
- Security. For details, see Overview of Web Application Security Threats.

See Also

Setting and Getting Data Command Parameters | Executing a Data Command That Returns a Result Set

Web Forms Data Binding

As in most types of forms, Web Forms pages allow you to display information by binding controls to a source of data. However, due to the nature of Web Forms pages themselves and to the architecture of Web programming, data binding in Web Forms pages is somewhat different than it is in other, more traditional forms such as Windows Forms.

The topics in this section discuss how data binding works in Web Forms pages and how to specify data binding for various types of controls.

In This Section

Introduction to Data Binding in Web Forms Pages
Provides an overview of how controls in a Web Forms page are bound to data.
Multi-Record and Single-Value Data Binding for Web Forms Pages
Describes the differences between binding a multi-record control (such as a **DataList** control) and a control that displays only a single data value (such as a **TextBox** control).
Data-Binding Expressions for Web Forms Pages
Describes how Web Forms control data binding is specified through the use of a special syntax.
Data Binding Multi-Record Web Server Controls
Provides step-by-step instructions on how to bind a control such as a **DataList** or **DataGrid** control to a data source containing more than one record.
Data Binding Single-Value Web Server Controls at Design Time
Provides step-by-step instructions on how to bind a Web server control property (using a data-binding expression) to a single data value.
Data Binding HTML Server Controls
Provides step-by-step instructions on how to bind property values of HTML server controls (such as the **HtmlInputText** or **HtmlSelect** control) to a data source.
Data Binding Single-Value Web Server Controls at Run Time
Describes how to programmatically use event handlers to load data values into control properties.

Related Topics

Introduction to Data Access with ADO.NET
Provides an overview of how ADO.NET provides data access capability for your applications.
Data Access in Web Forms Pages
Lists topics that describe how data access works in Web Forms pages.
Data Form Wizard
Creates data-bound Windows Forms or Web Forms pages. The topics describes how to run the wizard, what options you can choose, and what the wizard generates.
Performing Database Operations Directly
Lists topics that describe how to use data commands to execute SQL statements or stored procedures against a database.
Dataset Updates in Visual Studio .NET
Lists topics that describe how ADO.NET provides capabilities for writing data from your application a data source.
Database Updates from Datasets
Lists topics that describe how to write data from an ADO.NET dataset to a data source.
Walkthrough: Updating Data Using a Database Update Query in Web Forms
Gives step-by-step details on updating data using a database query in a Web Forms page.
Walkthrough: Using a DataGrid Web Control to Read and Write Data
Illustrates step-by-step how to use a **DataGrid** control on a Web Forms page to read data and update it.

Introduction to Data Binding in Web Forms Pages

As in most types of forms, Web Forms pages allow you to display information by binding controls to a source of data. However, due to the nature of Web Forms pages themselves and to the architecture of Web programming, data binding in Web Forms pages is somewhat different than it is in other, more traditional forms such as Windows Forms.

This topic provides background information on data binding in Web Forms pages, how you work with data binding in your Web applications, and where you can find more information about data binding and data access in Web Forms pages.

Web Forms Data Binding: Read-Only

The data-binding architecture of Web Forms pages is based on the idea that the majority of data access in a Web application is read-only — that is, that pages frequently display information from a data store, but much less frequently need to write data back to the store. There are many examples of pages that present read-only data:

- A search results page for other Web pages, products, and so on. Information in a search results page is often presented as links to other pages, and rarely is there a need for users to enter data to be saved in a database.
- A page that provide sports scores, weather information, and so on.
- A portal page. Many portal pages (such as the MSN home page) allow users to customize the page by choosing from various types of information to display — news, financial information, entertainment options, and so on. The information itself is often provided to the page as XML data that can be transformed according to user preferences.
- A page that provides account and financial transaction information to a bank or brokerage customer.

Because the Web Forms data-binding architecture does not perform updates — that is, it does not write data from the control back to the data source — you must provide this logic. The logic for performing updates in Web Forms pages varies considerably according to a variety of factors:

- What the data source is.
- Whether you are caching data by using a dataset.
- What type of control you are working with — for example, whether the control displays multiple records, as in a grid, or single values, as in a text box.
- What type of concurrency control you want to use.

Because of the variables associated with updating from Web Forms pages, you implement your own update logic to match the specific requirements of your Web Forms page and application. For more information, see Dataset Updates in Visual Studio .NET.

Binding Properties

Data binding in Web Forms pages is flexible — you can bind any property of a control to data. In contrast, in traditional data binding, you typically bind the display property (for example, the text property of a text box) to the data source.

In Web Forms pages, your data source can be almost any type of information available to the page — whether it comes from a database, from an XML document, from another control or process, or even from a calculation you perform yourself.

Web Forms data binding, moreover, allows you to assign this data to any property of any control. Because of this, you can use data binding to perform actions such as:

- Setting the text to display in a **TextBox**, **Label**, **Button**, **LinkButton**, or other control.
- Setting the target page (the HREF attribute) of a link control.
- Binding a **CheckBox** control to a Boolean value to directly set the checked state of the control.
- Setting the graphic of an **Image** control by binding its **ImageUrl** property to a database column containing the URL or name of a graphics file.
- Setting the color, font, or size of controls. This can be useful as a way of implementing user preferences that you store in a database or other data store.

In other words, data binding becomes an automatic way of setting any run-time accessible property of any control on the page.

Binding Multi-Record and Single-Value Controls

There are two types of data binding that a control might support:

- Multi-record controls can display one or more data records at once, such as the **Repeater**, **DataList**, **DataGrid** Web server controls and the **HtmlSelect** HTML server control.

- Single-value controls can display one data value at a time. This includes most Web server and HTML server controls, including the **TextBox**, **Label**, **HtmlInputText**, and **HtmlAnchor** controls.

> **Note** All controls allow you to bind individual properties to single data values. For example, a **DataList** control can display multiple records at once, but you might bind its **BackColor** property to a single data value.

Data binding works differently depending on whether you are binding to multiple records or to a single data value. To display multiple records, you bind a multi-record control to a data source that contains multiple records — an array, a collection, a data table, and so on. The only requirement is that the data source implement the IEnumerable interface. The default behavior is that when data binding occurs, the multi-record control loops through every data item in the data source.

To bind to a single value, on the other hand, you specify what property to bind and what data value it should get its value from.

For more information about how data binding works in each of these types of controls, see Multi-Record and Single-Value Data Binding for Web Forms Pages.

Binding at Design Time vs. at Run Time

If you are binding controls to data in a dataset, you can specify data-binding properties for controls at design time. This includes binding to dataset tables and to data views. For example, you can use design-time tools to specify that a **DataGrid** control be bound to a table in a dataset. For more information, see Data Binding Multi-Record Web Server Controls.

Similarly, you can bind a **TextBox** control to a value in a row returned by a data view. Visual Studio includes tools to help you easily specify this kind of design-time binding. For details, see Data Binding Single-Value Web Server Controls at Design Time.

In some situations, however, it is impractical to specify binding properties at design time. Generally, if the data source you want to bind to does not have an instance on your Web Forms page at design time, you need to perform data binding at run time. Some examples where you might do this include:

- Binding controls to data returned by executing a SQL statement or a stored procedure in a database
- Binding to objects of any sort that are not instantiated until run time.

At run time, controls raise a DataBinding event that you can use to fetch data values and assign them to control properties.

See Also

Multi-Record and Single-Value Data Binding for Web Forms Pages | Data-Binding Expressions for Web Forms Pages | Data Binding HTML Server Controls | Data Form Wizard

Multi-Record and Single-Value Data Binding for Web Forms Pages

Some Web Forms controls display values from multiple records at once. These controls include the **Repeater**, **DataList**, **DataGrid**, **ListBox**, **CheckBoxList**, and **RadioButtonList** Web server controls, as well as the **HtmlSelect** control. Other controls, such as **Label**, **TextBox**, **CheckBox**, and **Hyperlink** controls, display single values, which in data-binding terms means values from a single record.

Data Binding in Multi-Record Controls

To bind multi-record controls, including the **Repeater**, **DataList**, **DataGrid**, **ListBox**, **RadioButtonList**, **CheckBoxList**, and **HtmlSelect** controls, you set properties that specify where the records are that you want to display. You generally set properties indicating:

- The data source, or the container, containing the records you want to display. A typical data source is a dataset.
- The data member, such as the name of a specific table, if there is more than one bindable member.
- A specific field or column to display. This applies to controls such as the **ListBox** and **HtmlSelect** controls that display multiple data items but only one value from each item.

> **Note** You can bind individual properties of multi-record controls to single data values. For example, a **DataList** control can display multiple records at once, but you might bind its **BackColor** property to a single data value.

Although you often bind to data in a dataset, you can bind a multi-record control to almost any structure that supports the IEnumerable interface. This includes the following:

- ArrayList objects
- Collections
- Data readers (objects of type OleDbDataReader and SqlDataReader)
- DataView objects
- DataRow objects
- DataTable objects in datasets

During data binding, the multi-record control loops through the data source, extracting each item in turn. What happens with each data item in turn depends on what type of multi-record control you are working with. A **DataGrid** control can automatically display all items or a subset of the information based on properties you specify. The **Repeater** and **DataList** controls, in contrast, act as containers for controls embedded in their templates (for information see Web Server Controls Templates), which perform their own data binding.

Design-time and Run-time Data Binding

If there is an instance of the data source on your Web Forms page at design time, you can specify binding at design time also. For example, imagine that you want to bind a **DataGrid** control to a table in a dataset. If there is an instance of the dataset on your Web Forms page, Visual Studio recognizes it as a valid data source and allows you to set the grid's properties to bind to the table. You can bind this way to dataset tables, data views, and other objects on the page.

In many cases, however, the object you want to bind to does not exist until run time. This is true of structures such as an **ArrayList** or similar object, which you typically instantiate and fill at run time. It is also true for some sources that contain data from a database. For instance, you can get data for the page by using a data command to execute a SQL statement, and then fetch the individual data rows using a data reader. Because you cannot put an instance of the data reader directly onto your page, you must bind to it programmatically at run time.

For more information, see Data Binding Multi-Record Web Server Controls.

Binding to Single Data Values

You can bind individual control properties to virtually any information that can be resolved into a single value appropriate for the property you are binding. A very common scenario, for example, is to bind the **Text** property of a **TextBox** control to a data value.

To perform this type of binding, you do not use explicit data binding properties. Instead, you get the data value you want and assign it as a property value of the control.

> **Note** Controls that do not display multiple records — that is, controls other than **DataList**, **DataGrid**, and so on —

do not include explicit data-binding properties.

Design-time Data Binding

At design time, you can bind a control property to a data value using a "data-binding expression." The expression can reference a row in a dataset table or in a data view, or it can reference a property of another control, or almost anything else. You set a property value to this data-binding expression. When the page runs, the data-binding expression is resolved and the resulting value is assigned to the property. For more information, see Data-Binding Expressions for Web Forms Pages.

If you are using a data-binding expression to bind to data in a dataset, creating a reference to one particular record can be impractical. The dataset does not maintain a pointer or support the concept of a "current record," so there is no implicit way to indicate what record to bind to.

> **Note** This is different from Windows Forms. In Windows Forms, the data-binding architecture built into the form maintains its own set of pointers to records in a dataset, so that controls can be bound to a kind of "current record."

There are several ways to pinpoint the record you want to work with when binding to single data values:

- Create a data view that filters a dataset table to return only one record (for example, by filtering on a unique key). You can then bind the controls to the data view. The data-binding expression for this scenario might look something like the following:

  ```
  DataBinder.Eval(DataView1, "[0].au_lname")
  ```

  The index value zero (`[0]`) indicates that the expression will get data from the first w(and presumably, only) record represented by the data view. For details, see Filtering and Sorting Data Using Data Views.

- Fill a dataset with just one record. However, this might not be an efficient way to use datasets.

Alternatively, you can data bind the control at run time. Doing so allows you to determine programmatically what data row to bind to, which can include executing a SQL statement or stored procedure and binding to the results. For details, see Data Binding Single-Value Web Server Controls at Run Time.

Binding Single-value Controls in Templates

Multi-record controls such as the **DataList**, **Repeater**, and the **DataGrid** controls include templates used to render individual records from the data source. The templates in turn often contain single-value controls such as **TextBox** and **Label** controls that are bound to data.

> **Note** For details about templates, see Web Server Controls Templates.

Because the single-value controls in the template can bind to only one value, you need to specify which record to bind to. The parent control makes this possible by exposing two objects that you can use in a data-binding expression:

- The **Container** object refers to the parent control, which in turn is bound to a data source.
- The **DataItem** object references the record currently being processed by the parent control.

A data-binding expression for a **TextBox** control in a **DataList** control template might look like the following:

```
DataBinder.Eval(Container, "DataItem.au_lname")
```

For more information, see Data Binding Single-Value Web Server Controls at Design Time.

See Also

Introduction to Data Binding in Web Forms Pages | Data-Binding Expressions for Web Forms Pages | Web Server Controls Templates | Data Binding Multi-Record Web Server Controls | Data Binding Single-Value Web Server Controls at Design Time | Data Binding Single-Value Web Server Controls at Run Time | Data Binding HTML Server Controls

Data-Binding Expressions for Web Forms Pages

Data binding individual control properties in Web Forms pages is not implemented by directly tying the property to a source of data. Instead, data binding is implemented by using a special expression format. Information about what data you want to bind to is built into the expression; you then assign the results of the expression to a control property.

For example, imagine that you want to bind a **TextBox** Web server control to some data. You create a data-binding expression and assign it to the control's **Text** property so that the value will be displayed in the control.

The following example shows what the control declaration might look like in HTML view. The control's **Text** property is bound to a data view that contains a single record. The data-binding expression is delimited with the characters `<%#` and `%>`.

```
<asp:TextBox id="TextBox1" runat="server"
   Text='<%# DataView1(0)("au_lname") %>'>
</asp:TextBox>
```

Similarly, you can use a data-binding expression to set the ImageUrl property of an **Image** Web server control. In that case, you are extracting a string from the database that contains the path and filename of the graphic to display. An example might look like the following:

```
<asp:Image id=Image1 runat="server"
   ImageUrl='<%# DataView(0)("productPhotoURL") %>'>
```

In Visual Studio, the Properties window provides you with a tool to create data-binding expressions. Alternatively, you can create them yourself and enter them in HTML view of the Web Forms designer.

Advantages of Using Data-binding Expressions

Using an expression for data binding gives you flexibility in a number of ways:

- You can use any expression, as long as it resolves to a value or values the control can use. Most frequently, the data-binding expression will resolve to a value derived from a data source, but it can also reference properties of the page or of other controls, values that you calculate at run time, or almost anything else.
- You can assign the expression to any property — that is, you can bind any property to data. For example, you can keep information about user preferences in a database and use data binding to implement those preferences in properties for fonts, colors, sizes, styles, and so on. In addition, you can bind more than one control property, which allows you to bind one property to one data source, and another property to a different source.

Using the DataBinder Class for Binding

Although you can data bind using virtually any expression that resolves to a value, in most cases you will bind to a data source of some sort. This is most often a table in a dataset or a data view that contains the single record you are interested in. To simplify this type of data binding, ASP.NET server controls support a class called DataBinder that performs some of the work of extracting data and making it available to a control property.

You use the **DataBinder** class by invoking its **Eval** method, which requires two parameters:

- A reference to the data container, typically the dataset, data table, or data view.
- A reference to the individual value to be derived. This frequently references a single row (row zero) and a column value in that row.

The following example shows the same data binding that was done above for a text box, but this time using the **DataBinder** class.

```
<asp:TextBox id="TextBox1" runat="server"
   Text='<%# DataBinder.Eval(DataView1, "[0].au_lname") %>'>
</asp:TextBox>
```

The earlier example of setting the **ImageUrl** property of an **Image** control might look like the following. In this example, a formatting expression is passed in the optional second parameter of the **DataBinder.Eval** method; the expression adds a path onto the data as a prefix.

```
<asp:Image id=Image1 runat="server"
   ImageUrl='<%# DataBinder.Eval(Container, "DataItem.ProductImage", "http://myserver/myapps/
images/{0}") %>'>
```

The advantages of using the **DataBinder** class are:

- The syntax is consistent for all binding, enforced by the parameters required for the **Eval** method.
- The **DataBinder** class is supported by the Visual Studio design tools for Web Forms pages.
- The class performs type conversion automatically. For example, if you are binding a text box to a data column containing an integer, the **DataBinder** class will convert the integer to a string automatically.
- You can optionally specify a formatting expression that can convert or amend the data.

Resolving Data-binding Expressions

Data-binding expressions must be resolved at run time in order to provide values that controls can bind to. You explicitly perform this step during page processing by calling the DataBind method, which is a method of the System.Web.UI.Control class. You can call the method for individual controls or, more efficiently, you can call it for the **Page** class (which also derives from the **Control** class). The method cascades the call to all child controls, so by calling it once for the page, you invoke it for all controls on the page.

You typically call the **DataBind** method under these circumstances:

- The first time the page runs, but after the data source has been populated (for example, after you have filled a dataset).
- After the data source has changed (for example, because you have updated records in it).

The following example shows a typical way to call the **DataBind** method during the page initialization event:

```
' Visual Basic
Private Sub Page_Load(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles MyB
ase.Load
   SqlDataAdapter1.Fill(DsAuthors1, "authors")
   If Not (Me.IsPostBack) Then
     Me.DataBind()
   End If
End Sub

// C#
private void Page_Load(object sender, System.EventArgs e)
{
   SqlDataAdapter1.Fill(dsAuthors1, "authors");
   if (!this.IsPostBack)
   {
      this.DataBind();
   }
}
```

You generally do not want to call the **DataBind** method on each round trip (that is, in the page initialization without checking for a post back), because doing so replaces the values in controls. For example, if you are working with a **DataGrid** control, the control might contain changes that you want to process. When you call **DataBind**, the contents of the grid are replaced with values from the data source. If you do this during page initialization, you will lose the changes in the grid before you get a chance to process them. Instead, you usually call the **DataBind** method in an event handler, after you have performed the data processing for that event.

See Also

Data Binding Expression Syntax | Multi-Record and Single-Value Data Binding for Web Forms Pages |
Data Binding Single-Value Web Server Controls at Design Time | Data Binding HTML Server Controls

Data Binding Multi-Record Web Server Controls

Multi-record controls — such as the **Repeater**, **DataList**, **DataGrid**, **ListBox**, **CheckBoxList**, and **RadioButtonList** controls — are bound to a data source that contains multiple data items. For example, you can bind a **DataGrid** control to a table in a dataset and the control can directly display the data. Or you can bind a **DataList** or **Repeater** control to a table in a dataset and the control can make it available to child controls.

You can also bind an HtmlSelect control to multiple records. The procedure is different than it is for the Web server controls listed above. For details, see Data Binding HTML Server Controls.

> **Note** This procedure discusses setting properties that bind the control to an object containing multiple data items. You can also use data binding to set the value of single-value properties in these controls. For example, you can set the control's **Width** or **BackColor** property using single-value data binding. For details, see Data Binding Single-Value Web Server Controls at Design Time.

To bind a multi-record control

1. Set the control's DataSource property to the object containing the data items you want to bind to.

   The container object must implement the IEnumerable interface; this includes objects such as:

   - ArrayList objects
   - Collections
   - Data readers (objects of type OleDbDataReader and SqlDataReader)
   - DataView objects
   - DataRow objects
   - DataTable objects in datasets

2. If the data source is a dataset, set the DataMember property to the name of the table to bind to. (If this property is not set, by default the first table in the dataset is used.)
3. If you are working with a **DataList** or **DataGrid** control, set the **DataKeyField** property to the name of a column that you can use at run time to identify individual items. If it is available, you usually use the primary key column from the data source.
4. If you are working with a **ListBox**, **DropDownList**, **CheckBoxList**, **RadioButtonList**, or **HtmlSelect** control, set the DataTextField property to the name of the data column to display in the control.
5. Optionally, set the DataValueField property to specify the name of a column that the control should return as its value when the user selects an item.
6. Add code to the Web Forms page to call the control's or the page's DataBind method.

   > **Note** You generally do not want to call the **DataBind** method on each round trip (that is, in the page initialization without checking for a post back), because doing so replaces the values in controls.

   The following example shows how you can data bind a **ListBox** control. The example assumes that you have a dataset called `DsTitles1` that contains a table called `titles`. The table contains two columns, the title column that you assign to the **ListBox** display text and the `title_id` column that you assign as the control's value.

```
' Visual Basic
Private Sub Page_Load(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles
MyBase.Load
   SqlDataAdapter1.Fill(DsTitles1)
   If Not (Me.IsPostBack) Then
      ListBox1.DataSource = DsTitles1
      ListBox1.DataMember = "titles"
      ListBox1.DataTextField = "title"
      ListBox1.DataValueField = "title_id"
      ListBox1.DataBind()
   End If
End Sub

// C#
```

```
private void Page_Load(object sender, System.EventArgs e)
{
   sqlDataAdapter1.Fill(dsTitles1);
   if (! this.IsPostBack)
   {
      ListBox1.DataSource = dsTitles1;
      ListBox1.DataMember = "titles";
      ListBox1.DataTextField = "title";
      ListBox1.DataValueField = "title_id";
      ListBox1.DataBind();
   }
}
```

The following example shows how to bind a **ListBox** control to the data returned when you use a **DataCommand** object to execute a SQL statement. The command object's **ExecuteReader** method returns a data reader object (in this case, of type SqlDataReader), which you can directly bind a **ListBox** control to.

> **Note** For details about using data commands and data readers, see Executing a Data Command That Returns a Result Set.

```
' Visual Basic
Private Sub Page_Load(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles
MyBase.Load
   If Not (Me.IsPostBack) Then
      SqlConnection1.Open()
      SqlCommand1.CommandText = "Select au_id, au_lname from authors"
      ListBox1.DataSource = SqlCommand1.ExecuteReader()
      ListBox1.DataTextField = "au_lname"
      ListBox1.DataValueField = "au_id"
      ListBox1.DataBind()
      SqlConnection1.Close()
   End If
End Sub

// C#
private void Page_Load(object sender, System.EventArgs e)
{
   if (! this.IsPostBack)
   {
      sqlConnection1.Open();
      sqlCommand1.CommandText = "Select au_id, au_lname from authors";
      ListBox1.DataSource = sqlCommand1.ExecuteReader();
      ListBox1.DataTextField = "au_lname";
      ListBox1.DataValueField = "au_id";
      ListBox1.DataBind();
      sqlConnection1.Close();
   }
}
```

See Also

Introduction to Data Access in Web Forms Pages | Multi-Record and Single-Value Data Binding for Web Forms Pages | Data Binding Single-Value Web Server Controls at Design Time

Data Binding Single-Value Web Server Controls at Design Time

You can bind any single-value Web server control property — for example, the **Text** property of a **TextBox** or the **ImageUrl** property of an **Image** control — to a data value. This includes not only display properties, but also behavior properties such as size, width, and font.

> **Note** If you are working with list controls (**Repeater**, **DataList**, **DataGrid**, **CheckBoxList**, and **RadioButtonList**), you specify data binding for the control as a whole using special data-binding properties. For details, see Data Binding Multi-Record Web Server Controls. However, you can bind their single-value properties such as size and font using the procedure in this topic.

You specify single-value data binding by creating a data-binding expression. In Visual Studio, you can use design-time tools to create this expression automatically. When the control performs data binding, it resolves the expression and assigns the resulting value to the specified property. You can data bind as many properties on a control as you need.

> **Note** If you are working with HTML server controls (such as the **HtmlInputText** or **HtmlSelect** control), data-binding procedures are different. For details, see Data Binding HTML Server Controls.

To data bind single-value Web server controls in Design view

1. Select the control, and in the Properties window, click the button in the **(Data Bindings)** property.

   The **DataBindings** dialog box is displayed, showing all the control's properties and data sources available at design time.

2. Under **Bindable Properties**, select the property you want to bind to data.
3. If you want to bind to a data source and member available at design time, select **Simple Binding**. Expand the appropriate node as follows:

   - If you are binding directly to data in a dataset, expand the name of the dataset and then the name of the table. This displays the **DefaultView** object (a data view automatically generated by the dataset), which in turn will expand into **DefaultView.[0]** — that is, to the first record in the data view.

     > **Note** To use this binding effectively, you must set the default data view's RowFilter property at run time so that the data view contains only one record. For details, see Adding Data Views to a Form or Component.

   - If you are adding controls to a template in a **Repeater**, **DataList**, or **DataGrid** control, expand the **Container** node, which references the data source for the parent control. Expand the **DataItem** node and then select the column to bind to. When the control runs, the **DataItem** will contain the record currently being processed by the parent container.

     > **Note** The objects available in the **Simple Binding** tree are these: the **Page** object; any data source the derives from the **Component** class; any objects instantiated in the page's **InitializeComponent** method; and any objects that implement the IEnumerable interface.

   When you select a data source and member, a data-binding expression is constructed and displayed in the disabled **Custom binding expression** box below.

4. If you are working with a property that is of type **String**, and you want to format the results of the expression, select a formatting expression from the **Format** list.
5. If you want to create your own data-binding expression or modify one generated by selections in the **Simple binding**, choose **Custom binding expression**.

   The editing box is enabled and you can type or edit the expression.

   > **Note** You can edit the binding expression in HTML view as well.

6. Repeat Steps 2 through 5 for any other properties you want to bind.
7. Click **OK** to close the dialog box.

   In the Properties window, a data-binding glyph ( ) appears in the property name box to indicate that the property is data bound.

8. Add code to the Web Forms page to call the control's or the page's DataBind method.

Note You generally do not want to call the **DataBind** method on each round trip (that is, in the page initialization without checking for a post back), because doing so replaces the values in controls.

You can also specify data binding by editing the control declaration in HTML view of the Web Forms Designer.

To data bind single-value controls in HTML view

1. Create a data-binding expression and assign it as the value for a property (attribute). The expression's syntax is:

```
<%# expression %>
```

   The following example shows a data-binding expression. The expression gets the value of the `au_lname` column from the first record of a data view and assigns it to the **Text** property of a **TextBox** Web control.

```
<asp:TextBox id="TextBox1" runat="server"
   Text='<%# DataView1(0)("au_lname") %>'>
</asp:TextBox>
```

   Note For more information, see Data-Binding Expressions for Web Forms Pages and Data Binding Expression Syntax.

2. Add code to the Web Forms page to call the control's or the page's DataBind method.

   Note You generally do not want to call the **DataBind** method on each round trip (that is, in the page initialization without checking for a post back), because doing so replaces the values in controls.

See Also

Introduction to Data Access in Web Forms Pages | Data Binding Expression Syntax | Multi-Record and Single-Value Data Binding for Web Forms Pages | Data Binding HTML Server Controls

Data Binding HTML Server Controls

Visual Studio supports data binding for HTML server controls — for example, the HtmlButton and HtmlInputText controls — differently than it does Web server controls such as the **DataList**, **DataGrid**, and **TextBox** controls.

To data bind an HTML server control

- Set the property to bind to a data-binding expression. For details, see Data-Binding Expressions for Web Forms Pages. The expression should resolve to a single value.

  You can set HTML server control properties in the Properties window or in HTML view. If you use the Properties window, enter the data-binding expression without quotation marks around it. In HTML view, put quotation marks around the expression.

  > **Note** If you set the **value** property of a control such as the **HtmlInputText** control, the data-binding expression will appear as the control's text or caption at design time. However, at run time the expression will be evaluated and the result will be displayed instead.

  The following example shows what an **HtmlInputText** control might look like in HTML view after you have bound it to an author's name. The binding assumes that the page contains a data view called `DataView1` that displays information from an `authors` table. The expression gets the `au_lname` value from the first (number zero) record returned by the data view.

```
<INPUT id="Text1"
   type="text"
   name="Text1"
   runat="server"
   value='<%# DataBinder.Eval(DataView1, "[0].au_lname") %>'
>
```

Because the HtmlSelect server control (the HTML **Listbox** and **Dropdown** controls) can display multiple records, it supports special data-binding properties and the data-binding procedure is slightly different.

To data bind an HtmlSelect server control

1. Set the control's DataSource property to a data-binding expression that resolves to multiple values. For details, see Data-Binding Expressions for Web Forms Pages. A typical value is the name of a dataset instance on your page.

   > **Tip** You can set properties in the Properties window or in HTML view. If you use the Properties window, enter them without quotation marks. In HTML view, put quotation marks around the values.

2. If the data source contains multiple members (for example, if you are binding to a dataset with multiple tables in it), set the control's DataMember property to the name of the member to use.
3. If the member contains multiple fields, set the control's DataTextField property to the name of the field to display. For example, this is often the name of a column in a data table.
4. If you want to return a different value than what the user selects (for example, the control displays names but you want to return ID numbers), set the control's DataValueField property.

   The following example shows what an **HtmlSelect** control might look like in HTML view after you have bound it to the authors table in a dataset instance called `DsAuthors1`. The control displays the value of the `au_lname` column and when the user makes a selection, it returns the value of the `au_id` column.

```
<SELECT id="Select1" size="2" name="Select1" runat="server"
   DataSource="<%# DsAuthors1 %>"
   DataMember="authors"
   DataTextField="au_lname"
   DataValueField="au_id">
      <OPTION></OPTION>
</SELECT>
```

See Also

Data-Binding Expressions for Web Forms Pages | Controls You Can Use on Web Forms Pages | ASP.NET Server Controls Recommendations

Data Binding Single-Value Web Server Controls at Run Time

If you want to assign data values to a control property at run time, you can respond to a data-binding event that is raised by the control. In the event code, you can perform whatever logic is required to assign property values.

> **Note** At design time, you can create data-binding expressions that are evaluated at run time to assign values to control properties. For details, see Data Binding Single-Value Web Server Controls at Design Time.

To data bind single-value controls at run time

1. Make sure that code in your Web Forms page calls the DataBind method of either the control you are working with or of the page. The data-binding event will be raised in response to this method call.

   > **Note** You generally do not want to call the **DataBind** method on each round trip (that is, in the page initialization without checking for a post back), because doing so replaces the values in controls.

2. Create an event handler for the control's DataBinding event.
3. In the event handler, add code to assign values to the control's properties as required.

The following example shows how to perform simple data binding at run time. The Web Forms page contains two **TextBox** Web server controls that display information from the titles table in the SQL Server pubs database. Previous and Next buttons set a "record position" variable that is stored in session state. Whenever the user clicks these buttons to change the record position, the page's **DataBind** method is called, which raises the event used to fill the text boxes.

```
' Visual Basic
Private Sub Page_Load(ByVal sender As System.Object, ByVal e _
   As System.EventArgs) Handles MyBase.Load
        SqlDataAdapter1.Fill(DsTitles1)
        If Not (Me.IsPostBack) Then
            Session("RecordPos") = 0
            Me.DataBind()
        End If
End Sub

Private Sub btnNext_Click(ByVal sender As System.Object, ByVal e _
   As System.EventArgs) Handles btnNext.Click
   Dim RecordPos As Integer = CType(Session("RecordPos"), Integer)
   RecordPos += 1
   If RecordPos > DsTitles1.titles.Count Then
       RecordPos -= 1
   End If
   Session("RecordPos") = RecordPos
   Me.DataBind()
End Sub

Private Sub btnPrevious_Click(ByVal sender As System.Object, ByVal e _
   As System.EventArgs) Handles btnPrevious.Click
   Dim RecordPos As Integer = CType(Session("RecordPos"), Integer)
   If RecordPos > 0 Then
       RecordPos -= 1
   End If
   Session("RecordPos") = RecordPos
   Me.DataBind()
End Sub

Private Sub TextBox1_DataBinding(ByVal sender As Object, ByVal e _
   As System.EventArgs) Handles TextBox1.DataBinding
   Dim RecordPos As Integer = CType(Session("RecordPos"), Integer)
   TextBox1.Text = DsTitles1.titles(RecordPos).title_id
   TextBox2.Text = DsTitles1.titles(RecordPos).title
End Sub

// C#
private void Page_Load(object sender, System.EventArgs e)
{
```

```
   sqlDataAdapter1.Fill(dsTitles1);
   if (!IsPostBack)
   {
Session["RecordPos"] = 0;
this.DataBind();
   }
}

private void btnNext_Click(object sender, System.EventArgs e)
{
    int RecordPos = (int) Session["RecordPos"];
    RecordPos++;
    if (RecordPos > dsTitles1.titles.Count)
    {
        RecordPos--;
    }
    Session["RecordPos"] = RecordPos;
    this.DataBind();
}

private void btnPrevious_Click(object sender, System.EventArgs e)
{
    int RecordPos = (int) Session["RecordPos"];
    if (RecordPos > 0)
    {
        RecordPos--;
    }
    Session["RecordPos"] = RecordPos;
    this.DataBind();
}

private void TextBox1_DataBinding(object sender, System.EventArgs e)
{
    int RecordPos = (int) Session["RecordPos"];
    TextBox1.Text = dsTitles1.titles[RecordPos].title_id;
    TextBox2.Text = dsTitles1.titles[RecordPos].title;
}
```

The following example binds a **Label** control to a value returned by executing a SQL command. The command object (which is explicitly for a SQL Server, as indicated by the named parameter) has previously been defined with a single parameter called `@empid`.

The user enters an employee ID and clicks a button. The button calls the **Label** control's **DataBind** method, which in turn raises the **DataBinding** event. In the event handler, the code gets the employee ID, sets it as a parameter for a **SqlCommand** object, and runs the command's SQL statement. The command returns a data reader object with one record in it, which the code reads and displays in the label.

> **Security Note** User input in a Web Forms page can include potentially malicious client script. By default, the Web Forms page validates that user input does not include script or HTML elements. For more information, see Scripting Exploits and Protecting Against Script Exploits in a Web Application.

```
' Visual Basic
Private Sub btnLogin_Click(ByVal sender As System.Object, ByVal e _
   As System.EventArgs) Handles btnLogin.Click
   Label1.DataBind()
End Sub

    Private Sub Label1_DataBinding(ByVal sender As Object, ByVal e _
       As System.EventArgs) Handles Label1.DataBinding
        SqlCommand1.Parameters("@empid").Value = LoginID.Text
        SqlCommand1.CommandText = _
           "SELECT emp_id, fname, lname FROM employee WHERE (emp_id = @empid)"
        SqlCommand1.CommandType = CommandType.Text
        Dim dr As System.Data.SqlClient.SqlDataReader
        SqlConnection1.Open()
        dr = SqlCommand1.ExecuteReader()
        If dr.Read() Then
```

```
            Label1.Text = "Welcome, " & CType(dr(1), String) & " " & CType(dr(2), String)
        Else
            Label1.Text = "Login not successful."
        End If
        dr.Close()
        SqlConnection1.Close()
    End Sub

// C#
private void btnLogin_Click(object sender, System.EventArgs e)
{
   Label1.DataBind();
}

private void Label1_DataBinding(object sender, System.EventArgs e)
{
   sqlCommand1.Parameters["@empid"].Value = LoginID.Text;
   sqlCommand1.CommandText =
      "SELECT emp_id, fname, lname FROM employee WHERE (emp_id = @empid)";   sqlCommand1.Comm
andType = CommandType.Text;
   System.Data.SqlClient.SqlDataReader dr;
   sqlConnection1.Open();
   dr = sqlCommand1.ExecuteReader();
   if (dr.Read())
   {
      Label1.Text = "Welcome, " + dr[1] + " " + dr[2];
   }
   else
   {
      Label1.Text = "Login not successful."
   }
   dr.Close();
   sqlConnection1.Close();
}
```

See Also

Introduction to Data Access in Web Forms Pages | Multi-Record and Single-Value Data Binding for Web Forms Pages | Data Binding Single-Value Web Server Controls at Design Time | Data Binding HTML Server Controls

ASP.NET Server Controls

Server controls are specifically designed to work with Web Forms pages. They differ from controls for Windows applications because they work within the ASP.NET page framework. Because of this, server controls feature unique design considerations. The topics listed here provide background information on what server controls are and how they work.

In This Section

Introduction to ASP.NET Server Controls
Describes the general types of server controls and how they differ.
Creating and Programming ASP.NET Server Controls
Lists topics that provide the information about how to work with these controls.
Web Forms Validation
Lists topics that provide information on how to use the validation controls to verify user input on a Web Forms page.
Client Script in Web Forms Pages
Describes how you can combine client (browser) script with server-based code in a Web Forms page.
Controls You Can Use on Web Forms Pages
Links to a description of each type of ASP.NET server control.
Web User Controls and Web Custom Controls
Provides an overview of user controls (controls you create as Web Forms pages and then embed in other Web Forms pages) and of custom controls (controls you build by deriving from the base Web Forms control class).

Related Sections

Web Forms Pages
Lists topics pertaining to Web Forms generally.
ASP.NET Server Controls by Function
Provides a breakdown of what types of controls are available and what each is for.
ASP.NET Server Controls Recommendations
Provides suggestions for when to use each type of ASP.NET server control and why.

Code for Web Forms Controls

This topic is designed to help you find code that demonstrates how to perform common programming tasks with Web Forms controls.

Note For tasks that pertain to Web Forms pages generally, see Code for Web Applications.

The following categories are represented in the list below:

• Adding Controls to a Page • Properties, Attributes, and Styles • Validation • Templates • AdRotator Control • Button Control • Calendar Control • CheckBox, RadioButton Controls • DataGrid Control	• DataList, Repeater Controls • ListBox, DropDown, CheckBoxList, RadioButtonList Controls • Placeholder Control • Table, TableRow, TableCell Controls • TextBox Control • User Controls and Custom Controls • XML Control • Related Code Examples

You can also find sample code for Web Forms controls in the following locations:

- Web Applications Example Topics
- Web Forms Controls Example Topics
- Visual Basic Sample Applications
- Visual Basic QuickStarts and Other Samples
- .NET Framework Reference Information
- Developing ASP.NET Server Controls

Adding Controls to a Page

Example of...

Adding Web server controls using HTML syntax

Adding HTML server controls using HTML syntax

Adding controls at run time (general)

Adding controls at run time (Placeholder Control)

Referencing controls on the page

Properties, Attributes, and Styles

Example of...

Setting properties using HTML syntax

Setting properties of Web server controls dynamically

Setting properties of HTML server controls dynamically

Adding HTML attributes to ASP.NET server controls

Applying CSS styles to controls

Validation

Example of...

Creating a Web Forms page with several controls on it

Validating user input against a database

Creating custom validation logic

Testing for valid input

Displaying custom error messages

Validating programmatically

Templates

Example of...

Creating templates using HTML syntax

Creating templates from a template definition file

Creating templates in code

Creating templated columns for a DataGrid control in code

AdRotator Control

Example of...

Specifying an ad programmatically

Tracking responses to an ad

Button Control

Example of...

Responding to a button click

Determining where the user clicked in an ImageButton control

Calendar Control

Example of...

Responding when the user selects a specific day or date range

Responding when the user changes months

Customizing the appearance of individual days

Format individual elements in the calendar

Setting the current date

Selecting specific dates

Selecting specific dates

CheckBox, RadioButton Controls

Example of...

Adding items at run time

Getting or setting the check box state

Getting or setting the radio button state

Responding when the user checks the checkbox

Responding when the user clicks a radiobutton

DataGrid Control

Example of...

Sorting the grid

Deleting items

Editing items

Selecting rows

Responding when the user clicks a button in a DataGrid control

Customizing grid entries at run time

Creating templated columns dynamically

Displaying items in pages

Creating templates from a template definition file

Creating templated columns for a DataGrid control in code

DataList, Repeater Controls

Example of...

Selecting items in a DataList control

Editing items in a DataList control

Deleting items in a DataList control

Responding when the user clicks a button in a DataList or Repeater control

Creating templates from a template definition file

Creating templates in code

Customizing items at run time

ListBox, DropDown, CheckBoxList, RadioButtonList Controls

Example of...

Adding items to the control

Data-binding the control

Responding when the user makes a selection

Setting the selection

Determining the current selection

Setting layout in a CheckBoxList or RadioButtonList control

Placeholder Control

Example of...

Adding child controls at run time

Table, TableRow, TableCell Controls

Example of...

Adding rows and cells at run time

Displaying database information in a table

TextBox Control

Example of...

Getting or setting text box values

Responding when the user changes text box values

Filtering HTML elements out of user input

Allowing only predetermined characters in user input

User Controls and Custom Controls

Example of...

Creating a simple user control (general)

Creating a simple user control (walkthrough)

Adding a user control to a Web Forms page

Creating a simple custom control

XML Control

Example of...

Displaying XML data using a transformation

Loading XML data into the control

Applying transformations

Related Code Examples

Code for Data Access

Code Examples for Visual Basic Programmers

Introduction to ASP.NET Server Controls

When you create Web Forms pages, you can use these types of controls:

- **HTML server controls** HTML elements exposed to the server so you can program them. HTML server controls expose an object model that maps very closely to the HTML elements that they render.
- **Web server controls** Controls with more built-in features than HTML server controls. Web server controls include not only form-type controls such as buttons and text boxes, but also special-purpose controls such as a calendar. Web server controls are more abstract than HTML server controls in that their object model does not necessarily reflect HTML syntax.
- **Validation controls** Controls that incorporate logic to allow you to test a user's input. You attach a validation control to an input control to test what the user enters for that input control. Validation controls are provided to allow you to check for a required field, to test against a specific value or pattern of characters, to verify that a value lies within a range, and so on.
- **User controls** Controls that you create as Web Forms pages. You can embed Web Forms user controls in other Web Forms pages, which is an easy way to create menus, toolbars, and other reusable elements.

> **Note** You can also create output for mobile devices. To do so, you use the same ASP.NET page framework, but you create Mobile Web Forms instead of Web Forms pages and use controls specifically designed for mobile devices. For details, see Creating ASP.NET Mobile Web Applications.

You can use all types of controls on the same page. The following sections provide more detail about ASP.NET server controls. For more information about validation controls, see Web Forms Validation. For information about user controls, see Introduction to Web User Controls.

> **Note** In some situations, server controls require client script in order to function properly. If a user has disabled scripting in the browser, the controls might not function as you intend. For details, see ASP.NET Server Controls and Browser Capabilities.

HTML Server Controls

HTML server controls are HTML elements containing attributes that make them visible to — and programmable on — the server. By default, HTML elements on a Web Forms page are not available to the server; they are treated as opaque text that is passed through to the browser. However, by converting HTML elements to HTML server controls, you expose them as elements you can program on the server.

The object model for HTML server controls maps closely to that of the corresponding elements. For example, HTML attributes are exposed in HTML server controls as properties.

Any HTML element on a page can be converted to an HTML server control. Conversion is a simple process involving just a few attributes. As a minimum, an HTML element is converted to a control by the addition of the attribute RUNAT="SERVER". This alerts the ASP.NET page framework during parsing that it should create an instance of the control to use during server-side page processing. If you want to reference the control as a member within your code, you should also assign an ID attribute to the control.

The page framework provides predefined HTML server controls for the HTML elements most commonly used dynamically on a page: forms, the HTML <INPUT> elements (text box, check box, Submit button, and so on), list box (<SELECT>), table, image, and so on. These predefined HTML server controls share the basic properties of the generic control, and in addition, each control typically provides its own set of properties and its own event.

HTML server controls offer the following features:

- An object model that you can program against on the server using the familiar object-oriented techniques. Each server control exposes properties that allow you to manipulate the control's HTML attributes programmatically in server code.
- A set of events for which you can write event handlers in much the same way you would in a client-based form, except that the event is handled in server code.
- The ability to handle events in client script.
- Automatic maintenance of the control's state. If the form makes a round trip to the server, the values that the user entered into HTML server controls are automatically maintained when the page is sent back to the browser.
- Interaction with validation controls so you can easily verify that a user has entered appropriate information into a control.
- Data binding to one or more properties of the control.
- Support for HTML 4.0 styles if the Web Forms page is displayed in a browser that supports cascading style sheets.
- Pass-through of custom attributes. You can add any attributes you need to an HTML server control and the page framework

will read them and render them without any change in functionality. This allows you to add browser-specific attributes to your controls.

For details about how to convert an HTML element to an HTML server control, see Adding HTML Server Controls to a Web Forms Page.

Web Server Controls

Web server controls are a second set of controls designed with a different emphasis. They do not map one-to-one to HTML server controls. Instead, they are defined as abstract controls in which the actual HTML rendered by the control can be quite different from the model that you program against. For example, a **RadioButtonList** Web server control might be rendered in a table or as inline text with other HTML.

Web server controls include traditional form controls such as buttons and text boxes as well as complex controls such as tables. They also include controls that provide commonly used form functionality such as displaying data in a grid, choosing dates, and so on.

Web server controls offer all of the features described above for HTML server controls (except one-to-one mapping to HTML elements) and these additional features:

- A rich object model that provides type-safe programming capabilities.
- Automatic browser detection. The controls can detect browser capabilities and create appropriate output for both basic and rich (HTML 4.0) browsers.
- For some controls, the ability to define your own look for the control using templates.
- For some controls, the ability to specify whether a control's event causes immediate posting to the server or is instead cached and raised when the form is submitted.
- Ability to pass events from a nested control (such as a button in a table) to the container control.

At design time in HTML view, the controls appear in your page in a format such as:

```
<asp:button attributes runat="server"/>
```

The attributes in this case are not those of HTML elements. Instead, they are properties of the Web control.

When the Web Forms page runs, the Web server control is rendered on the page using appropriate HTML, which often depends not only on the browser type but also on settings that you have made for the control. For example, a **TextBox** control might render as an <INPUT> tag or a <TEXTAREA> tag, depending on its properties.

Detailed how-to and reference documentation is available for each control separately. For more information, see Controls You Can Use on Web Forms Pages.

See Also

Web Forms Pages | ASP.NET Server Controls Recommendations | ASP.NET Server Controls by Function | Web Forms Validation | Introduction to Web User Controls

ASP.NET Server Controls by Function

The ASP.NET server controls are listed here according to general function. The following sections individually provide details about Web server controls, HTML server controls, and validation controls.

> **Note** For more about the differences between Web and HTML server controls, see Introduction to ASP.NET Server Controls.

Web Server Controls

Web server controls include traditional user-entry controls as well as special-purpose controls such as a calendar. All Web server controls can be bound to a data source.

> **Note** You can also create output for mobile devices. To do so, you use the same ASP.NET page framework, but you create Mobile Web Forms instead of Web Forms pages and use controls specifically designed for mobile devices. For details, see Creating ASP.NET Mobile Web Applications.

The Web server controls are listed below according to their function.

> **Security Note** User input in a Web Forms page can include potentially malicious client script. By default, the Web Forms page validates that user input does not include script or HTML elements. For more information, see Scripting Exploits and Protecting Against Script Exploits in a Web Application.

Function	Control	Description
Text display (read only)	Label	Displays text that users cannot directly edit.
Text edit	TextBox	Displays text that can be edited by users at run time or changed programmatically.
Selection from a list	DropDownList	Allows users to select from a list that appears when the user clicks a button.
	ListBox	Displays a list of choices. Optionally, the list can allow multiple selections.
Graphics display	Image	Displays an image.
	AdRotator	Displays a sequence (predefined or random) of images.
Value setting	CheckBox	Displays a box that users can click to turn on and off.
	CheckBoxList	Creates a grouping of check boxes. The list control makes it easy to create check boxes using data binding.
	RadioButton	Displays a single button that can be turned on or off.
	RadioButtonList	Creates a grouping of radio buttons. Inside the group, only one button can be selected.
Date setting	Calendar	Displays a graphic calendar to allow users to select a date.
Commands **Note** These controls always cause the form to be posted to the server for processing.	Button	Performs a task.
	LinkButton	Like a **Button** control, but has the appearance of a hyperlink.
	ImageButton	Like a **Button** control, but incorporates an image instead of text.
Navigation	HyperLink	Creates a Web navigation link.
Table manipulation	Table	Creates a table.
	TableCell	Creates an individual cell within a table row.
	TableRow	Creates an individual row within a table.
Grouping other controls	CheckBoxList	Creates a collection of check boxes.
	Panel	Creates a borderless division on the form that serves as a container for other controls.
	RadioButtonList	Creates a grouping of radio buttons. Inside the group, only one button can be selected.

Selection from a list	Repeater	Displays information from a data source using a set of HTML elements and controls you specify, repeating the elements once for each record in the data set.
	DataList	Like the **Repeater** control, but with more formatting and layout options, including the ability to display information in a table. The **DataList** control also allows you to specify editing behavior.
	DataGrid	Displays information, usually data-bound, in tabular form with columns. Provides mechanisms to allow editing and sorting.
Place holding	PlaceHolder	Enables you to place an empty container control in the page and then dynamically add child elements to it at run time.
	Literal	Renders static text into a Web page without adding any HTML elements.
	XML	Reads XML and writes it into a Web Forms page at the location of the control.

HTML Server Controls

HTML server controls map directly to HTML elements. The controls listed in this section are predefined controls. However, any HTML element can be made into a control; in that case, the control supports a generic set properties and methods. All HTML server controls can be bound to a data source.

Function	Control	Description
Forms	HtmlForm	Defines an HTML form. The values of controls within the form are posted to the server when the form is submitted.
Text edit	HtmlInputText	Displays text that can be edited by users at run time, or changed programmatically.
	HtmlTextArea	Displays large quantities of text. Used for multi-line text entry and display.
Commands **Note** These controls always cause the form to be posted to the server for processing.	HtmlAnchor	Enables Web navigation.
	HtmlButton	Performs a task. This control can contain any arbitrary HTML, and therefore it is very flexible in look and feel. However, it is not compatible with all browsers.
	HtmlInputButton	Performs a task. This button is supported on all browsers.
	HtmlInputImage	Like a button, but displays a graphic.
Selection from a list	HtmlSelect	Displays a list of text and graphical items (icons).
Graphics display	HtmlImage	Displays an image.
Information storage	HtmlInputHidden	Stores state information for a form (information that needs to be available with each round trip to the server).
Navigation	HtmlAnchor	Creates a Web navigation link.
Value setting	HtmlInputCheckbox	Creates a box that users can click to turn on and off. The **CheckBox** control includes a label.
	HtmlInputRadioButton	Displays a button that can be turned on or off. Radio buttons are typically used to allow the user to select one item from a short list of fixed options.
Table manipulation	HtmlTable	Creates a table.
	HtmlTableRow	Creates an individual row within a table.
	HtmlTableCell	Creates an individual cell within a table row.
File transfer	HtmlInputFile	Allows users to specify files to upload to a server. (The server must allow uploads.)
Other functions	HtmlGenericControl	Creates a basic object model (properties, methods, events) for any HTML element converted to a control.

Validation Controls

Validation controls provide you with a way to check user input in Web or HTML server controls. The following table lists the type

of validation controls you can use in Web Forms. For more details, see Web Forms Validation.

Type of validation	Control to use	Description
Required entry	**RequiredFieldValidator**	Ensures that the user does not skip an entry.
Comparison to a value	**CompareValidator**	Compares a user's entry against a constant value or a property value of another control using a comparison operator (less than, equal, greater than, and so on).
Range checking	**RangeValidator**	Checks that a user's entry is between specified lower and upper boundaries. You can check ranges within pairs of numbers, alphabetic characters, and dates. Boundaries can be expressed as constants or as values derived from another control.
Pattern matching	**RegularExpressionValidator**	Checks that the entry matches a pattern defined by a regular expression. This type of validation allows you to check for predictable sequences of characters, such as those in social security numbers, e-mail addresses, telephone numbers, postal codes, and so on.
User-defined	**CustomValidator**	Checks the user's entry using validation logic that you code yourself.
None	**ValidationSummary**	Displays the error messages of all other validation controls on the page in a single unified summary.

See Also

Creating and Programming ASP.NET Server Controls | ASP.NET Server Controls Recommendations | Web Forms Validation

Creating and Programming ASP.NET Server Controls

ASP.NET server controls expose a unique programming model that differs from that used with controls on Windows Forms. The topics in this section provide the information you need to understand how to work with these controls.

In This Section

Adding Web Server Controls to a Web Forms Page
Describes how to add Web server controls (as opposed to HTML server controls) to a page.
Adding HTML Server Controls to a Web Forms Page
Describes how to add HTML server controls (as opposed to Web server controls) to a page.
Converting HTML Server Controls to HTML Elements
Describes how to convert existing HTML elements on a page into controls.
Adding Controls to a Web Forms Page Programmatically
Describes how to instantiate server controls dynamically in code.
ASP.NET Server Controls and CSS Styles
Provides information on how server controls interact with cascading style sheets (CSS) and how you can set styles for controls.
Customizing the Appearance of ASP.NET Server Controls Using Styles
Describes how to manipulate CSS styles in ASP.NET server controls.
Web Server Controls Templates
Provides information on using templates — snippets of HTML combined with controls — to define individual elements of a complex control (for example, the header, footer, and selected row in a **DataList** control).
Creating Web Server Control Templates
Describes how to define templates in either the Web Forms Designer or directly in the .aspx file.
Creating Web Server Control Templates Dynamically
Describes how to create a template at run time and assign it to a **DataList** or **Repeater** control.
Creating Templates Programmatically in the DataGrid Control
Describes how to create a template column at run time for a **DataGrid** control.
Setting ASP.NET Server Control Properties
Describes how to set properties using the Web Forms Designer or by setting properties in the .aspx file.
Setting Web Server Control Properties Programmatically
Describes how to set properties such as colors, units (pixels) in code for Web server controls (as opposed to HTML server controls).
Setting HTML Server Control Properties Programmatically
Describes how to set properties of HTML server controls (as opposed to Web server controls) in code.
Specifying Whether Controls Post to the Server
Describes how to specify whether events in controls such as the **CheckBox** control cause a form to be sent to the server or, instead, cache event information until the form is posted some other way.
Web Forms Control Identification
Discusses how the ASP.NET page framework provides mechanisms to ensure that controls on a page and in an application have unique identifiers.
Referencing Controls in Web Forms Pages
Explains how to get access to controls when you are working with naming containers and child controls.

Related Topics

Creating and Managing Web Forms Pages
Describes how to create and edit Web Forms.
Programming Web Forms
Provides information on how to code Web Forms, including information on page processing stages, and so on.
Web Forms Pages
Provides a summary of topics pertaining to Web Forms generally.
Creating Templates Programmatically in the DataGrid Control
Describes how to create a template column at run time for a **DataGrid** control.
Introduction to Validating User Input in Web Forms
Describes how you can validate user input made with ASP.NET server controls.

Adding Web Server Controls to a Web Forms Page

You add controls to a Web Forms page much the same way you add any HTML element. You can either use the Visual Studio Web Forms Designer and add a control from the Toolbox, or type the element representing the control into the HTML.

> **Note** You can also add controls programmatically at run time. For details, see Adding Controls to a Web Forms Page Programmatically.

Adding Web Server Controls Using the Web Forms Designer

The Toolbox contains tabs with controls that you can use when you design a Web Forms page.

To add a Web server control to a Web Forms page

1. Switch to Design view.
2. From the **Web Forms** tab of the Toolbox, drag the control onto the form.

   A glyph ( ) appears on the control in Design view to indicate that it is a server-based control.

   > **Note** By default, the Web Forms page uses Grid layout, and you place controls at absolute positions on the page (using x and y coordinates). If you want to use linear layout, in which the page elements flow as in a word processing document, you can change the page's **pageLayout** property, or include a **Flow Layout Panel** HTML server control. For more details, see Positioning HTML Elements in Design View.

Adding Web Server Controls using ASP.NET Syntax

You can add a control to a page by declaring it directly in the .aspx file.

> **Note** For background information about controls for Web Forms pages, see Introduction to ASP.NET Server Controls.

To add a control using ASP.NET syntax

1. Switch to HTML view.
2. Type the element representing the control into the .aspx file. In Visual Studio, you do this in HTML view. The exact syntax you use depends on the control you are adding, but in general the following applies:

   - Controls must include the attribute **runat="server"**.
   - Set the control's **ID** attribute unless the control is part of a complex control and will be repeated (as in the **Repeater**, **DataList**, and **DataGrid** controls).
   - Web server controls are declared with an XML tag that references the **asp** namespace.
   - Control declarations must be properly closed. You can specify an explicit closing tag, or, if the control has no child elements, you can specify a self-closing tag. The only exceptions are HTML input controls that cannot have child elements, such as the input controls (for example, HtmlInputText, HtmlImage, and HtmlButton).
   - Control properties are declared as attributes.

     The following examples show typical declarations for Web server controls:

```
<!-- Textbox Web server control -->
<asp:textbox id=TextBox1 runat="Server" Text=""></asp:textbox>

<!-- Same, but with self-closing element -->
<asp:textbox id=Textbox1 runat="Server" Text="" />

<!-- Web DropDownList control, which contains subelements -->
<asp:DropDownList id=DropDown1 runat="server">
   <asp:ListItem Value="0">0</asp:ListItem>
   <asp:ListItem Value="1">1</asp:ListItem>
   <asp:ListItem Value="2">2</asp:ListItem>
   <asp:ListItem Value="3">3</asp:ListItem>
</asp:DropDownList>
```

```
<asp:Repeater id=Repeater2 runat="server">
   <HeaderTemplate>
       Company data:
   </HeaderTemplate>
   <ItemTemplate>
       <asp:Label runat="server"
           Font-Name="verdana" Font-Size="10pt"
           Text='<%# Container.DataItem.Name %>' />
       ( <asp:Label runat=server
             Font-Name="verdana" Font-Size="10pt"
             Text='<%# Container.DataItem.Ticker %>'/>
        )
   </ItemTemplate>
   <SeparatorTemplate>
       ,
   </SeparatorTemplate>
</asp:Repeater>
```

For information about the declarative syntax for a specific Web server control, search in the Index in Help for "Web controls" and choose the name of the control you want to use.

Note If the Web Forms Designer cannot render a Web server control correctly, it displays a grey box with the text "Error Creating Control." This often means that the ASP.NET syntax of the control is incorrect — for example, if the
runat="server"
attribute is missing in a Web server control element, you will see this error. Point to the information icon ( ); a ToolTip is displayed with details about the error.

See Also

ASP.NET Server Controls | Adding Controls to a Web Forms Page Programmatically |
Adding HTML Server Controls to a Web Forms Page | Converting HTML Server Controls to HTML Elements |
Creating Event Handlers in Web Forms Pages | Setting HTML Server Control Properties Programmatically |
Setting ASP.NET Server Control Properties | Web Forms Validation | Referencing Controls in Web Forms Pages

Adding HTML Server Controls to a Web Forms Page

Adding HTML server controls is like adding any HTML element, except that you convert the element to a control so that you can work with it in server code.

> **Note** For background information about controls for Web Forms pages, see Introduction to ASP.NET Server Controls.

Adding HTML Server Controls Using the Web Forms Designer

Adding HTML server controls to a Web Forms page is a two-step process.

To add an HTML server control to a Web Forms page

1. From the **HTML** tab of the Toolbox, drag an HTML element onto the page.
2. Convert the element to a control by right-clicking it and choosing **Run As Server Control**.

   The Web Forms Designer adds the attribute `runat="server"` to the element, which alerts the server to treat the element as a control. A glyph (▶) appears on the control in Design view to indicate that it is a server-based control.

   > **Note** By default, the Web Forms page uses Grid layout, and you place controls at absolute positions on the page (using x and y coordinates). If you want to use linear layout, in which the page elements flow as in a word processing document, you can change the page's **pageLayout** property, or include a **Flow Layout Panel** HTML server control. For more details, see Positioning HTML Elements in Design View.

Adding HTML Server Controls using ASP.NET Syntax

You can add an HTML server control to a page by declaring it directly in the .aspx file.

To add an HTML server control using ASP.NET syntax

1. Switch to HTML view.
2. Type the HTML syntax of the element you want to use as a control. Include the normal HTML syntax for the element, and in addition, do the following:

   - Set the control's **ID** attribute (property) to a unique value for that page, unless the control is part of a complex control and will be repeated (as in **Repeater**, **DataList**, and **DataGrid** controls).
   - Set the **runat="server"** attribute to convert the element to a control.

   The following example shows how to declare an HtmlInputText control, an HtmlInputButton control, and an HtmlAnchor control:

```
<input id="Name" type=text size=40 runat="server">
<input type=submit id="Enter" value="Enter" runat="server">
Click <a id="Anchor1" runat="server" href="more.html">More </a> to see the next page.
```

For information about the declarative syntax for a specific HTML server control, search in the Index in Help for "HTML server controls" and choose name of the control you want to use.

See Also

ASP.NET Server Controls | Adding Controls to a Web Forms Page Programmatically | Adding Web Server Controls to a Web Forms Page | Converting HTML Server Controls to HTML Elements | Creating Event Handlers in Web Forms Pages | Setting HTML Server Control Properties Programmatically | Web Forms Validation | Referencing Controls in Web Forms Pages

Converting HTML Server Controls to HTML Elements

If you no longer need to program an HTML server control in server code, you should convert it back to a plain HTML element. Each HTML server control in a form uses resources, so it is a good practice to minimize the number of controls that the Web Forms page has to work with.

To convert an HTML server control to an HTML element

- In Design view or in the Document Outline window, right-click the control and clear the check mark next to **Run As Server Control**.

You can also perform the conversion by editing HTML directly.

To convert an HTML server control to a HTML element using ASP.NET syntax

- In HTML view, remove the attribute `runat="server"` from the control's tag.

  You do not have to remove the ID attribute. You want to be especially careful not to remove the ID attribute if there is client script that references the element.

See Also

Adding Controls to a Web Forms Page Programmatically | Adding HTML Server Controls to a Web Forms Page | Creating Event Handlers in Web Forms Pages | Setting HTML Server Control Properties Programmatically | Web Forms Validation | ASP.NET Server Controls

Adding Controls to a Web Forms Page Programmatically

There are times when it is more practical to create a control at run time than at design time. For example, imagine a search results page in which you want to display results in a table. Because you do not know how many items will be returned, you want to dynamically generate one table row for each returned item.

> **Note** Existing controls can often provide the functionality you get from creating controls dynamically. For example, controls such as the **Repeater**, **DataList**, and **RadioButtonList** controls can dynamically create rows or other control elements when the page runs.

In order to programmatically add a control to a page, there must be a container for the new control. For example, if you are creating table rows, the container is the table. If there is no obvious control to act as container, you can use a **Placeholder** or **Panel** Web server control.

In some instances, you might want to create both static text and controls. To create static text, you can use either a **Literal** or a **Label** Web server control. You can then add these controls to the container as you would any other control.

> **Note** Manipulating controls programmatically is often used when creating composite controls. For details, see Developing a Composite Control.

Dynamic Controls and View State

When a control is created dynamically at run time, some information about the control is stored in the view state that is rendered with the page. When the page is posted back to the server, however, non-dynamic controls (those defined on the page) are instantiated in the **Page.Init** event and view state information is loaded before the dynamic controls can be recreated (generally in the **Page_Load** handler). Effectively, before the dynamic controls are recreated, view state is temporarily out of sync with the page's controls. After the **Page_Load** event has run, but before control event handling methods are called, the remaining view state information is loaded into the dynamically created controls.

In most scenarios, this view state processing model works fine. Typically, you add dynamic controls to the end of a container's collection of controls. The view state information stored for the dynamic controls is therefore extra information at the end of the view state structure for the appropriate container, and the page can ignore it until after the controls are created.

However, view state information about dynamically created controls can be a problem in two scenarios:

- If you insert dynamic controls between existing controls.
- If you insert controls dynamically and then reinsert them during a round trip, but with different values.

If you insert dynamic controls between existing controls, the dynamic control's view state information is inserted into the corresponding location of the view state structure. When the page is posted and the view state is loaded, the dynamic control does not yet exist; therefore, the extra information in view state does not correspond to the right control. The result is usually an error indicating an invalid cast.

If you reinsert controls with each round trip, each generation of dynamically created controls will pick up property values from the view state of the preceding set of controls. In many cases, you can avoid this problem by setting the EnableViewState property of the container control to **false**. In that case, no information about the dynamic controls is saved, and there is no conflict with successive versions of the controls.

For details about view state, see Web Forms Page Processing, Introduction to Web Forms State Management, and Control.ViewState.

To add a control to a Web Forms page programmatically

1. Create an instance of the control and set its properties:

```
' Visual Basic
Dim myLabel As Label = New Label()
myLabel.Text = "Sample Label"

// C#
Label myLabel = new Label();
myLabel.Text = "Sample Label";
```

> **Tip** Controls are typically added to the page during the page's load stage. For details about page stages, see

Web Forms Page Processing.

2. Add the new control to the **Controls** collection of a container already on the page:

```
' Visual Basic
Panel1.Controls.Add(myLabel)

// C#
Panel1.Controls.Add(myLabel);
```

> **Note** Because **Controls** is a collection, it is possible to use the AddAt method to place the new control at a specific location — for example, in front of other controls. However, this can introduce errors into the page. For details, see "Dynamic Controls and View State" earlier in this topic.

The following example shows the event handler for the **SelectedIndexChanged** event of a control called "DropDownList1". The handler creates as many label controls as the user has selected from the drop-down list. The labels display the word "Label" plus a counter. The container for the controls is a **Placeholder** Web server control called "Placeholder1."

> **Security Note** User input in a Web Forms page can include potentially malicious client script. By default, the Web Forms page validates that user input does not include script or HTML elements. For more information, see Scripting Exploits and Protecting Against Script Exploits in a Web Application.

```
' Visual Basic
Private Sub DropDownList1_SelectedIndexChanged(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles DropDownList1.SelectedIndexChanged
   Dim i As Integer
   Dim numlabels As Integer
   ' Get the number of labels to create.
   numlabels = CInt(DropDownList1.SelectedItem.Text)
   For i = 1 To numlabels
      Dim myLabel As Label = New Label()
      ' Set the label's Text and ID properties.
      myLabel.Text = "Label " & i
      myLabel.ID = "Label" & i
      PlaceHolder1.Controls.Add(myLabel)
      ' Add a spacer in the form of an HTML <BR> element
      Dim spacer As LiteralControl = New LiteralControl("<br>")
      PlaceHolder1.Controls.Add(spacer)
   Next
End Sub

// C#
private void DropDownList1_SelectedIndexChanged(object sender, System.EventArgs e)
{
   // Get the number of labels to create.
   int numlabels = System.Convert.ToInt32(DropDownList1.SelectedItem.Text);
   for (int i=1; i<=numlabels; i++)
   {
      Label myLabel = new Label();
      // Set the label's Text and ID properties.
      myLabel.Text = "Label" + i.ToString();
      myLabel.ID = "Label" + i.ToString();
      PlaceHolder1.Controls.Add(myLabel);
      // Add a spacer in the form of an HTML <BR> element.
      PlaceHolder1.Controls.Add(new LiteralControl("<br>"));
   }
}
```

See Also

Setting Web Server Control Properties Programmatically | Setting HTML Server Control Properties Programmatically | ASP.NET Server Controls | Referencing Controls in Web Forms Pages

ASP.NET Server Controls and CSS Styles

You can control the look of controls by setting various appearance properties such as **ForeColor**, **BackColor**, **Height**, and **Width**. In addition, some controls support style objects that expose additional style-related settings.

> **Note** Web Forms pages (.aspx files) function as HTML pages at run time. You can therefore use cascading style sheets (CSS) to set the appearance of any elements on the page other than Web server controls. For details, see Applying a CSS Style to Another Web Page.

The sections below provide information about how style information is rendered to the browser and how to work with styles both at design time and programmatically.

Rendering Appearance Properties to the Browser

When the page runs, appearance properties are rendered according to the capabilities of the user's browser. If the user's browser supports cascading stylesheets (CSS), the appearance properties are rendered as style attributes of the HTML elements that make up the control. For example, if you create a **LinkButton** Web server control and set its ForeColor property to **Red**, its FontInfo.Bold property to **true**, and its FontInfo.Size property to **xx-small**, the control is rendered as the following if the user's browser supports stylesheets:

```
<a id="HyperLink1" style="color: Red; font-size: XX-Small; font-weight: bold;">HyperLink</a>
```

On the other hand, if the user's browser does not support styles, the control is rendered using other means, such as a <FONT> element. The following shows the rendering for the example from above, but for a browser that does not support styles:

```
<a id="HyperLink1"><b><font color="Red" size="1">HyperLink</font></b></a>
```

Other examples of properties that are rendered differently depending on the browser are **BorderWidth** and **BorderColor**.

Some appearance properties, such as **BorderStyle**, cannot be rendered without using styles. These properties are therefore ignored in downlevel browsers. For more information, see ASP.NET Server Controls and Browser Capabilities.

Control Style Objects

In addition to simple appearance properties such as **ForeColor** and **BackColor**, controls expose one or more style objects that encapsulate additional appearance properties. One example is the **Font** style property, which exposes an object of type **FontInfo** containing individual properties pertaining to the font, such as **Size**, **Name**, **Bold**, and so on.

Some controls expose style objects you can use to set the appearance of specific portions of the control. For example, the **Calendar** Web server control contains style objects such as DayStyle (individual days), SelectedDayStyle (a day, week, or month selected by the user), and WeekendDayStyle. Using the **SelectedDayStyle** style object, for example, you can set the **BackColor** and **ForeColor** properties of the day selected by the user.

Most style objects are of type **TableItemStyle**, because they are setting the attributes of table cells. The **Font** style object is of type **FontStyle**.

Precedence and Inheritance of Style Objects

In complex controls, style objects often inherit characteristics from other style objects. For example, in the **Calendar** control, the **SelectedDayStyle** object is based on the **DayStyle** object. If you do not explicitly set any properties for **SelectedDayStyle**, it inherits its characteristics from the **DayStyle** object.

This inheritance means that style-object properties you set have an order of precedence. For example, the following list shows the order of style-object properties for the **Calendar** control, with the highest precedence given to the settings on the object last on the list.

1. Appearance properties of the base Calendar control
2. **DayStyle** style object
3. **WeekendDayStyle** style object
4. OtherMonthDayStyle style object
5. **TodayStyle** style object
6. **SelectedDayStyle** style object

Controlling CSS Styles and Classes Directly

In addition to the appearance properties and style objects, controls expose two properties that allow you to manipulate CSS styles more directly: **CSSStyle** and **CSSClass**. The **CSSClass** property enables you to assign a stylesheet class to the control. The **CSSStyle** property enables you to set a string of style attributes to be written as-is to the control. Using the **CSSStyle** property allows you to set style attributes that are not exposed via other properties.

Settings you make in the **CSSStyle** property are not reflected in the corresponding individual appearance property. For example, if you set a string `background-color:red` in the **CSSStyle** property, the **BackColor** property is not also set to red, even though the control will render with a red background. Furthermore, if you set both appearance properties and the **CSSStyle** property, the individual appearance properties take precedence over the **CSSStyle** property.

See Also

Programming Web Forms | Customizing the Appearance of ASP.NET Server Controls Using Styles | ASP.NET Server Controls | ASP.NET Server Controls and Browser Capabilities

Customizing the Appearance of ASP.NET Server Controls Using Styles

Some server controls allow you to customize the appearance of individual elements by setting properties of style objects. For example, in the **Calendar** Web server control, you can set the appearance of the selected day by setting properties of the SelectedDayStyle object. Similarly, in the **DataList** Web server control, you can set the appearance of individual items and selected items using properties of the ItemStyle and SelectedItemStyle objects.

Setting Style Properties in the Web Forms Designer

The Web Forms Designer provides tools that make it easy to set style properties.

To set style properties in the Web Forms Designer

1. In Design view, select the control that you want to set styles for.
2. In the Properties window, do the following:

   - Expand the style object and then set properties for that object. For example, to set properties for the **DataList** control's **ItemStyle** object, expand **ItemStyle** in the Properties window.

     –or–

   - If the control supports property builders (such as the DataList and DataGrid controls), click **Property Builder** at the bottom of the Properties window to open the **Properties** dialog box. Choose the **Format** option, choose a style object in the **Objects** list, and then set its properties.

Setting Style Properties

You can set style properties directly in the .aspx file in two ways.

To set style properties in the .aspx file

- Declare the style property using a convention of hyphenating the style object name and property:

```
<asp:Calendar ID="MyCalendar"
   SelectionMode="DayWeek"
   runat="server"
   TitleStyle-Backcolor="#3366ff"
   TitleStyle-ForeColor="White" />
```

  -or-

- Declare explicit elements in which you declare property elements of the appropriate data type:

```
<asp:Calendar id="MyCalendar"
   SelectionMode="DayWeek"
   runat="server">
   <TitleStyle BackColor="#3366ff"
               ForeColor="white" />
</asp:Calendar>
```

Note If the Web Forms Designer cannot render a Web server control correctly, it displays a grey box with the text "Error Creating Control." This often means that the ASP.NET syntax of the control is incorrect — for example, if the
runat="server"
attribute is missing in a Web server control element, you will see this error. Point to the information icon ( ); a ToolTip is displayed with details about the error.

Setting Style Properties Programmatically

You can set style properties of a control programmatically, which allows you to change the appearance of a control conditionally.

To set style properties programmatically

- Use the following hierarchical convention for specifying the style object and property you want to set:

```
Control.StyleObject.Property = value
```

  The following example shows how you might set the **BackColor** property for the **DayStyle** object of a **Calendar** control:

```
' Visual Basic
Calendar1.DayStyle.BackColor = Color.Green

// C#
Calendar1.DayStyle.BackColor = Color.Green;
```

You can also create a separate Style object and copy or merge the values of its properties to one of the styles on a control. This is a powerful way to apply the same styles to several different list objects in your project. In effect, you can create a virtual stylesheet as a **Style** object and then apply it to a series of controls.

To create a Style object and apply it to controls

1. Create an instance of the **Style** object and set its properties:

```
' Visual Basic
Dim s As Style = New Style()
s.BackColor = Color.Red

// C#
Style s = new Style();
s.BackColor = Color.Red;
```

2. Assign the **Style** object to a control using one of the following methods:

   - The CopyFrom method applies all settings from a **Style** object, including null ones.
   - The MergeWith method copies only the properties already set on the **Style** object, skipping **Style** properties that have not been set. Also, the **MergeWith** method will not overwrite any existing style elements.

   The following example shows how you might create a **Style** object, set one of its properties, and then apply it to two different control style objects, using each of the methods described:

```
' Visual Basic
' Apply all values in s, including nulls.
Calendar1.SelectedDayStyle.CopyFrom(s)
' Apply only values in s that are null in HeaderStyle.
DataGrid1.HeaderStyle.MergeWith(s)

// C#
// Apply all values in s, including nulls.
Calendar1.SelectedDayStyle.CopyFrom(s);
// Apply only values in s that are null in HeaderStyle.
DataGrid1.HeaderStyle.MergeWith(s);
```

See Also

Programming Web Forms | ASP.NET Server Controls and CSS Styles | Web Server Controls Templates | ASP.NET Server Controls | ASP.NET Server Controls and Browser Capabilities

Web Server Controls Templates

Most Web server controls have a default look and layout, which you can manipulate by setting properties or by using styles. Some Web server controls also allow you to customize their look by using templates.

A template is a set of HTML elements and controls that make up the layout for a particular portion of a control. For example, in the **DataList** Web server control you can use a combination of HTML elements and controls to create the layout for each row of the list. Similarly, the **DataGrid** Web server control has a default look for rows in the grid. However, you can customize the grid's look by defining different templates for individual rows, alternating rows, selected rows, and so on.

> **Note** Templates differ from styles. A template defines the content of a section of a control — for example, the contents of a row in the **DataList** control. Styles, on the other hand, specify the appearance of elements such as color, font, and so on. Styles can apply to the control as a whole (for example, to set the font for the **DataGrid** control) as well as to template items.

Templates consist of HTML and even embedded server controls. When the control runs in the Web Forms page, the control framework renders the contents of the template in place of the default HTML for the control.

Which Controls Support Templates?

Not all Web server controls support templates. For the most part, templates are supported by complex controls. In Visual Studio, this includes the **DataGrid**, **DataList**, and **Repeater** Web server controls. (In fact, the **Repeater** and **DataList** controls not only support templates, but require that you create templates to define their output). Controls that you acquire from other sources might also support templates.

Each control supports a slightly different set of templates that specify layouts for different portions of the control, such as the header, footer, item, and selected item. You can specify a template for any or all of these, depending on which ones you want to customize. In the **DataGrid** control, you can specify templates for columns (rather than rows).

The following table summarizes the Web server controls that support templates:

Control	Templates
Repeater	• HeaderTemplate • FooterTemplate • ItemTemplate • AlternatingItemTemplate • SeparatorTemplate
DataList	• HeaderTemplate • FooterTemplate • ItemTemplate • AlternatingItemTemplate • SeparatorTemplate • SelectedItemTemplate • EditItemTemplate
DataGrid	• HeaderTemplate • FooterTemplate • ItemTemplate (column) • EditItemTemplate (column) • Pager

Creating Templates

The Web Forms Designer includes a facility that allows you to lay out templates in a WYSIWYG way. You open a special template-editing mode that presents the control's templates in Design view. You can type HTML and drag controls from the Toolbox directly to the template you are editing.

You can also create templates directly in the .aspx file. Templates are created as XML declarations. The following example shows how you can display a list of employee names, phone numbers, and email addresses using templates in the **DataList** Web server

control. The layout of the employee information is specified in the ItemTemplate using data-bound controls.

```
<asp:datalist ID="DataList1" runat="server">
   <HeaderTemplate>
      Employee List
   </HeaderTemplate>
   <ItemTemplate>
      <asp:label id=Label1 runat="server"
         Text='<%# DataBinder.Eval(Container, "DataItem.EmployeeName")
         %>'></asp:label>
      <asp:label id=Label2 runat="server"
         Text='<%# DataBinder.Eval(Container, "DataItem.PhoneNumber")
         %>'></asp:label>
      <asp:Hyperlink id=Hyperlink1 runat="server"
         Text='<%# DataBinder.Eval(Container, "DataItem.Email") %>'
         NavigateURL='<%# DataBinder.Eval(Container, "DataItem.Link") %>'>
      </asp:Hyperlink>
   </ItemTemplate>
</asp:datalist>
```

See Also

Creating Web Server Control Templates | Creating Web Server Control Templates Dynamically | ASP.NET Server Controls | ASP.NET Server Controls and CSS Styles | Web Forms Data Binding | Creating Web Server Control Templates Dynamically | Creating Templates Programmatically in the DataGrid Control

Creating Web Server Control Templates

The **DataList**, **DataGrid**, and **Repeater** Web server controls allow you to specify templates, which are HTML elements that define the layout for a particular portion of the control. For example, in the **DataGrid** control, you can define templates for items, selected items, alternating items, and so on, so that each of these elements can have a custom look.

> **Note** For background information about templates, see Web Server Controls Templates.

Creating Templates in the Web Forms Designer

You can create templates directly from within the Web Forms Designer.

To create a template in the Web Forms Designer

1. Right-click the control, choose **Edit Template**, and then choose the type of template that you want to edit.

   For example, to edit the **Item**, **AlternatingItem**, **SelectedItem**, and **EditItem** templates for the **DataList** Web server control, choose **Item Templates**.

   The Web Forms Designer displays the control in template-editing mode.

2. Enter HTML text and drag controls into the template.
3. Edit controls you have added to templates as you would normal controls:

   - Select them and use the Properties window to set their properties.
   - Data bind them using the **DataBinding** dialog box. For details, see Introduction to Data Binding in Web Forms Pages.
   - Double-click the embedded controls to create an **ItemCommand** event handler for the container control. Events raised by embedded controls are bubbled to the container. For more details, see ASP.NET Server Control Event Model.

4. Repeat Steps 1 through 3 to edit other templates.
5. To finish, right-click the control and choose **End Template Editing**.
6. Set the appearance of individual templates by setting their style properties.For details, see ASP.NET Server Controls and CSS Styles.

   > **Tip** You can quickly set the appearance of templates by selecting a predefined style scheme. Select the control and then choose **AutoFormat** at the bottom of the Properties window. When you select a scheme, the Designer sets style properties for templates.

Creating Templates Using ASP.NET Syntax

To create a Web server control template using ASP.NET syntax

1. In the .aspx file, insert an element inside the control to identify what template you are creating:

```
<asp:DataList id="DataList1" runat="server">
   <ItemTemplate>
   </ItemTemplate>
</asp:DataList>
```

2. Inside the template element, add HTML text and other controls as the template's content. Include property and data-binding values for the embedded controls using normal syntax:

```
<ItemTemplate>
   Name: <asp:Label runat="server"
          Text="<%# Container.DataListItem.Name %>"/>
</ItemTemplate>
```

3. Repeat Steps 1 and 2 for each template you want to create.

The following example shows a complete declaration for a **DataList** Web server control with simple templates declared for the **Header**, **Item**, and **Separator** templates:

```
<asp:datalist id="DataList1" runat=Server >
   <HeaderTemplate>
      Items matching your query:
   </HeaderTemplate>
   <ItemTemplate>
      Name: <asp:Label id=Label1 runat="server"
      Text='<%# DataBinder.Eval(Container, "DataItem.EmployeeName")
      %>'></asp:Label>
   </ItemTemplate>
   <SeparatorTemplate>
      <br><hr>
   </SeparatorTemplate>
</asp:datalist></p>
```

Note If the Web Forms Designer cannot render a Web server control correctly, it displays a grey box with the text "Error Creating Control." This often means that the ASP.NET syntax of the control is incorrect — for example, if the runat="server"
attribute is missing in a Web server control element, you will see this error. Point to the information icon ( ); a ToolTip is displayed with details about the error.

See Also

Web Server Controls Templates | ASP.NET Server Controls | ASP.NET Server Controls and CSS Styles | Creating Web Server Control Templates Dynamically | Creating Templates Programmatically in the DataGrid Control

Creating Web Server Control Templates Dynamically

For various reasons, you might not know until run time what templates you need or what text or controls should be in them. In that case, you need to be able to create the templates dynamically (in code).

> **Note** You can also create templates as Web Forms user controls and bind them dynamically to controls on your page. For details, see Creating a Templated User Control.

You can create templates in code for all controls that use templates: the **DataList**, **Repeater**, and **DataGrid** controls. For the **DataGrid** control, you use templates to define columns instead of the row-like templates for the other two controls.

> **Note** There are a few differences when creating template columns for the **DataGrid** control. For details, see Creating Templates Programmatically in the DataGrid Control.

Creating the Template Class

To create dynamic templates, you create a template class that you can then instantiate when needed.

> **Note** For background on creating classes in Visual Basic .NET, see Understanding Classes. For similar information for Visual C# .NET, see class.

To create a template class

1. Create a new class that implements the **ITemplate** interface of the System.Web.UI namespace.
2. Optionally, pass into the class's constructor a value that the class can use to determine what type of template to create (**ItemTemplate**, **AlternatingItemTemplate**, and so on).

   > **Tip** A type-safe way to pass the template type to the constructor is to add a parameter to the constructor with the type ListItemType. The **ListItemType** enumeration defines the possible template types for the **Repeater**, **DataList**, and **DataGrid** controls.

3. In the class, implement the **InstantiateIn** method (the only member of the **ITemplate** interface). This method provides a way to insert an instance of text and controls into the specified container.
4. In the **InstantiateIn** method, create the controls for the template item, set their properties, and then add them to the parent's **Controls** collection. You can access the parent control via the reference passed to the **InstantiateIn** method.

   > **Note** You cannot directly add static text to the **Controls** collection, but you can create controls like the **Literal** control or the **LiteralControl** control, set their **Text** properties, and then add those controls to the parent collection.

   The following example illustrates a complete template class that displays some static text ("Item number:") and a counter. The counter is maintained as a shared or static value (depending on what language you are using) called `itemcount` for the class and incremented each time a new item is created.

   The class defines an explicit constructor that accepts a **ListItemType** enumeration value to indicate what type of template is being created. Depending on what type of template is being created, the code creates different types of controls and adds them to the **Controls** collection of the parent control. The end result is an HTML table with a different background color for the alternating item template.

```
' Visual Basic
Private Class MyTemplate
   Implements ITemplate
   Shared itemcount As Integer = 0
   Dim TemplateType As ListItemType

   Sub New(ByVal type As ListItemType)
      TemplateType = type
   End Sub

   Sub InstantiateIn(ByVal container As Control) _
      Implements ITemplate.InstantiateIn
      Dim lc As New Literal()
```

```
      Select Case TemplateType
         Case ListItemType.Header
            lc.Text = "<TABLE border=1><TR><TH>Items</TH></TR>"
         Case ListItemType.Item
            lc.Text = "<TR><TD>Item number: " & itemcount.ToString _
               & "</TD></TR>"
         Case ListItemType.AlternatingItem
            lc.Text = "<TR><TD bgcolor=lightblue>Item number: " _
               & itemcount.ToString & "</TD></TR>"
         Case ListItemType.Footer
            lc.Text = "</TABLE>"
      End Select
      container.Controls.Add(lc)
      itemcount += 1
   End Sub
End Class

// C#
public class MyTemplate : ITemplate
{
   static int itemcount = 0;
   ListItemType templateType;

   public MyTemplate(ListItemType type)
   {
      templateType = type;
   }

   public void InstantiateIn(System.Web.UI.Control container)
   {
      Literal lc = new Literal();
      switch( templateType )
      {
         case ListItemType.Header:
            lc.Text = "<TABLE border=1><TR><TH>Items</TH></TR>";
            break;
         case ListItemType.Item:
            lc.Text = "<TR><TD>Item number: " + itemcount.ToString() +
               "</TD></TR>";
            break;
         case ListItemType.AlternatingItem:
            lc.Text = "<TR><TD bgcolor=lightblue>Item number: " +
               itemcount.ToString() + "</TD></TR>";
            break;
         case ListItemType.Footer:
            lc.Text = "</TABLE>";
            break;
      }
      container.Controls.Add(lc);
      itemcount += 1;
   }
}
```

Using the Dynamic Template

When you have a dynamic template available, you can instantiate it in code.

> **Note** To work with a dynamic template as a column in a **DataGrid** control, see Creating Templates Programmatically in the DataGrid Control.

To use a dynamic template

1. Create an instance of your dynamic template, passing it an item type value if appropriate.
2. Assign the instance to one of the template properties of the **Repeater** or **DataList** control: ItemTemplate, AlternatingItemTemplate, HeaderTemplate, and so on.

   The following example shows how to use the dynamic template with a **Repeater** control. In this example, the templates are instantiated while the page is being loaded and before the control is bound to its data source.

```
' Visual Basic
Private Sub Page_Load(ByVal sender As System.Object, _
ByVal e As System.EventArgs) Handles MyBase.Load
   Repeater1.HeaderTemplate = New MyTemplate(ListItemType.Header)
   Repeater1.ItemTemplate = New MyTemplate(ListItemType.Item)
   Repeater1.AlternatingItemTemplate = _
      New MyTemplate(ListItemType.AlternatingItem)
   Repeater1.FooterTemplate = New MyTemplate(ListItemType.Footer)
   SqlDataAdapter1.Fill(DsCategories1)
   Repeater1.DataBind()
End Sub

// C#
private void Page_Load(object sender, System.EventArgs e)
{
   Repeater1.HeaderTemplate = new MyTemplate(ListItemType.Header);
   Repeater1.ItemTemplate = new MyTemplate(ListItemType.Item);
   Repeater1.AlternatingItemTemplate =
      new MyTemplate(ListItemType.AlternatingItem);
   Repeater1.FooterTemplate = new MyTemplate(ListItemType.Footer);
   sqlDataAdapter1.Fill(dsCategories1);
   Repeater1.DataBind();
}
```

Adding Data Binding to Templates

There are various ways to get access to data from within a template class, depending on how you have created the class. A good way is one in which the page architecture itself implements data binding — when you add controls to the template, you also add a handler for their **DataBinding** event. This event will be raised after the template item has been created with all its controls, and it provides you with an opportunity to fetch data and use it in a control.

> **Note** You cannot embed a data-binding expression as a string when creating controls in the template, as you do when defining templates at design time, because data-binding expressions are converted into code at a stage of page processing that occurs before your template is created.

In the handler for the **DataBinding** event, you have an opportunity to manipulate the contents of the control. Typically (but not necessarily), you fetch data from somewhere and assign it to the control's **Text** property.

> **Note** For background on data binding in Web Forms pages, see Web Forms Data Binding.

To add data binding to a dynamic template, you must do the following:

- Add a data-binding event handler to the controls you create in the template.
- Create the handler that you are binding to. In the handler, get the data that you want to bind to and assign it to the appropriate property of the control being bound.

To add the data-binding event handler

- After creating a control in the dynamic template, use standard commands to bind the control's **DataBinding** event to a method that you will create later.

  > **Note** For details on how to add event handlers dynamically, see AddHandler and RemoveHandler (for Visual Basic) and Events Tutorial (for Visual C#).

The following example shows code from the template class illustrating how you can bind a newly created control to a method called `TemplateControl_DataBinding`.

```
' Visual Basic
Dim lc As New Literal()
   Case ListItemType.Item
   lc.Text = "<TR><TD>"
   AddHandler lc.DataBinding, AddressOf TemplateControl_DataBinding

// C#
Literal lc = new Literal();
case ListItemType.Item:
   lc.Text = "<TR><TD>";
   lc.DataBinding += new EventHandler(TemplateControl_DataBinding);
   break;
```

Note that in this example, the text you add to the literal control's **Text** property is different than in the previous example. It contains only the beginning of the table row and cell for the item template. You will complete the cell and row in the data-binding event handler.

The next step is to create the event handler that will be called when the control is data bound.

To create the handler for the DataBinding event

1. Create a method that is part of your template class as a peer of the class's other methods (such as **InstantiateIn**). The handler's name must match the name you used when binding the event earlier. The method should have the following signature:

```
' Visual Basic
Private Sub TemplateControl_DataBinding(ByVal sender As Object, _
   ByVal e As System.EventArgs)

// C#
private void TemplateControl_DataBinding(object sender,
   System.EventArgs e)
```

2. Get a reference to the **DataItem** object containing the data for the current template item, by doing the following:
   a. Get a reference to the template item. You create a variable to hold the reference and then assign it the value you get from your control's **NamingContainer** property.
   b. Use that reference to get the naming container's (the template item's) **DataItem** property.
   c. Extract the individual data element (data column, for example) from the **DataItem** object and use it to set a property of the control you are binding.

   The following code illustrates one way to perform data binding within a dynamic template. It shows a complete data-binding event handler for **Literal** controls being created in a template for a **Repeater** control.

```
' Visual Basic
Private Sub TemplateControl_DataBinding(ByVal sender As Object, _
   ByVal e As System.EventArgs)
   Dim lc As Literal
   lc = CType(sender, Literal)
   Dim container As RepeaterItem
   container = CType(lc.NamingContainer, RepeaterItem)
   lc.Text &= DataBinder.Eval(container.DataItem, "CategoryName")
   lc.Text &= "</TD></TR>"
End Sub

// C#
private void TemplateControl_DataBinding(object sender,
```

```
System.EventArgs e)
{
   Literal lc;
   lc = (Literal) sender;
   RepeaterItem container = (RepeaterItem) lc.NamingContainer;
   lc.Text += DataBinder.Eval(container.DataItem, "CategoryName");
   lc.Text += "</TD></TR>";
}
```

Note If you have multiple types of controls in your templates, you would need to create a different data-binding event handler for each of the control types.

See Also

Web Server Controls Templates | Creating Templates Programmatically in the DataGrid Control

Setting ASP.NET Server Control Properties

Setting a control's properties defines its appearance and behavior. This topic addresses how to set control properties at design time.

Setting Properties in the Web Forms Designer

The Web Forms Designer provides tools for setting control properties.

To set server controls properties in the designer

- In Design view, select the control, and then set properties in the Properties window.
- In HTML view, place the insertion point in the element tag of the control, and then set properties in the Properties window.

Setting Properties using ASP.NET Syntax

Control properties can be set as part of a control's declaration.

To set server controls properties using ASP.NET syntax

1. Switch to HTML view.
2. In the .aspx file, set the attribute of the control declaration corresponding to the property you want.

   The exact attribute you set depends on the control and the property. For information about the properties for a specific control, search for the name of the control class (for example, "Button class (System.Web.UI.WebControls)" in the Help index.

   > **Note** In Web server controls, the property names do not have the same names as HTML attributes; be sure you are using the correct property name.

   The following example sets the **MaxLength** property of an HtmlInputText control:

   ```
   <input id="MyTextBox" type=text runat=server maxlength=20 >
   ```

   The following example sets the **TextMode** property of a **TextBox** control:

   ```
   <asp:textbox text="Hi" runat=Server TextMode=Multiline />
   ```

See Also

Setting Web Server Control Properties Programmatically | Setting HTML Server Control Properties Programmatically | Customizing the Appearance of ASP.NET Server Controls Using Styles

Setting Web Server Control Properties Programmatically

You can set Web server control properties programmatically to change the control's appearance and behavior at run time. Properties for Web server controls are strongly typed, so the exact technique you use to set properties depends on what type of property you are setting. For example, setting a unit property such as width is different from setting a color property. The sections below provide details about setting different types of properties. For information about the properties for a specific control, search for the name of the control class (for example, "Button class (System.Web.UI.WebControls)" in the Help index.

> **Note** You set HTML server control properties slightly differently. For details, see Setting HTML Server Control Properties Programmatically.

Setting Properties Based on Simple Values or Enumerations

If a Web server control property's data type is a primitive — a **String**, **Boolean**, or numeric type — then you can set the property value by simply assigning it to the property. Similarly, if the property's values are defined in an enumeration class, you can simply assign the enumeration to the property.

To set a property value based on simple values

- Assign the value as a literal or variable, as in the following examples:

```
' Visual Basic
Label1.Text = "Hello"
DataGrid1.PageSize = 5

// C#
Label1.Text = "Hello";
DataGrid1.PageSize = 5;
```

To set a property value based on an enumeration

- Assign the value using one of the enumeration values. The framework can resolve the enumeration based on the property's type. The following illustrates setting a property using an enumeration:

```
' Visual Basic
' Uses TextBoxMode enumeration
Me.TextBox1.TextMode = TextBoxMode.SingleLine
' Uses ImageAlign enumeration
Me.Image1.ImageAlign = ImageAlign.Middle

// C#
// Uses TextBoxMode enumeration
this.TextBox1.TextMode = TextBoxMode.SingleLine;
// Uses ImageAlign enumeration
this.Image1.ImageAlign = ImageAlign.Middle;
```

Setting Unit Properties

Width, height, and similar properties are set in units. Units are implemented as objects (the Unit structure), which allows you to specify a value and the measurement unit in various ways.

To set unit-based properties

- Assign a reference to the **Unit** structure to your control. The following examples show variations on how to do this.

```
' Visual Basic
' Default is pixels.
TextBox1.Width = New Unit(100)
TextBox1.Width = New Unit(100, UnitType.Pixel)
TextBox1.Width = New Unit("100px")
' Centimeters.
```

```
TextBox1.Width = New Unit("2cm")
TextBox1.Width = New Unit(10, UnitType.Percentage)
TextBox1.Width = New Unit("10%")

// C#
// Default is pixels.
TextBox1.Width = new Unit(100);
TextBox1.Width = new Unit(100, UnitType.Pixel);
TextBox1.Width = new Unit("100px");
// Centimeters
TextBox1.Width = new Unit("2cm");
TextBox1.Width = new Unit(10, UnitType.Percentage);
TextBox1.Width = new Unit("10%");
```

Setting Color Properties

To set a property to a color (such as the **BackColor** property), you assign a reference to the **Color** object.

To set properties to a color

- Assign a reference to the **Color** structure to your control. To specify the color, call the **Color** object's **FromArgb** method, passing it a numeric value (RGB) or string name of the color. Alternatively, you can assign the color using a static method that references a predefined color name:

```
' Visual Basic
Button1.BackColor = Color.FromName("Red")
' White in RGB.
Button1.BackColor = Color.FromArgb(255, 255, 255)
Button1.BackColor = Color.Red
' HTML 4.0 Color
Button1.BackColor = Color.MediumSeaGreen

// C#
Button1.BackColor = Color.FromName("Red");
// White in RGB.
Button1.BackColor = Color.FromArgb(255, 255, 255);
Button1.BackColor = Color.Red;
// HTML 4.0 color.
Button1.BackColor = Color.MediumSeaGreen;
```

Setting Control Properties in Collections

The properties of some controls are not simple values or objects, but collections. For example, the individual values of a **ListBox** Web server control are implemented as a collection of **ListItem** objects.

To set a complex control property

- Instantiate the item you want to use, and then add it to the control's collection. The following example shows how to add a **ListItem** object to a list box by adding it to the list box's **Items** collection. In the first example, the item is explicitly created before being added. In the second example, items are created and added at the same time.

```
' Visual Basic
Dim li As ListItem = New ListItem("Item 1")
ListBox1.Items.Add(li)

' Create and add the items at the same time
ListBox1.Items.Add(New ListItem("Apples"))
ListBox1.Items.Add(New ListItem("Oranges"))
ListBox1.Items.Add(New ListItem("Lemons"))

// C#
```

```
// Create the items and then add them to the list.
ListItem li = new ListItem("Item 1");
ListBox1.Items.Add(li);

// Create and add the items at the same time.
ListBox1.Items.Add(new ListItem ("Apples"));
ListBox1.Items.Add(new ListItem ("Oranges"));
ListBox1.Items.Add(new ListItem ("Lemons"));
```

See Also

Customizing the Appearance of ASP.NET Server Controls Using Styles | Setting HTML Server Control Properties Programmatically | Setting ASP.NET Server Control Properties

Setting HTML Server Control Properties Programmatically

HTML server controls are of two slightly different types. The HTML elements most commonly used in forms are available as individual HTML server controls, such as HtmlInputText, HtmlInputButton, HtmlTable, and so on. These HTML server controls expose their own, control-specific properties that map directly to HTML attributes. However, any HTML element can be converted to a control. In that case, the element becomes an HtmlGenericControl with base class properties such as **TagName**, **Visible**, and **InnerHTML**.

To set properties of HTML server controls

- Get or set the property name as you would with any object. All properties are either strings or integers. The following examples illustrate this:

```
' Visual Basic
myAnchor.Href = "http://www.microsoft.com"
Text1.MaxLength = 20
Text1.Value = string.Format("{0:$###}", TotalCost)
Span1.InnerHTML = "You must enter a value for Email Address."

// C#
myAnchor.HRef = "http://www.microsoft.com";
Text1.MaxLength = 20;
Text1.Value = string.Format("{0:$####}", TotalCost);
Span1.InnerHtml = "You must enter a value for Email Address.";
```

All HTML server controls also support an **Attributes** collection, which gives you direct access to all the control's attributes. This is particularly useful for working with attributes that are not exposed as individual properties.

To work with control attributes directly

- Use the properties and methods of a control's **Attributes** collection, such as **Add**, **Remove**, **Clear**, and **Count**. The **Keys** property returns a collection containing the names of all the attributes in the control. The following examples show various ways to use the **Attributes** collection:

```
' Visual Basic
' Adds new attribute.
Text1.Attributes.Add("bgcolor", "red")
' Removes one attribute.
Text1.Attributes.Remove("maxlength")
' Removes all attributes, clearing all properties.
Text1.Attributes.Clear()
' Creates comma-delimited list of defined attributes
Dim strTemp As String = ""
Dim key As String
For Each key In Text1.Attributes.Keys
   strTemp &= Text1.Attributes(key) & ", "
Next

// C#
// Adds a new attribute.
Text1.Attributes.Add("bgcolor","red");
// Removes one attribute.
Text1.Attributes.Remove("maxlength");
// Removes all attributes, clearing all properties.
Text1.Attributes.Clear();
// Creates comma-delimited list of defined attributes
string strTemp = "";
foreach (string key in Text1.Attributes.Keys)
{
   strTemp += Text1.Attributes[key] + ", ";
```

```
}
```

See Also

Setting ASP.NET Server Control Properties | Setting Web Server Control Properties Programmatically

Specifying Whether Controls Post to the Server

By default, information and selections that the user specifies in a Web Forms page are not posted immediately to the server. Instead, the values are posted only when the entire form is posted, usually in response to clicking a button. However, you might find it useful at times to send user changes to the server immediately.

To post control values to the server immediately

- Set the control's **AutoPostBack** property to **true**.

You can also respond to user actions by creating client scripts. For example, you can create a client script to respond to the **Click** event of a button or check box. For more information, see Client Script in Web Forms Pages.

See Also

ASP.NET Server Control Event Model | Programming Web Forms

Web Forms Control Identification

Every control on a Web Forms page must be uniquely identifiable. Generally, you assign a value to a control's **ID** property to uniquely identify it. This value becomes the instance name of the control — that is, the name by which you can refer to the control in code. For example, if you set the ID property of a **TextBox** control to "Text1," then you can reference the control in code as `Text1`.

A number of data-bound controls, including the DataList, Repeater, and DataGrid controls, act as containers for other (child) controls. When these controls run, they generate multiple instances of the child control. For example, if you create a **DataList** template with a **Label** control in it, when the page runs, there are as many instances of that **Label** control in the page as there are records in the **DataList** control's data source.

> **Note** Controls that use templates, such as the **DataList** and **Repeater** controls, host template objects. For example, when the **DataList** control runs, it creates multiple instances of the **DataListItem** class. These template objects in turn contain individual controls such as labels, text boxes, buttons, and so on.

Because controls can be instantiated multiple times on the same page, and because you can reuse control names on different pages, the ASP.NET page framework provides mechanisms to ensure that controls on the page and in your application have unique identifiers. It also provides you with ways to find these individual controls so that you can manipulate them in your own code.

The Naming Container

Controls that can act as containers for other controls generate a *naming container*, or an ID namespace, for their child controls. By providing this naming container, controls can guarantee that ID attributes of their child controls are unique within the entire application. (Controls generate a naming container by implementing the INamingContainer interface.) When child controls are created at run time, the naming container is combined with the child control's ID property to create the value of the UniqueID property of each child control. The **UniqueID** property therefore becomes a fully-qualified identifier for a control, referencing its naming container as well the controls' individual ID value.

In the example from above, the multiple instances of the **Label** control are created within the naming container — the namespace — of the parent **DataList** control. The **UniqueID** property of each **Label** control will reflect this namespace, which will have a format something like `DataList1:_ctl:MyLabel`, `DataList1:_ct2:MyLabel`, and so on.

> **Note** As a rule, you should not write code that references controls using the value of the generated **UniqueID** property. You can treat the **UniqueID** property as a handle (for example, by passing it to a process), but you should not rely on it having a specific structure.

In addition to each container control providing a naming container for its child controls, the page itself also provides a naming container for all of its child controls. This creates a unique namespace within the application for all the controls on that page.

Using the NamingContainer Property

Child controls can reference their naming container via the NamingContainer property. This property returns an object of type Control that you can cast to the appropriate **DataList** control, **DataListItem** object, and so on.

Referencing the naming container is useful when you need access from a child control to a property of the container control. For example, in a handler for a child control's **DataBinding** event, you can access the **DataItem** object by getting it from the naming container.

> **Note** The **NamingContainer** property does not necessarily reference the same control as the **Parent** property. For example, in a **Repeater** control, you might have an item template containing a table that in turn contains a **Label** control. The parent control of the label is a table cell (for example, a **HtmlTableCell** object), but its naming container is the **DataListItem** object, because it is the **DataListItem** that defines the namespace for the **Label** control, not the table.

Referencing Controls

If your page contains controls that are generated at run time, such as those in a template for the **DataList**, **Repeater**, or **DataGrid** controls, you cannot directly reference them by their ID, because the ID is not unique. However, there are various ways to find individual controls in the page. For details, see Referencing Controls in Web Forms Pages.

See Also

Referencing Controls in Web Forms Pages

When you are working with controls in templates, you cannot always directly reference the controls in code. For example, if you add a **Label** control to a **DataList** control and name it "Label1," you cannot get or set its properties at run time by simply referring to `Label1` in code. The page class does not contain an instance of the control that you can use directly, because there are typically multiple instances of the control. For more information, see Web Forms Control Identification.

Similarly, there are times when you are working with a control and need access to properties or methods of its naming container.

The sections below describe various ways in which you can get access to controls when you are working with naming containers and child controls.

Locating a Child Control by ID

You can get a reference to a specific control using a method that searches its naming container by its assigned ID.

To locate a control by ID

- Call the FindControl method of the naming container, passing it a string containing the ID of the control you want to use. The method returns an object of type Control that you can cast to the appropriate type.

  The following example shows how you can locate a specific control. The sample is a handler for the **Click** event of a button in a **DataGrid** control. When the button is clicked, the code searches for a label named `Label1` in the current **DataGrid** item (which is the **Label** control's naming container). If the label is found, its text is displayed in a second label (`LabelText`) elsewhere on the page.

```
' Visual Basic
Private Sub DataGrid1_ItemCommand(ByVal source As Object, _
   ByVal e As System.Web.UI.WebControls.DataGridCommandEventArgs) _
   Handles DataGrid1.ItemCommand
   Dim l As Label
   l = CType(e.Item.FindControl("Label1"), Label)
   If (Not l Is Nothing) Then
      LabelText.Text = l.Text
   End If
End Sub

// C#
private void DataGrid1_ItemCommand(object source,
   System.Web.UI.WebControls.DataGridCommandEventArgs e)
{
   Label l;
   l = (Label) e.Item.FindControl("Label1");

   if(!(l == null) ){
      LabelText.Text = l.Text;
   }
}
```

Accessing Members of the Naming Container

At times, you need to access properties or methods of a control's naming container. For example, during data binding, the container control makes available a **DataItem** property containing the data to which controls are bound. You can access the containing control in different ways, depending on context.

In Declarative Data-Binding Expressions

In data-binding expressions, you can access the container control using the special keyword `Container`. (For information on these expressions, see Data-Binding Expressions for Web Forms Pages.)

To access the naming container from a data-binding expression

- In the data-binding expression, use the **Container** keyword, which returns a reference to the container. You can then access the container's properties or methods.

  This keyword is used most commonly in the **DataBinder.Eval** method to get values from the naming container's **DataItem** object, but can be used outside of that method as well. The following example shows a **Label** control that might be in a template for a **DataList**, **Repeater**, or **DataGrid** control. It displays the current item number.

```
<asp:Label id=Label1 runat="server"
   Text='<%# Container.ItemIndex + 1  %>'>
</asp:Label>
```

  The following example is similar, but gets a value from the naming container's **DataItem** object:

```
<asp:Label id=Label1 runat="server"
   Text='<%# Container.DataItem("title") %>'>
</asp:Label>
```

In Code

In the code for a page, you can access a control's naming container by getting its **NamingContainer** property.

> **Note** The **NamingContainer** property does not necessarily reference the same control as the **Parent** property. For example, in a **Repeater** control, you might have an item template containing a table that in turn contains a **Label** control. The parent control of the label is a table cell (for example, a **HtmlTableCell** object), but its naming container is the **DataListItem** object, because it is the **DataListItem** that defines the namespace for the **Label** control, not the table.

To access the naming container from code

- Get the control's **NamingContainer** property and cast it to the container's class type, such as DataGridItem.

  For an example of using the **NamingContainer** property to get information about an individual control's naming container, see the section "Selecting Multiple Items Using a Check Box (Hotmail Model)" in the article Top Questions about the DataGrid Web Server Control.

Locating a Control by Walking the Controls Collection

Every container control on the page, and the page itself, has a **Controls** collection that you can use to get to individual controls.

To locate controls in the Controls collection

- Loop through the **Controls** collection of the container control. The collection is of type ControlCollection, and returns objects of type Control.

  The following example illustrates how to walk the **Controls** collection. The example assumes a Web Forms page with at least one **Textbox** control on it, a **Label** control, and a button. The code gets all the child controls of the **Page** object. Because this would produce only a few high-level child controls, including the **HtmlForm** object, the code also walks the **Controls** collection of each individual child control. The code looks for text boxes by comparing the type of each control. When it finds a text box, it gets that control's value and concatenates it into a string that is displayed at the end in a **Label** control.

  This example finds only the controls contained in the **Page** object and the controls that are direct children of the page. It does not find text boxes that are children of a control that is in turn a child of the page. For example, if you dragged a **Panel** control on the designer, the **Panel** control would be a child of the **HtmlForm** control contained by the **Page**, and it would be found in this example. However, if you then dragged a **TextBox** control onto the **Panel** control, the **TextBox** control text would not be displayed by the example, because it is not a child of the page or of a control that is a child of the page. A more practical application of walking the controls this way would be to create a recursive method that can be called to walk the **Controls** collection of each control as it is encountered. However, for clarity, the example below is not created as a recursive function.

```
' Visual Basic
Private Sub Button1_Click(ByVal sender As System.Object, _
```

```
    ByVal e As System.EventArgs) Handles Button1.Click
    Dim allTextBoxValues As String = ""
    Dim c As Control
    Dim childc As Control
    For Each c In Page.Controls
       For Each childc In c.Controls
          If TypeOf childc Is TextBox Then
             allTextBoxValues &= CType(childc, TextBox).Text & ","
          End If
       Next
    Next
    If allTextBoxValues <> "" Then
       Label1.Text = allTextBoxValues
    End If
End Sub

// C#
private void Button1_Click(object sender, System.EventArgs e)
{
   string allTextBoxValues = "";
   foreach (Control c in Page.Controls)
   {
      foreach (Control childc in c.Controls)
      {
         if (childc is TextBox)
         {
            allTextBoxValues += ((TextBox)childc).Text + ",";
         }
      }
   }
   if(allTextBoxValues != "")
   {
      Label1.Text = allTextBoxValues;
   }
}
```

See Also

Web Forms Control Identification

Web Forms Validation

An important aspect of creating Web Forms pages for user input is to be able to check that the information users enter is valid. ASP.NET provides a set of validation controls that provide an easy-to-use but powerful way to check for errors and, if necessary, display messages to the user.

The following topics tell you more about validation controls and how to use them.

In This Section

Introduction to Validating User Input in Web Forms
Provides background information about using validation controls.
Types of Validation for ASP.NET Server Controls
Lists which validation controls are available and what types of validation you can perform with them.
Client-Side Validation for ASP.NET Server Controls
Explains how validation works in browsers that support Dynamic HTML (such as Internet Explorer 4.0 or later).
Special-Case Validation Results for ASP.NET Server Controls
Describes how validation controls work under specific circumstances and how they interact with one another.
Validation Error Message Layout for ASP.NET Server Controls
Describes how error message display affects the layout of your page.
Validating Required Entries for ASP.NET Server Controls
Gives directions for making sure users do not skip a field.
Validating Against a Specific Value for ASP.NET Server Controls
Gives directions for making sure users enter only one of a restricted set of values.
Validating Against a Data Type for ASP.NET Server Controls
Gives directions for making sure users enter only a date, a number, or some other specific data format.
Validating Against Patterns for ASP.NET Server Controls
Gives directions for checking that a user's entry follows a specific sequence of numbers, letters, or other characters (such as a phone number, postal code, e-mail address, and so on).
Validating Against a Range of Values for ASP.NET Server Controls
Gives directions for checking that a user's entry falls between a specified low and high value.
Validating Against Values in a Database for ASP.NET Server Controls
Gives directions for checking if a user's entry is contained in a table of acceptable or correct values.
Validating with a Custom Function for ASP.NET Server Controls
Gives directions for creating your own validation logic and tying it to a validation control.
Testing Validity Programmatically for ASP.NET Server Controls
Gives directions for testing whether the user's entries passed validation checking.
Controlling Validation Error Message Display for ASP.NET Server Controls
Gives directions for specifying where error messages are displayed.
Formatting Validation Error Messages for ASP.NET Server Controls
Gives directions for setting font, color, and other attributes of error messages.
Specifying Layout for In-Place Messages On ASP.NET Server Controls
Gives directions for controlling page layout with error-message display.
Disabling Validation for ASP.NET Server Controls
Gives directions for turning off validation in code.
Displaying Custom Error Messages for ASP.NET Server Controls
Gives directions for creating your own error display.
Validating Programmatically for ASP.NET Server Controls
Gives directions for calling validation controls from your own code.
Walkthrough: Validating User Input in a Web Forms Page
Gives step-by-step details on validating user input in a Web Forms page.

Related Sections

Introduction to ASP.NET Server Controls
Describes the two general types of server controls and how they differ.
Introduction to ASP.NET Web Applications in Visual Studio
Provides an overview of the components of an ASP.NET Web application, including Web Forms pages and XML Web services, and how Visual Studio helps you create a Web application.
Introduction to Web Forms Pages
Presents a high-level overview of Web Forms pages and describes why Web Forms pages solve problems unique to Web

application development.

Walkthrough: Validating User Input in a Web Forms Page

Describes how to use Web Forms validation controls to check user entries without code, including required entries, data types, ranges, patterns, and specific values.

Introduction to Validating User Input in Web Forms

You can add input validation to Web Forms pages by using validation controls. Validation controls provide an easy-to-use mechanism for all common types of standard validation — for example, testing for valid dates or values within a range — plus ways to provide custom-written validation. In addition, validation controls allow you to completely customize how error information is displayed to the user.

Validation controls can be used with any controls that are processed in a Web Forms page's class file, including both HTML and Web server controls. For more information, see Introduction to ASP.NET Server Controls.

> **Security Note** By default, the Web Forms page automatically validates that malicious users are not attempting to send script to your application. For more information, see Scripting Exploits.

Using Validation Controls

You enable validation of user input by adding validation controls to your form as you would other server controls. There are controls for different types of validation, such as range checking or pattern matching. A complete list appears in the topic Types of Validation for ASP.NET Server Controls.

Each validation control references an input control (a server control) elsewhere on the page. When the user's input is being processed (for example, when the form is submitted), the ASP.NET page framework passes the user's entry to the appropriate validation control or controls. The validation controls test the user's input and set a property to indicate whether the entry passed the test. After all validation controls have been called, a property on the page is set; if any of the controls shows that a validation check failed, the entire page is set to invalid.

You can test the state of the page and of individual controls in your own code. For example, you would test the state of the validation controls before updating a data record with information entered by the user. If you detect an invalid state, you bypass the update. Typically, if any validation checks fail, you skip all your own processing and return the page to the user. Validation controls that detect errors then produce an error message that appears on the page.

When Validation Occurs

Validation occurs after page initialization (that is, after view-state and postback data have been processed) but before Change or Click event handlers are called.

You can perform the validation procedure earlier by calling the control's **Validate** method, for example during page load. For more details, see Validating Programmatically for ASP.NET Server Controls.

Validating for Multiple Conditions

Generally speaking, each validation control performs just one test. However, frequently you want to check for multiple conditions. For example, you might want to establish that a field is required and that it is limited to accepting dates within a specific range.

You can attach more than one validation control to an input field on a form. In that case, the tests performed by the controls are resolved using a logical AND. The data input by the user must pass all the tests in order to be considered valid.

In some instances, entries in several different formats might be valid. For example, if you are prompting for a phone number, you might allow users to enter a local number, a long-distance number, or an international number. This situation arises primarily when checking for specific patterns of numbers or characters. To perform this type of test — a logical OR — you use the pattern-matching validation control and specify multiple valid patterns within the control. Using multiple validation controls would not work in this instance because they are tested with a logical AND.

Displaying Error Information

Validation controls are normally not visible in the rendered form. However, if the control detects an error, it produces error message text that you specify. The error message can be displayed in a variety of ways, as listed in the following table.

Display option	Description
In place	Each validation control can individually display an error message or glyph in place (usually next to the control where the error occurred).
Summary	Validation errors can be collected and displayed in one place — for example, at the top of the page. This strategy is often used in combination with displaying a glyph next to the input fields with errors. If the user is working in Internet Explorer 4.0 or later, the summary can be displayed in a message box.

In place and Summary	The error message can be different in the summary and in place. You can use this option to show a shorter error message in place with more detail in the summary, or to show an error glyph next to the input field and an error message in the summary.
Custom	You can create custom error display by capturing the error information and designing your own output.

If you use the in-place or summary display options, you can format the error message text using HTML.

> **Security Note** If you create custom error messages, be sure you do not display information that might help a malicious user compromise your application. For details, see Displaying Safe Error Messages.

Server-Side and Client-Side Validation

Validation controls perform input checking in server code. When the user submits a form to the server, the validation controls are invoked to review the user's input, control by control. If an error has occurred in any of the input controls, the page itself is set to an invalid state so you can test for validity before your code runs.

If the user is working with a browser that supports DHTML, the validation controls can also perform validation using client script. This can substantially improve response time in the page; errors are detected immediately and error messages are displayed as soon as the user leaves the control containing the error. If client-side validation is available, you have greater control over the layout of error messages and can display an error summary in a message box. For more details, see Client-Side Validation for ASP.NET Server Controls.

The page framework performs validation on the server even if the validation controls have already performed it on the client, so that you can test for validity within your server-based event handlers. In addition, it helps prevent users from being able to bypass validation by disabling or changing the client script.

Validation Object Model

You can interact with validation controls using the object model exposed by individual validation controls and by the page. Each validation control exposes its own **IsValid** property that you can test to see if a validation test has passed or failed for that control.

At the page level you have access to another **IsValid** page property that summarizes the **IsValid** state of all the validation controls on the page. This property makes it easy to do a quick test to see if you can safely proceed with your own processing.

The page also exposes a **Validators** collection containing a list of all the validation controls on the page. You can loop through this collection to examine the state of individual validation controls.

> **Note** There are a few differences in the object model for client-side validation. For details, see Client-Side Validation for ASP.NET Server Controls.

Custom Validation

You can customize the validation process in a number of ways:

- You have complete control over the format, text, and location of error messages, and you can specify whether they appear individually or in summary form.
- You can create custom validation using a special control. The control calls your logic but otherwise works the same as other validation controls in setting error state, displaying error messages, and so on. This provides an easy way to create custom validation logic but still participate in the validation framework of the page.
- If you have specified client-side validation, you can intercept the validation call and substitute or add your own validation logic.

Validating Programmatically

By default, validation controls perform their checks when the Click events are processed. However, you can perform validation at any time in your code by calling a validation control's **Validate** method. Calling this method allows you to evaluate the validity of property settings made programmatically. For example, you might gather data from a user by a method other than a Web Forms page, and then set properties of server controls programmatically.

For more details, see Validating Programmatically for ASP.NET Server Controls.

See Also

Web Forms Validation | Walkthrough: Validating User Input in a Web Forms Page

Types of Validation for ASP.NET Server Controls

The following table lists the types of validation controls available and how you can use them.

> **Security Note** By default, the Web Forms page automatically validates that malicious users are not attempting to send script to your application. For more information, see Scripting Exploits.

Type of validation	Control to use	Description
Required entry	**RequiredFieldValidator**	Ensures that the user does not skip an entry.
Comparison to a value	**CompareValidator**	Compares a user's entry against a constant value, or against a property value of another control, using a comparison operator (less than, equal, greater than, and so on).
Range checking	**RangeValidator**	Checks that a user's entry is between specified lower and upper boundaries. You can check ranges within pairs of numbers, alphabetic characters, and dates.
Pattern matching	**RegularExpressionValidator**	Checks that the entry matches a pattern defined by a regular expression. This type of validation allows you to check for predictable sequences of characters, such as those in social security numbers, e-mail addresses, telephone numbers, postal codes, and so on.
User-defined	**CustomValidator**	Checks the user's entry using validation logic that you write yourself. This type of validation allows you to check for values derived at run time.

You can attach more than one validation control to an input control. For example, you might specify that a control is required, and that it contain a specific range of values.

A related control, the ValidationSummary control, does not perform validation, but it is often used in conjunction with the controls listed above. The purpose of the ValidationSummary control is to display the error messages from all the validation controls on the page together, in a single location. For more information, see Controlling Validation Error Message Display for ASP.NET Server Controls.

See Also

Web Forms Validation | Introduction to Validating User Input in Web Forms

Client-Side Validation for ASP.NET Server Controls

If the user is working with Internet Explorer 4.0 or later, validation controls can perform validation using client script. Because the controls can provide immediate feedback (without a round trip to the server), the user experience with the page is enhanced.

Under most circumstances, you do not have to make any changes to your page or to the validation controls to use client-side validation. The controls automatically detect if the browser supports DHTML and perform their checking accordingly. Client-side validation uses the same error display mechanism as server-side checking.

> **Security Note** Validation is performed on the server even if it was already performed on the client. This allows you to check validation status in server code and provides security against users bypassing validation.

Differences in Client-Side Validation

There are some additional features of validation available if it is performed on the client:

- If you are summarizing validation error messages, you can display them in a message box that appears when the user submits the page. For details, see Controlling Validation Error Message Display for ASP.NET Server Controls.
- The object model for validation controls is slightly different on the client. See "Client Validation Object Model" below.

There are a few minor differences associated with client-side validation:

- If client-side validation is enabled, the page includes references to script libraries that are used to perform the client-side validation.
- When you use a **RegularExpressionValidator** control, the expressions can be checked on the client if an ECMAScript-compatible language (such as Microsoft JScript) is available. This differs in some very small details from the regular expression checking done on the server using the System.Text.RegularExpressions.Regex class.
- The page includes a client-side method to intercept and handle the **OnClick** event before the page is submitted.

Client Validation Object Model

Validation controls present almost the same object model on the client as on the server. For example, you can test validation by reading the value of a validation control's **IsValid** property the same way on both the client and the server.

However, there are differences in the validation information exposed at the page level. On the server, the page supports properties; on the client, it contains global variables. The following table compares the information exposed on the page.

Client Page Variable	Server Page Property
Page_IsValid	**IsValid**
Page_Validators (array) Contains references to all validation controls on the page.	**Validators** (collection) Contains references to all validation controls.
Page_ValidationActive A Boolean value that indicates whether validation should take place. Set this variable to **False** to turn off client-side validation programmatically.	(no equivalent)

> **Note** All page-related validation information should be considered read-only.

Posting with Client-Side Validation Errors

Client-side validation is enabled automatically if you are using Microsoft Internet Explorer 4.0 or later. In some instances, however, you might prefer to use only server-side validation, even if client-side validation is available. For example, client-side validation might not be possible if the validation requires information or resources that are only available on the server, such as access to a database.

By default, when client-side validation is being performed, the user cannot post the form to the server if there are errors on the page. However, you might find it necessary to allow the user to post even with errors. For example, you may have a "Cancel" button or a navigation button on a page, and you want the button to submit the page even if some controls would fail validation. For more information, see Disabling Validation for ASP.NET Server Controls.

> **Note** Client-side validation requires that your browser support ECMAScript 1.2 (JScript) or later, and a Document Object Model (DOM) compatible with Internet Explorer 4.0 or later.

See Also

Special-Case Validation Results for ASP.NET Server Controls

Validation ensures that what the user has entered meets criteria that you specify. In most cases, this is a straightforward operation. However, at times the situations under which input is validated introduce special conditions. It is important that you understand what those conditions are, how validation is performed, and what the results of a validation attempt can be.

Comparing Against Blank Fields

All validation controls except the **RequiredFieldValidator** control consider a field valid if it is blank. You must use the **RequiredFieldValidator** control to prevent a user from leaving a field blank. For example, if you attach a **RangeValidator** control to a date field, validation passes if the user enters a date in the specified range, or if the field is left blank. If you want to ensure that the user enters a date, you must add a **RequiredFieldValidator** control to the date field along with the range check.

Comparing Against Another Control

When using the **CompareValidator** control, you can specify that validation be performed against the value of another control. For example, you might have two fields for entering dates, a start date and an end date. Using a **CompareValidator** control, you can ensure that the start date is less than or equal to the end date.

> **Security Note** User input in a Web Forms page can include potentially malicious client script. By default, the Web Forms page validates that user input does not include script or HTML elements. For more information, see Scripting Exploits and Protecting Against Script Exploits in a Web Application.

If the validation controls are unable to resolve the value in the other control, they do not raise an error; instead, the validation check passes. This occurs if the value in the other control is missing or cannot be converted to the data type of the **CompareValidator** control.

The specific tests performed by these controls, and possible results, are:

- If the target control entered in the originating control's **ControlToValidate** property does not have a value, **IsValid** = **true** and validation passes.
- If the value of the target control in the **ControlToValidate** property cannot be converted to the appropriate data type, **IsValid** = **false**.
- If the value of the target control in the **ControlToCompare** property cannot be converted to the appropriate data type, then **IsValid** = **true**.
- Finally, if all of these tests have passed, then both controls are non-empty and can be converted to the data type, so the control performs the actual comparison and sets **IsValid** appropriately.

This strategy prevents multiple errors from being reported for the same invalid entry. Validation controls are designed to test and report on only one value and to be able to display as precise a message as possible when an error is encountered.

For example, in the example of the end date, imagine that the user enters an invalid start date. The user then enters an end date. When the validation check is performed (usually on the server), the same bad date could potentially raise an error both for the start date field and for the end date. To prevent this from occurring, the end-date validation — the validation dependent on another field — does not raise an error.

You should therefore ensure that you provide sufficient validation for each individual field. In the example, it would be up to you to make sure that the user has entered a proper date in the start date field. In many cases, you should include a **RequiredFieldValidator** control in addition to a validation control that tests the contents of a specific entry. It is also useful in some cases to add a **CompareValidator** control with **Operator**=**DataTypeCheck**, which does a simple type check without doing a comparison to another value or control.

See Also

Web Forms Validation | Validating Against a Data Type for ASP.NET Server Controls |
Validating Against Patterns for ASP.NET Server Controls | Validating Against a Range of Values for ASP.NET Server Controls

Validation Error Message Layout for ASP.NET Server Controls

When an error message appears on the page, it becomes part of the layout of the page. You need to design the layout of your page to accommodate error text that might appear.

If validation is being performed on the server, validation controls are not rendered to the page unless an error occurs and therefore take up no space whatsoever. If an error has occurred, the page is recreated during the round trip and the error text is added to the page.

If the user is using a DHTML-compatible browser, validation occurs dynamically on the client. In that case, in-place error message text appears immediately and can potentially change the layout of the page.

To control how error message text is displayed with client validation, you can set the Display property of validation controls to one of three options:

Layout Option	**Description**
Static layout	Each validation control takes up space even when no error message text is visible, allowing you to define a fixed layout for the page. Validation controls cannot occupy the same space on the page, so you must give them each separate locations on the page. This setting only works in Internet Explorer 4.0 or later; in other browsers the layout effectively becomes Dynamic.
Dynamic layout	Validation controls take up no space unless they are displaying an error message, which allows them to share the same location (a table cell, for example). However, when the error message is displayed, the layout of the page changes, sometimes causing controls to change positions.
None	The validation control does not appear on the page.

> **Note** Dynamic layout requires Internet Explorer 4.0 or later. If your Web application will be used with other browsers, it is advisable instead to lay out the validation controls in a table cell or a panel control that is large enough to accommodate the longest possible error message. (For more information see Panel Web Server Control.)

See Also

Web Forms Validation | Specifying Layout for In-Place Messages On ASP.NET Server Controls

Validating Required Entries for ASP.NET Server Controls

You can specify that a user must provide information in a specific control by adding a **RequiredFieldValidator** control to the page and linking it to the required control. For example, you can specify that users must fill in a Name field before they can submit a registration form.

If validation is being performed on the client, the user can leave a required field blank while working in the page, but must fill it in before submitting. However, once a value has been entered in the field, the user cannot clear the field. If the field is cleared, the user sees an error message immediately when leaving the field. In server-side validation, no checking is done until the form is submitted, so the user will not see an error message until after the form is submitted.

> **Tip** Required-entry validation is frequently used in conjunction with other types of validation. You can use as many validation controls for a user-entry field as needed.

To validate a required entry

1. Add a **RequiredFieldValidator** control to the page and set the following properties:

Property	Description
ControlToValidate	The ID of the control for which the user must provide a value.
ErrorMessage, **Text**, **Display**	Properties that specify the text and location of the error or errors that will appear if the user skips the control. For details, see Controlling Validation Error Message Display for ASP.NET Server Controls.

2. Add a test in your Web Forms code to check for validity. For details, see Testing Validity Programmatically for ASP.NET Server Controls.

The following example shows the .aspx file of a Web Forms **TextBox** control with required field validation.

```
<asp:Textbox id="txtLastName" runat="server"></asp:Textbox>
<asp:RequiredFieldValidator id="RequiredFieldValidator1" runat="server"
   ControlToValidate="txtLastName"
   ErrorMessage="Last name is a required field."
   ForeColor="Red">
</asp:RequiredFieldValidator>
```

See Also

Introduction to Validating User Input in Web Forms | Controlling Validation Error Message Display for ASP.NET Server Controls

Validating Against a Specific Value for ASP.NET Server Controls

You can validate a user's entry against a specific single value using logical operators. For example, you can specify that the user's entry is a date after January 1, 1950 or that it is an integer value greater than or equal to zero. Alternatively, you can specify that the user's entry be compared against a value from another control.

> **Security Note** User input in a Web Forms page can include potentially malicious client script. By default, the Web Forms page validates that user input does not include script or HTML elements. For more information, see Scripting Exploits and Protecting Against Script Exploits in a Web Application.

To validate against a specific value

1. Add a **CompareValidator** control to the page and set the following properties:

Property	Description
ControlToValidate	The ID of the control to validate.
ErrorMessage, **Text**, **Display**	Properties that specify the text and location of the error or errors that will appear if the validation fails. For details, see Controlling Validation Error Message Display for ASP.NET Server Controls.

2. Set the value to compare to by setting the following properties:

Property	Description
ValueToCompare - or - **ControlToCompare**	An expression entered as a string. To compare to a constant value, set the **ValueToCompare** property. To compare against the value of another control, set the **ControlToCompare** property to the ID of that control (the **CompareValidator** control compares the user's entry against whatever property is specified by the other control's ValidationPropertyAttribute). If you set both **ValueToCompare** and **ControlToCompare**, **ControlToCompare** takes precedence.
Type	The data type of the two values to be compared. Types are specified using the **ValidationDataType** enumeration, which allows you to use the type name **String**, **Integer**, **Double**, **Date**, or **Currency**. The values are converted to this type before the comparison is performed.
Operator	The comparison to use. Operators are specified using the **ValidationCompareOperator** enumeration, which allows you to enter the name of the comparison operators, such as **Equal**, **NotEqual**, **GreaterThan**, **GreaterThanEqual**, and so on.

> **Note** When validation is done against another control, invalid values in the other control are ignored and the validation passes. For details, see Special-Case Validation Results for ASP.NET Server Controls.

3. Add a test in your Web Forms code to check for validity. For details, see Testing Validity Programmatically for ASP.NET Server Controls.

The following example shows the .aspx file of a Web Forms **Textbox** control with a **CompareValidator** control. A table is used to handle layout.

```
<TABLE>
<TR>
<TD>
   <asp:Textbox id="txtAge" runat="server"></asp:Textbox>
</TD>
<TD>
   <asp:CompareValidator id="CompareFieldValidator1" runat="server"
      ForeColor="Red"
      ControlToValidate="txtAge"
      ValueToCompare=0
      Type="Integer"
```

```
        Operator="GreaterThanEqual"
        ErrorMessage="Please enter a whole number zero or greater.">
    </asp:CompareValidator >
</TD>
</TR>
</TABLE>
```

The following example shows a **CompareValidator** that compares the user's entry against the value in another control. In a form that allows users to make reservations at a hotel, the validator checks that the user does not enter a departure date earlier than the arrival date. (In a real application, the departure date would be required and validated as a date as well.)

```
<TABLE>
<TR>
<TD>
   Arrive<asp:Textbox id=txtArrivalDate runat="server"></asp:Textbox>
   Depart<asp:Textbox id=txtDepartureDate runat="server"></asp:Textbox>
</TD>
<TD>
   <asp:CompareValidator id="CompareFieldValidator1" runat="server"
         ForeColor="Red"
         ControlToValidate="txtDepartureDate"
         ControlToCompare="txtArrivalDate"
         Type="Date"
         Operator="GreaterThanEqual"
         ErrorMessage="Departure date cannot be earlier than arrival date.">
   </asp:CompareValidator >
</TD>
</TR>
</TABLE>
```

See Also

Introduction to Validating User Input in Web Forms | Controlling Validation Error Message Display for ASP.NET Server Controls | Validating Required Entries for ASP.NET Server Controls | Validating Against a Data Type for ASP.NET Server Controls

Validating Against a Data Type for ASP.NET Server Controls

You can validate a user's entry against a specific data type to be sure that what the user has entered is a number, or a date, and so on. For example, if you want to collect birth date information on a user registration page, you can use a **CompareValidator** control to make sure that the date is in a recognized format before it is submitted to your database.

To validate against a data type

1. Add a **CompareValidator** control to the page and set the following properties:

Property	Description
ControlToValidate	The ID of the control for which you are checking the data type.
ErrorMessage, **Text**, **Display**	Properties that specify the text and location of the error or errors that will display if the validation fails. For details, see Controlling Validation Error Message Display for ASP.NET Server Controls.

2. Set the data type to compare to by setting the following properties:

Property	Description
Type	The data type to check against. The type is specified using the **ValidationDataType** enumeration, which allows you to use the type names **String**, **Integer**, **Double**, **Date**, or **Currency**.
Operator	**DataTypeCheck**.

   Note If the user leaves a control blank, the control passes the comparison validation. To force the user to enter a value, add a **RequiredFieldValidator** control as well. For details, see Validating Required Entries for ASP.NET Server Controls.

3. Add a test in your Web Forms code to check for validity. For details, see Testing Validity Programmatically for ASP.NET Server Controls.

See Also

Introduction to Validating User Input in Web Forms | Controlling Validation Error Message Display for ASP.NET Server Controls

Validating Against Patterns for ASP.NET Server Controls

You can check that a user's entry matches a predefined pattern, such as a phone number, postal code, e-mail address, and so on. To do so, you use a regular expression. For more information on regular expressions, see .NET Framework Regular Expressions.

> **Security Note** By default, the Web Forms page automatically validates that malicious users are not attempting to send script or HTML elements to your application. As long as this validation is enabled, you do not need to explicitly check for script or HTML elements in user input. For more information, see Scripting Exploits.

To validate against a regular expression

1. Add a **RegularExpressionValidator** control to the page and set the following properties:

Property	Description
ControlToValidate	The ID of the control you are validating.
ErrorMessage, **Text**, **Display**	Properties that specify the text and location of the error or errors that will display if the validation fails. For details, see Controlling Validation Error Message Display for ASP.NET Server Controls.

2. Set the pattern to compare to by setting the **ValidationExpression** property to a regular expression. To select from predefined patterns, click the **ValidationExpression** property in the Properties window and then click the ellipsis button (...).

   If you want to allow multiple valid patterns, use the bar character (|) to separate expressions.

   > **Note** In client-side validation, regular expressions are evaluated using ECMAScript (JScript). This differs in minor ways from server-side regular-expression checking.

3. Add a test in your Web Forms code to check for validity. For details, see Testing Validity Programmatically for ASP.NET Server Controls.

   The following example shows how you can use a **RegularExpressionValidator** control to check whether users enter a valid United States ZIP code. The validator checks for two patterns: five digits, and five digits plus a hyphen plus four more digits.

```
ZIP: <asp:TextBox id=txtZIP runat="SERVER"></asp:TextBox>
<asp:RegularExpressionValidator
     id=txtZIP_validation runat="SERVER"
     ControlToValidate="txtZIP"
     ErrorMessage="Enter a valid US ZIP code."
     ValidationExpression="\d{5}(-\d{4})?">
</asp:RegularExpressionValidator>
```

See Also

Introduction to Validating User Input in Web Forms | Controlling Validation Error Message Display for ASP.NET Server Controls

Validating Against a Range of Values for ASP.NET Server Controls

You can check that a user's entry falls within a specific range of values — for example, between two numbers, between two dates, or between alphabetic characters. You set the upper and lower bounds of the range as properties of a **RangeValidator** control.

You must also specify the data type of the values that the control will validate. If the user's entry cannot be converted to the specified data type — for example, it cannot be converted to a date — the validation fails.

To validate against a range of values

1. Add a **RangeValidator** control to the page and set the following properties:

Property	Description
ControlToValidate	The ID of the control you are validating.
ErrorMessage, **Text**, **Display**	Properties that specify the text and location of the error or errors that will display if the validation fails. For details, see Controlling Validation Error Message Display for ASP.NET Server Controls.

2. Set the low and high values of the range using the **MinimumValue** and **MaximumValue** properties.
3. Set the **Type** property to specify the data type of the range settings. Use the **ValidationDataType** enumeration, which allows you to use the type name **String**, **Integer**, **Double**, **Date**, or **Currency**.

   > **Note** If the user leaves a control blank, the control passes the range validation. To force the user to enter a value, add a **RequiredFieldValidator** control as well. For details, see Validating Required Entries for ASP.NET Server Controls.

4. Add a test in your Web Forms code to check for validity. For details, see Testing Validity Programmatically for ASP.NET Server Controls.

See Also

Introduction to Validating User Input in Web Forms | Controlling Validation Error Message Display for ASP.NET Server Controls

Validating Against Values in a Database for ASP.NET Server Controls

You can validate a user's entry against a database to be sure that what the user has entered is a recognized value. To do this, you must write code in a CustomValidator control that loops through a table and looks for data matches.

To validate against a database

1. Add a **CustomValidator** control to the page and set the following properties:

Property	Description
ControlToValidate	The ID of the control that you want to validate.
ErrorMessage, **Text**, **Display**	Properties that specify the text and location of the error or errors that will display if the validation fails. For details, see Controlling Validation Error Message Display for ASP.NET Server Controls.

2. Create an event handler for the CustomValidator control's ServerValidate event. In the event handler, add code to look through the database and check the user's input against the dataset.

   Note If the user leaves a control blank, the control passes the comparison validation. To force the user to enter a value, add a **RequiredFieldValidator** control as well. For details, see Validating Required Entries for ASP.NET Server Controls.

3. Add a test in your Web Forms code to check for validity. For details, see Testing Validity Programmatically for ASP.NET Server Controls.

The following example shows how you can validate a user's entry by looking it up in a table. In this instance, the user has entered an e-mail address that is validated against e-mail addresses stored in a table. The custom validation logic loops through the rows in a table that is part of the dataset available to the page.

```
' Visual Basic
Private Sub CustomValidator1_ServerValidate(ByVal _
   source As System.Object, ByVal args As _
   System.Web.UI.WebControls.ServerValidateEventArgs) _
   Handles CustomValidator1.ServerValidate
   Dim dv As DataView
   dv = dataset11.Tables(0).DefaultView
   Dim datarow As DataRowView
   Dim txtEmail As String
   args.IsValid = False    ' Assume False
   ' Loop through table and compare each record against user's entry
   For Each datarow In dv
      ' Extract e-mail address from the current row
      txtEmail = datarow.Item("Alias").ToString()
      ' Compare e-mail address against user's entry
      If txtEmail = args.Value Then
         args.IsValid = True
         Exit For
      End If
   Next
End Sub

// C#
private void CustomValidator1_ServerValidate(object source, System.Web.UI.WebControls.ServerV
alidateEventArgs args)
{
   DataView dv;
   dv = dataSet11.Tables[0].DefaultView;
   string txtEmail;
   args.IsValid = false;    // Assume False
   // Loop through table and compare each record against user's entry
   foreach(DataRowView datarow in dv)
   {
      // Extract e-mail address from the current row
      txtEmail = datarow["Alias "].ToString();
```

```
        // Compare e-mail address against user's entry
        if (txtEmail == args.Value)
        {
           args.IsValid = true;
        }
    }
}
```

See Also

Introduction to Validating User Input in Web Forms | Controlling Validation Error Message Display for ASP.NET Server Controls

Validating with a Custom Function for ASP.NET Server Controls

If the existing validation controls do not suit your needs, you can define a custom server-side validation function and call it using the **CustomValidator** control. You can also add client-side validation for a quick check before the form is submitted by writing a function in ECMAScript (JScript) that duplicates the logic of the server-side method.

You should perform a server-side validation even if you use a client-side check. Server-side validation helps prevent users from bypassing validation by disabling or changing the client script.

> **Security Note** By default, the Web Forms page automatically validates that malicious users are not attempting to send script or HTML elements to your application. As long as this validation is enabled, you do not need to explicitly check for script or HTML elements in user input. For more information, see Scripting Exploits.

Server-Side Validation

You define server-side validation by writing code in the control's ServerValidate event handler that checks the user's input. You hook up the event handler by setting the OnServerValidate attribute of the control. The string from the input control to validate can be accessed by using the **Value** property of the **ServerValidateEventArgs** object passed into the event handler as a parameter. The result of the validation is then stored in the **IsValid** property of the **ServerValidateEventArgs** object.

To validate on the server using a custom function

1. Add a **CustomValidator** control to the page and set the following properties:

Property	Description
ControlToValidate	The ID of the control you are validating.
ErrorMessage, **Text**, **Display**	Properties that specify the text and location of the error or errors that will display if the validation fails. For details, see Controlling Validation Error Message Display for ASP.NET Server Controls.

2. Create a server-based event handler for the control's **ServerValidate** event. This event is called to perform the validation. The method has a signature such as the following:

```
Protected Sub CustomValidator1_ServerValidate(ByVal _
   source As System.Object, ByVal args As _
   System.Web.UI.WebControls.ServerValidateEventArgs) _
   Handles CustomValidator1.ServerValidate

// C#
protected void ValidationFunctionName(object source, ServerValidateEventArgs args)
```

   The source parameter is a reference to the custom validation control raising this event. The property **args.Value** will contain the user input to validate, and **args.IsValid** should be set to **true** if the value is valid, **false** otherwise.

   The following example shows how you can create custom validation. The event handler checks whether the user's entry is eight characters long or longer.

```
' Visual Basic
Protected Sub TextValidate(ByVal source As System.Object, _
   ByVal args As System.Web.UI.WebControls.ServerValidateEventArgs) _
   Handles CustomValidator1.ServerValidate
   args.IsValid = (args.Value.Length >= 8)
End Sub

// C#
protected void TextValidate(object source, ServerValidateEventArgs args)
{
   args.IsValid = (args.Value.Length >= 8);
}
```

3. Add the **OnServerValidate** attribute to the validator in HTML view to indicate the name of the validation function.

```
<asp:textbox id=TextBox1 runat="server"></asp:textbox>
<asp:CustomValidator id="CustomValidator1" runat="server"
   OnServerValidate="TextValidate"
   ControlToValidate="TextBox1"
   ErrorMessage="Text must be 8 or more characters.">
</asp:CustomValidator>
```

4. Add a test in your Web Forms code to check for validity. For details, see Testing Validity Programmatically for ASP.NET Server Controls.

Client-Side Validation

You can also create client-side custom validation. To do so, you specify a function name for the control's ClientValidationFunction property, and create a function in client script that duplicates the logic of the server-side method.

To create custom validation logic on the client

1. Create a validation function on the client side in ECMAScript (JavaScript, JScript).

   The following example illustrates custom client-side validation. An excerpt from the page shows a **Textbox** control referenced by a **CustomValidator** control. The validation control calls a function called `validateLength` to make sure that the user has entered at least eight characters into the **Textbox** control.

```
<SCRIPT LANGUAGE="JavaScript">
function validateLength(oSrc, args){
   args.IsValid = (args.Value.length >= 8);
}
</SCRIPT>

<asp:Textbox id="text1" runat="server" text="">
</asp:Textbox>

<asp:CustomValidator id="CustomValidator1" runat=server
      ControlToValidate = "text1"
      ErrorMessage = "You must enter at least 8 characters!"
   ClientValidationFunction="validateLength" >
</asp:CustomValidator>
```

2. Add a test in your Web Forms code to check for validity. For details, see Testing Validity Programmatically for ASP.NET Server Controls.

See Also

Introduction to Validating User Input in Web Forms | Controlling Validation Error Message Display for ASP.NET Server Controls

Testing Validity Programmatically for ASP.NET Server Controls

Validation controls test user input, set an error state, and produce error messages. They do not change the flow of page processing — for example, they do not bypass your code if they detect a user input error. Instead, you test the state of the controls in your code before performing application-specific logic. If you detect an error, you prevent your own code from running; the page continues to process and is returned to the user with error messages.

> **Security Note** By default, the Web Forms page automatically validates that malicious users are not attempting to send script or HTML elements to your application. As long as this validation is enabled, you do not need to explicitly check for script or HTML elements in user input. For more information, see Scripting Exploits.

You can test for a general, page-wide state, and you can test the state of individual controls. Typically, you do these things in the event handlers that you create for a page.

> **Note** Validation information is not available during a page's initialization or load stage. However, you can call the **Validate** method manually during **Page_Load** and then test the **IsValid** property. For details about page states, see Web Forms Page Processing.

To test for a general error state

- In your code, test the page's **IsValid** property. This property rolls up the values of the **IsValid** properties of all the validation controls on the page (using a logical AND); if any one validation control is set to invalid, the page's property will return false.

  The following example shows the event handler for a button. The code tests the **IsValid** property of the entire page. Note that there is no need for an else clause, because the page will be returned automatically to the browser and the validation controls will display their own error messages.

```
' Visual Basic
Public Sub Button1_Click(ByVal sender As Object, _
ByVal e As System.EventArgs) _
Handles Button1.Click
   If Me.IsValid Then
      ' Perform database updates or other logic here
   End If
End Sub

// C#
void Button1_Click(object sender, System.EventArgs e)
{
   if (IsValid)
   {
      // Perform database updates or other logic here.
   }
}
```

To test for the error state of individual controls

- Loop through the page's Validators collection, which contains references to all the validation controls. You can then examine the **IsValid** property of each validation control.

  > **Note** If you want to perform this check during **Page_Load**, you must manually call the **Validate** method first.

  The following example shows how you can get information about individual validation controls.

```
' Visual Basic
If (Me.IsPostBack) Then
   Me.Validate()
   If (Not Me.IsValid) Then
      Dim msg As String
      ' Loop through all validation controls to see which
```

```
        ' generated the error(s).
        Dim oValidator As IValidator
        For Each oValidator In Validators
           If oValidator.IsValid = False Then
              msg = msg & "<BR>" & oValidator.ErrorMessage
           End If
        Next
        Label1.Text = msg
     End If
  End If

  // C#if (this.IsPostBack)
  {
     this.Validate();
     if (!this.IsValid)
     {
        string msg = "";
        // Loop through all validation controls to see which
        // generated the errors.
        foreach (IValidator aValidator in this.Validators)
        {
           if (!aValidator.IsValid)
           {
              msg += "<br>" + aValidator.ErrorMessage;
           }
        }
        Label1.Text = msg;
     }
```

See Also

Introduction to Validating User Input in Web Forms | Controlling Validation Error Message Display for ASP.NET Server Controls | Validating Programmatically for ASP.NET Server Controls

Controlling Validation Error Message Display for ASP.NET Server Controls

You can display error message text in four ways:

Display method	Description
In-line (only)	Next to the control (in the place where the validation control is located).
Summary (only)	In a separate summary of all errors, which is available only when the user submits the page. Optionally, the errors can appear in a message box (the option to use a message box is available only if the user is using Internet Explorer 4.0 or later).
In-line and summarized	The error message can be different in the summary and in place. You can use this option to show a shorter error message in place with more detail in the summary, or to show an error glyph next to the input field and an error message in the summary.
Custom	You can create your own error message display. For details, see Displaying Custom Error Messages for ASP.NET Server Controls.

You also have options for specifying the format and layout of error messages. For details, see Formatting Validation Error Messages for ASP.NET Server Controls.

If you want to display summarized error messages, you need to add a control to the page.

To add summarized error messages

1. Add a **ValidationSummary** control to the page at the location where you want to display the collected error messages.
2. Set the summary control's **DisplayMode** property to format the error messages, using one of the following values defined in the **ValidationSummaryDisplayMode** enumeration:

Value	Description
BulletList	(Default) Each message appears as a bulleted item.
List	Each message appears on its own line.
SingleParagraph	Each message appears as a sentence in a paragraph.

3. Optionally set the control's **HeaderText** property to provide a title or header for the error messages.

Choosing in-line and summary display involves setting different combinations of properties in the individual validation controls.

To choose in-line and summary error display

1. If you want to display summary errors, add a **ValidationSummary** control as described in the procedure above.
2. Set the **ErrorMessage**, **Text**, and **Display** properties of the individual validation controls according to the following table:

Option	ValidationSummary control required?	Property settings on validation control
Inline only	No	**Display** = **Static** or **Dynamic** **ErrorMessage** = (error text) or **Text** = (error text)
Summary only (including optional message box)	Yes	**Display** = **None** **ErrorMessage** = (error text) or **Text** = (error text)
Inline and summary (including optional message box)	Yes	**Display** = **Static** or **Dynamic** **ErrorMessage** = (error text for summary) **Text** = (in-line error text or glyph)

If the user is working with Internet Explorer 4.0 or later, summarized error messages can also appear in a message box.

To display summarized error messages in a message box

- Set the **ShowMessageBox** property of the **ValidationSummary** control to **true**.

When the user submits the page, errors are displayed both in the summary control and in the message box. If you do not want the errors to be displayed in the summary control, set the ShowSummary property to **False**.

See Also

Introduction to Validating User Input in Web Forms | Formatting Validation Error Messages for ASP.NET Server Controls | Specifying Layout for In-Place Messages On ASP.NET Server Controls

Formatting Validation Error Messages for ASP.NET Server Controls

You can control the format — font, size, and so on — of error text, or you can substitute a glyph for error text. For example, you can set the Text property of a validation control to an asterisk (*). When an error occurs, the asterisk will appear on the page in the location of the validation control (for example, beside the invalid field). You can include a detailed error message in the ErrorMessage property of the validation control and add a ValidationSummary control to the page; the detailed ErrorMessage property text will appear on the page in the location of the ValidationSummary control.

Validation controls support the same properties as other controls for specifying their appearance.

To format error messages

- Set validation control properties such as the following:

Property	Description
ForeColor	The color of the error message text.
BackColor	The color behind the text.
Font	The font face, size, weight, and so on.
BorderWidth, **BorderColor**, and **BorderStyle**	The size and color of a border around the error message.
CSSStyle and **CSSClass**	Style settings that are used if the user is working with a browser that supports CSS style sheets.

> **Note** Some of the settings you make with these properties may not be supported in all browsers.

You can provide formatting within an error message by using HTML elements in the text for the **ErrorMessage** and **Text** properties. For example, to apply italics to only one word in the error message, you can enclose the word in <I> and </I> tags.

> **Note** To display reserved HTML characters within the error message text, encode them. For example, to display "<", use the character sequence
> <
> .

If you are displaying validation errors in a summary, you can specify that a glyph appear in place of the individual validation control.

To display glyphs next to input controls

1. Set the **ErrorMessage** property of each validation control to the error text that should appear in the **ValidationSummary** control.
2. Set the **Text** property of each validation control to the glyph to display. The glyph can consist of any valid HTML, including text (such as an asterisk) or an image.

> **Note** The glyph is displayed in place of the validation control. It is up to you to lay out the validation control in such a way that the glyph displays properly.

See Also

Introduction to Validating User Input in Web Forms | Controlling Validation Error Message Display for ASP.NET Server Controls | Specifying Layout for In-Place Messages On ASP.NET Server Controls

Specifying Layout for In-Place Messages On ASP.NET Server Controls

When an error message appears on the page, it becomes part of the layout of the page. You can specify that the layout be fixed — the error message does not change the page layout when it appears — or dynamic. For details about layout options, see Validation Error Message Layout for ASP.NET Server Controls.

To specify fixed or dynamic layout for client error messages

- Set the **Display** property of each validation control to a layout option defined by the **ValidatorDisplay** enumeration:

Option	Description
None	Used when you are using only the summary control
Static	For a fixed layout
Dynamic	For a dynamic layout

Note Internet Explorer 4.0 or later is necessary to view static layout.

See Also

Introduction to Validating User Input in Web Forms | Controlling Validation Error Message Display for ASP.NET Server Controls | Formatting Validation Error Messages for ASP.NET Server Controls

Disabling Validation for ASP.NET Server Controls

You might want to bypass validation under certain circumstances. For example, if you have a "Cancel" button or a navigation button on a page, the page should be able to post even if the user did not fill out all the validated fields correctly. You can set an ASP.NET server control to bypass validation on both the server and the client, or just on the client.

> **Security Note** By default, the Web Forms page performs request validation to ensure that user input does not include script or HTML elements. You can disable this feature explicitly. For more information, see Scripting Exploits and Protecting Against Script Exploits in a Web Application.

Disabling Server-Side and Client Side Validation

You can specify that individual controls on a Web Forms page cause a postback without triggering a validation check.

To cause a specific control to bypass validation

- Set the control's **CausesValidation** property to **false**.

  The following example shows how you can create a Cancel button that does not perform a validation check:

```
<asp:Button id="Button1" runat="server"
   Text="Cancel" CausesValidation="False">
</asp:Button>
```

You can also disable a validation control, so that it is not rendered to the page at all and no validation takes place using that control.

To disable a validation control

- Set the validation control's **Enabled** property to **false**.

Disabling Client-Side Validation Only

If you want to perform validation on the server but not on the client, you can set an individual validation control to not emit client-side script. This is useful if dynamic updating on the client creates problems with the layout of the page, or if you want to execute some server code before validation takes place.

To disable client-side validation

- Set the validation control's **EnableClientScript** property to **false**.

See Also

Introduction to Validating User Input in Web Forms | Client-Side Validation for ASP.NET Server Controls

Displaying Custom Error Messages for ASP.NET Server Controls

You do not have to use the validation control to output a response when the **IsValid** property is false. On both the client and server side you can create a custom response, such as a color change in a control or a font change for text on a label.

To display custom validation messages

- In the **Page_Load** event handler, call the **Validate** method, then check the page's **IsValid** property and conditionally add text or a control, or change the properties (for example, color) of a control. The following example displays the label "All entries are valid" when the **IsValid** property is **true**, and "There are one or more invalid entries" when the property is **false**.

```
' Visual Basic
If (Me.IsPostBack) Then
   ValidationControl1.Validate()
   If (ValidationControl1.IsValid) Then
      lblOutput.Text = "All entries are valid."
   Else
      lblOutput.Text = "There are one or more invalid entries."
   End If
End If

// C#
if (this.IsPostBack)
{
   ValidationControl1.Validate();
   if (ValidationControl1.IsValid)
   {
      lblOutput.Text = "All entries are valid.";
   }
   else
   {
      lblOutput.Text = "There are one or more invalid entries.";
   }
}
```

See Also

Introduction to Validating User Input in Web Forms | Controlling Validation Error Message Display for ASP.NET Server Controls | Formatting Validation Error Messages for ASP.NET Server Controls

Validating Programmatically for ASP.NET Server Controls

By default, the validation controls perform validation automatically when the page is posted back to the server, after page initialization (that is, after view-state and postback data have been processed) and before your event-handling code is called. The controls might also perform validation in the browser if the browser supports the necessary client script.

However, there are times when you might want to perform validation yourself under program control. When you do, you are doing what the page does automatically: calling a validation control's **Validate** method, which causes the control to perform its validation and set its **IsValid** property. You might validate programmatically under these circumstances:

- If validation values are not set until run time. For example, if you are working with a **RangeValidator** control, you might be setting its **MinimumValue** and **MaximumValue** properties at run time based on values entered by the user. In this case, default validation does not work, because at the time that the page calls the validation controls to perform validation, there is insufficient information in the **RangeValidator** control.
- If you want to check the validity of a control (or of the page as a whole) in the **Page_Load** event handler. At that stage of page processing, the validation controls have not yet been invoked, so the **IsValid** property of the page or of individual controls is not set. (It will throw an exception if you try to get its value.) However, if you want to check validity, you can programmatically call validation.
- If you are adding controls (either input controls or validation controls) at run time.

More generally, you can validate programmatically any time when you want to have precise control over when validation is carried out.

> **Security Note** By default, the Web Forms page automatically validates that malicious users are not attempting to send script or HTML elements to your application. If you have disabled this feature, you can call the Request.ValidateInput method yourself.

To validate programmatically

1. In an event handler, write code to set the validation control properties that you want to change.
2. Call the validation control's **Validate** method.

   The control will perform its check and, if an error is detected, will set the **IsValid** property of the control and the page to **False**. When the page is returned to the user, error messages will be displayed as usual.

   The following example shows how you can set properties programmatically. In this case, a Web Forms page is set up to take reservations at a resort that features a free tour during every visit. Users must enter an arrival date and a departure date, and then schedule the tour during their visit. A **RangeValidator** control is used to ensure that the user entered a typical date format, and that the tour date falls between the arrival and departure dates.

   > **Note** If a user enters a value that cannot be converted to a date, the validator control will throw an exception that must be handled. This example, however, does not include error handling.

   The arrival and departure dates come from two **TextBox** Web server controls on the page, txtArrival and txtDeparture. The tour date is entered in a third **TextBox** control, txtTourDate, which is validated by the **RangeValidator** control.

   > **Note** When validating programmatically, you should disable client script so that the control will not show the wrong error message before your server-side validation code executes. For details, see Disabling Validation for ASP.NET Server Controls.

```
' Visual Basic
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As _
System.EventArgs) Handles Button1.Click
   RangeValidator1.MinimumValue = txtArrival.Text
   RangeValidator1.MaximumValue = txtDeparture.Text
   RangeValidator1.Type = ValidationDataType.Date
   RangeValidator1.Validate()
   If Not RangeValidator1.IsValid Then
      RangeValidator1.ErrorMessage = "The tour date must " & _
      "fall between the arrival and departure dates."
   End If
```

```
End Sub

//C#
private void Button1_Click(object sender, System.EventArgs e)
{
   RangeValidator1.MinimumValue = txtArrival.Text;
   RangeValidator1.MaximumValue = txtDeparture.Text;
   RangeValidator1.Type = ValidationDataType.Date;
   RangeValidator1.Validate();
   if (!RangeValidator1.IsValid)
   {
      RangeValidator1.ErrorMessage = "The tour date must " +
      "fall between the arrival and departure dates.";
   }
}
```

See Also

Introduction to Validating User Input in Web Forms | Controlling Validation Error Message Display for ASP.NET Server Controls | Testing Validity Programmatically for ASP.NET Server Controls

Walkthrough: Validating User Input in a Web Forms Page

As with any input form, Web Forms pages require you to make sure that users are entering information that is valid, is in the right format, and so on. To make this common task easier, you can use validation controls created especially for Web Forms pages. Validation controls allow you to check for required entries, data types, ranges, patterns, and specific values by simply setting properties. The controls perform checking automatically and include a variety of ways for displaying error information to users. For an overview, see Web Forms Validation.

> **Security Note** By default, the Web Forms page automatically validates that malicious users are not attempting to send script to your application. For more information, see Scripting Exploits.

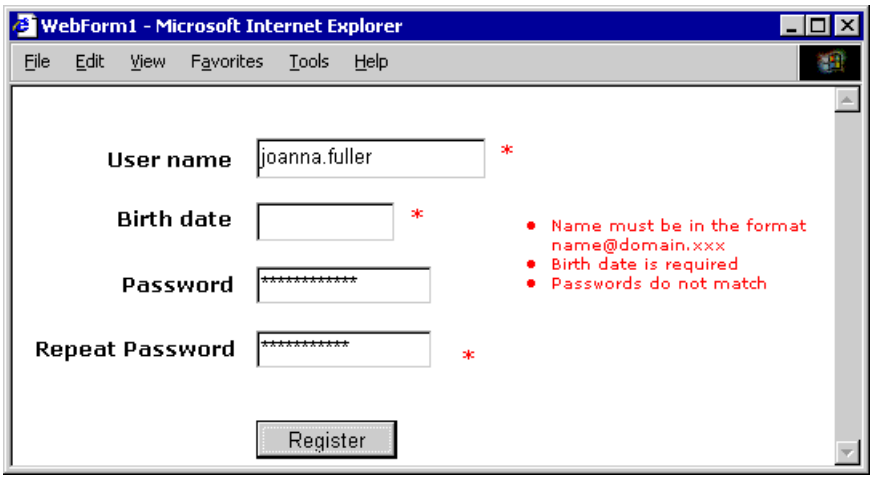
This walkthrough illustrates a simple Web Forms page that you might create as a simple registration page. The page includes text boxes to allow users to enter a login name, a birth date, a password, and a repeat of the password. You will add validation controls to the page to be sure that users enter values into all the boxes and that the values are formatted correctly.

In order to complete this walkthrough, you will need sufficient permissions to be able to create ASP.NET Web application projects on the computer where your Web server is.

The walkthrough is split into a number of smaller pieces:

- Creating the basic form by creating a project and Web Forms page and adding the input controls and labels.
- Adding the validation controls. You will validate for required entries, that the user has entered a valid login name and date, and that the passwords match.
- Testing.

When you have finished the walkthrough, you will have a form that looks similar to the following.



Creating the Basic Form

You can start by creating a form with input controls on it.

Creating the Form

The first step is to create a form that you will later add validation controls to. Start by creating a new Web application project and Web Forms page.

To create the project and form

1. On the **File** menu, point to **New**, and then click **Project**.
2. In the **New Project** dialog box, do the following:
   a. In the **Project Types** pane, choose either **Visual Basic Projects** or **Visual C# Projects**.
   b. In the **Templates** pane, choose **ASP.NET Web Application**.
   c. In the **Location** box, enter the complete URL for your application, including http://, the name of the server, and the name of your project. The Web server must have IIS version 5 or later and the .NET Framework installed on it. If you have IIS on your computer, you can specify `http://localhost` for the server. (If you normally use a proxy server to access the Internet, you might need to configure Internet Explorer to bypass the proxy server in order to use localhost.)

      When you click **OK**, a new Web Forms project is created at the root of the Web server you specified. In addition, a new

Web Forms page called WebForm1.aspx is displayed in the Web Forms Designer in Design view.

Tip If you have trouble creating a Web application project, see Web Access Failed Dialog Box.

3. If the page is not already in grid mode, use the Properties window to set the document object's **pageLayout** property to **GridLayout**.

   This step is optional, but makes it easier to lay out the form. If you prefer to work in flow layout, you can, but you might have to make minor adjustments to where the controls appear on the page. The page will work the same way in either case.

Adding Input Controls

For this walkthrough, which assumes that you are creating a simple registration page, you will add controls to allow users to enter a name (an e-mail address), a birth date, and a password. Users enter the password twice to confirm it.

To add the controls

1. From the **Web Forms** tab of the Toolbox, add the following controls to the form, setting their **ID** property as indicated:

Control	Purpose	ID
TextBox	Enter user name	**txtName**
TextBox	Enter birth date	**txtBirthdate**
TextBox	Enter password	**txtPassword**
TextBox	Repeat password to confirm it	**txtPasswordAgain**
Button	Submit form	**btnRegister** Set the button's **Text** property to Register.

2. Set the **TextMode** property of the two password boxes to **Password** so that they display stars (*) as users type in them.
3. Add labels in front of the text boxes to indicate their function. If the Web Forms page is in grid mode, add **Label** controls in front of each text box and set their **Text** property. If the page is in flow mode, simply type static text in front of each box.
4. Adjust the font, size, and layout of the controls to your liking. For details about layout, see Positioning HTML Elements in Design View.

Adding Validation Controls

Now that you have created the input form, you can add validation. You do so by adding validation controls to the form, one control for each validation test you want to perform. (In all, you will add seven validation controls to cover all the tests you should perform plus a control to display errors.) The validation tests you will perform are these:

- Required fields. You want to be sure that users enter values into all four boxes.
- Name format. For this walkthrough, you will check that the user's login name follows the pattern of typical e-mail names: *name@domain.xxx*. In this walkthrough, you will not validate that the e-mail name the user provides belongs to a valid e-mail account.
- Date format. You will validate that the user's entry for birth date is a valid date. (In this walkthrough, you will not validate that the date falls within a specific range.)
- Password match. You will validate that the passwords that the user supplies in the two password boxes match.

Checking For Required Fields

You want to be sure that users enter values into all four text boxes — that is, all four text boxes are required fields.

To check for required fields

1. From the **Web Forms** tab of the Toolbox, drag a **RequiredFieldValidator** control onto the form, next to the **txtName** text box.
2. Select the validation control and in the Properties window, set the following properties.

Property	Setting
ControlToValidate	`txtName` This indicates that the validator will check the contents of the name field.

Text	`*` (star) This is the text that will be displayed where the validator control is now. (It is common to put a red star next to an input box containing errors.)
ErrorMessage	`Email name is required.` This text will be displayed in a summary on the page.

3. If you like, set the font, color, and formatting properties for the validation control. These properties affect how errors are displayed.
4. Repeat Steps 1 through 3 for the other three text boxes, creating three more **RequiredFieldValidation** controls with the following settings.

Control	Settings
RequiredFieldValidator2	• **ControlToValidate**: Set to `txtBirthdate` • **Text**: `*` • **ErrorMessage**: `You must enter a birth date.`
RequiredFieldValidator3	• **ControlToValidate**: Set to `txtPassword` • **Text**: `*` • **ErrorMessage**: `You must enter a password.`
RequiredFieldValidator4	• **ControlToValidate**: Set to `txtPasswordAgain` • **Text**: `*` • **ErrorMessage**: `Re-enter the password to confirm it.`

Checking Name Format

For the walkthrough, you are assuming that the user should enter an e-mail address as the registration name. Although you cannot check that the name represents a valid e-mail account, you can check that the name conforms to typical e-mail address format (such as *name@domain.com*). This catches entry errors that users often make, such as forgetting to enter the ".com" portion of their e-mail address.

To check for patterns such as this, you use a validator control that incorporates a regular expression. If you know regular expression syntax, you can create your own pattern-matching expressions. However, you can also choose from a list of predefined regular expressions, including those to check for e-mail addresses, phone numbers, and postal codes.

When you add this validation control, you will have two controls for the name box; the contents of the box must pass both checks to be considered valid. When you have multiple validators for a single control, you generally want them to display their text (typically, a star) in the roughly the same location, no matter which validation check fails. You can specify this by setting the validator control's Display property. For more information, see Validation Error Message Layout for ASP.NET Server Controls.

To check name format

1. From the **Web Forms** tab of the Toolbox, drag a **RegularExpressionValidator** control next to the **txtName** box.
2. Select the validation control and in the Properties window, set the following properties.

Property	Setting
ControlToValidate	`TxtName`
Text	`*` (star)
ErrorMessage	`Name must be in the format name@domain.xxx.`
ValidationExpression	From the list, select **Internet E-mail Address**. Note that this sets the property value to a regular expression.

Checking Date Format

You want to be sure that users enter a valid date for the birth date field. You can do this with a validation control that performs two functions at once: it checks that the user has entered a date (by data type) and it compares that the date is later than a specified minimum date.

To check date format

1. From the **Web Forms** tab of the Toolbox, drag a **CompareValidator** control next to the **txtBirthdate** box.
2. Select the validation control and in the Properties window, set the following properties.

Property	Setting
ControlToValidate	`TxtBirthdate`
Text	* (star)
ErrorMessage	`Birth date is not a valid date.`
ValueToCompare	`1900/01/01` (Or any other minimum date.) **Note** Enter the date to compare in the format yyyy/mm/dd; this ensures that it will be interpreted properly for all culture settings. Users must enter the date using only numbers for the month.
Operator	`Greater Than`
Type	`Date`
Display	**Dynamic**

Checking for Password Match

As is done often, your registration form asks users to type their password twice to confirm that that is what they want to enter. You can check this using the same validation control you used for the date — a comparison control — but instead of checking against a specific value, you will compare the value of one field — the second password box — against what they typed into the first password box.

> **Security Note** Using a **TextBox** control set for passwords can help ensure that other people will not be able to determine a user's password if they observe the user entering it. However, the password text entered is not encrypted in any way, and you should protect it as you would any other confidential data. For example, for maximum security when posting a form with a password in it, you might use secure sockets layer (SSL) and encryption. For more information, see Web Application Security at Run Time.

To check for password match

1. From the **Web Forms** tab of the Toolbox, drag a **CompareValidator** control next to the **txtPasswordAgain** box.
2. Select the validation control and in the Properties window, set the following properties.

Property	Setting
ControlToValidate	`TxtPasswordAgain`
Text	* (star)
ErrorMessage	`The passwords do not match.`
ControlToCompare	`TxtPassword`
Operator	`Equal`
Display	**Dynamic**

Displaying Validation Errors

One step remains. You have configured the validation controls so that they display a red star next to an input box containing an error. However, this might not give users enough information. For example, they might realize that they have entered a bad registration name, but might not be able to figure out what is wrong with it.

To display details about validation errors, you can use a summary control. The control displays the value of the **ErrorMessage** property of any validation control that has detected an error. If there is more than one error, the summary control can display them all, either as bulleted items, in a simple list, or as a paragraph of text.

To display validation errors

1. From the **Web Forms** tab of the Toolbox, drag a **ValidationSummary** control onto the form. Position the control someplace where it has room to display several lines of text.
2. If you want to display errors in a format other than bulleted items, select a different display option for the **DisplayMode** property.

3. Set the **Font** and **ForeColor** properties to format the error message text.
4. You are done adding validation controls. Now you can test to see how they work.

Testing Validation

You want to test the Web Forms page with different values to see the effect of the validation controls.

To test the validation controls

1. In Solution Explorer, right-click the name of your form and choose **View in Browser**.
2. When the page is displayed, enter various values into the boxes. To check the birth date, enter a date that is invalid for the locale of your Web server.

   Each time you make an error, a star should appear next to the box containing the error.

3. Click **Register**.

   If there are any errors, the error message text appears in the summary control. (Red stars appear and disappear immediately as you work with the fields.)

   > **Note** Required fields are checked the first time only when you submit the form, so that users are not forced to enter data into the form in a specific order. However, after the first time the form is submitted, errors are reported until the field is filled in.

If you are not seeing the errors you expect to see, review the settings for each of the validation controls and try again.

Next Steps

The validation you added in this walkthrough to the Web Forms page illustrates the basic concepts of Web validation controls. There is more you can do with validation controls, both with their basic functionality as well as in code. You could expand on the walkthrough in these ways:

- Check that the user enters a birth date earlier than today by adding a second **CompareValidator** control and setting its operator to **LessThan** and its comparison value to today. You can use a data-binding expression to set the value. Although you are not binding the validator control to data, you are taking advantage of the fact that data binding expressions are evaluated before the control is rendered. The ASP.NET syntax for the control might look like this:

  ```
  <asp:CompareValidator id=c1
  ValueToCompare='<%# DateTime.Today.ToShortDateString() %>'
  ControlToValidate="txtBirthDate" Operator="LessThan" Type="Date" Text="*" runat="server"
  />
  ```

  > **Note** You must call the validator control's **DataBind** method to resolve the expression.

- Check the length of the password to be sure it meets a certain minimum number of characters. You can do this with a **RegularExpressionValidator** control. There is no predefined expression that checks for a minimum number of characters. You can use an expression such as the following, which requires five or more characters:

  ```
  ^[\w]{5,}
  ```

- Test validation in code. Each validation control sets a flag that you can test programmatically. This complements the visual feedback that the controls provide; by testing for valid input in code, you can determine how to proceed (or not proceed) in your application. For details, see Validating Programmatically for ASP.NET Server Controls.
- Learn about the security issues associated with Web Forms. For details, see Overview of Web Application Security Threats.

See Also

Introduction to Validating User Input in Web Forms

Client Script in Web Forms Pages

If you know that the users of your Web Forms pages have uplevel browsers, you can add functionality to your pages by including client script that runs in the browser, apart from the server controls you are using.

> **Note** For information about uplevel and downlevel browsers, see ASP.NET Server Controls and Browser Capabilities.

Because client script runs in the browser, you can include it in the page for operations that manipulate the look of the page and require quick response time. Examples include enabling and disabling controls, performing mouseover operations, animating elements on the page, and so on.

As a rule, client script runs entirely separately from the server code that you create for your Web Forms page, and you can therefore create whatever script is required in your page. However, a few ASP.NET server controls depend on client script for their functionality. This does not prevent you from using client script when working with those controls, but you do need to understand the interaction of your client script with the client script that is produced by these controls.

The ASP.NET server controls that depend on being able to run client script include:

- The LinkButton and HtmlButton server controls require script. (This is not true for the Button Web server control or the HtmlInputButton or HtmlInputImage controls.)
- By default, the Calendar control implements month navigation and day selection using **LinkButton** controls. If you set control properties to allow users to select a day, week, or month, or if you allow users to navigate to other months, then the **Calendar** control will generate client script. If you use the **Calendar** control simply to display a single month with no selection or navigation, the control does not require client script.
- Any Web server control whose **AutoPostBack** property is set to **true**; the client script is required so that the control will post the page.
- The Web validation controls, which require client script to support client-side validation. If the client does not support script, validation will run on the server only.

> **Note** For information on creating event handlers that run in both client script and in server code, see ASP.NET Server Control Event Model.

The client script required is automatically generated and sent as part of the page. It is compatible with all types of browsers that support at least ECMAScript (JScript, JavaScript) — that is, it does not depend on the advanced scripting capabilities of Dynamic HTML (DHTML).

> **Note** Some validation features can take advantage of DHTML if it is available in the browser, but DHMTL is not required.

Users sometimes disable script in a browser as a security measure. If they have, the functionality provided by client script is lost. This disables some controls, such as the **LinkButton** control, entirely, and disables the functionality of the **AutoPostBack** property. Validation is only partially affected; validation still functions, but users do not get the immediacy and performance of client-side validation.

See Also

ASP.NET Server Controls and Browser Capabilities | ASP.NET Server Control Event Model | Web Forms Validation

Controls You Can Use on Web Forms Pages

This section discusses what you can do with each of the Web server controls available in Visual Studio. If you want to use HTML server controls in your application, see HTML Server Controls.

> **Note** You can also create output for mobile devices. To do so, you use the same ASP.NET page framework, but you create Mobile Web Forms instead of Web Forms pages and use controls specifically designed for mobile devices. For details, see Creating ASP.NET Mobile Web Applications.

In This Section

AdRotator Web Server Control
Describes the AdRotator, which cycles through a series of clickable ad banners.
Button Web Server Controls
Describes the Button controls, which allow the user to send a command, such as submitting a form to a server.
Calendar Web Server Control
Describes the Calendar control, which allows users to view and select dates.
CheckBox and CheckBoxList Web Server Controls
Describes these option controls, which provide a way for users to switch between true/false, yes/no, or on/off options.
DataGrid Web Server Control
Describes the DataGrid control, a multi-column, data-bound grid which offers extensive design-time customization.
DataList Web Server Control
Describes the DataList control, which displays rows of database information in customizable format.
DropDownList Web Server Control
Describes the DropDownList control, which enables users to select from a single-selection list box.
HyperLink Web Server Control
Describes the Hyperlink control, which provides a means of creating and manipulating links on a Web page from server code.
Image Web Server Control
Describes the Image control, which allows you to display images on a Web Forms page and manage the images in your own code.
Label Web Server Control
Describes the Label control, which provides a way to display text under programmatic control in a Web Forms page.
Literal Web Server Control
Describes the Literal control, which renders static text into a Web page without adding any HTML elements.
ListBox Web Server Control
Describes the ListBox control, which allows users to select one or more items from a predefined list.
Panel Web Server Control
Describes the Panel control, which provides a container within the page for other controls.
PlaceHolder Web Server Control
Describes the PlaceHolder control, which enables you to place an empty container control in the page and then dynamically add child elements to it at run time.
RadioButton and RadioButtonList Web Server Controls
Describes these option controls, which allow users to select one item from a short, predefined list.
Repeater Web Server Control
Describes the Repeater control, which is a data-bound container control that produces a list of individual items.
Table, TableRow, and TableCell Web Server Controls
Describes these controls, which are used to construct general-purpose tables on a Web Forms page.
TextBox Web Server Control
Describes the TextBox control, which provides a way for users to type information into a Web Forms page, including text, numbers, and dates.
Validation Web Server Controls
Describes the controls you can use to validate input on your Web Forms pages.
XML Web Server Control
Describes the XML control, which reads XML and writes it into a Web Forms page at the location of the control.
Simple HTML Controls
Briefly describes the controls that appear on the HTML tab of the Toolbox.

Related Sections

Introduction to ASP.NET Server Controls
Describes the primary types of server controls and how they differ.

ASP.NET Server Controls by Function

List the controls according to general function and includes links to the control topics.

HTML Server Controls

Provides reference information, including syntax for the HTML server controls.

Controls for Displaying Lists On Web Pages

You can use several different Web server controls to display data in list form. Except for the Table control, these controls are bound to a data source. They produce one item — an entry — for each record in the data source.

The following table lists the Web server controls that display lists and differences between them.

Type of Control	Purpose	Features
Table	• General-purpose, programmable table.	• Can display any combination of HTML text and controls. • Supports a **TableCell** control that can be set to display information. Does not use templates for display. • Is not inherently data-bound. • Exposes a model for dynamically creating rows (**TableRow** controls) and cells (**TableCell** controls).
Repeater	• Simple, read-only output	• Has no built-in support for selecting or editing items. • Has no default appearance; you must lay out the list by creating templates. The list can be vertical, horizontal, all on one line, or in any format you specify. • Has no default paging; all data is displayed in a single list. • Can define separators between elements using a template. • Supports custom functionality that can be applied to items (for example, specifying an "Add to shopping cart" button for each item).
DataList	• List output with editing • Non-tabular lists (for example, comma-delimited lists) • Easily customizable output	• Has a table appearance by default, but can be configured to show any list output. • Can customize look of list extensively. • Has an auto-format option. • WYSIWYG template editing. • Supports styles for custom look of items. • Can define separators between elements using a template. • Has editable contents (contents displayed in text boxes or other controls, based on data type of bound data). • Supports single selection. Multiple selection requires custom code. • Supports either horizontal (columns) or vertical (rows) layout of the data items. • Has no default paging; all data is displayed in a single list. • Supports custom functionality that can be applied to items (for example, specifying an "Add to shopping cart" button for each item).
DataGrid	• Full-featured list output with editing • Paged reporting	• Has a grid appearance by default. • Can customize look of grid extensively. • Has an auto-format option. • Can specify output using bound columns, columns of buttons or hyperlinks, and custom columns created using templates. • Has no separator template. However, the grid renders in a table, and you can specify table border size and color. • Supports WYSIWYG template editing. • Items support styles for custom look. • Can edit, update, and delete contents. • Supports single selection. Multiple selection requires custom code. • Has optional paged output. • Supports sorting. • Supports custom functionality that can be applied to items (for example, specifying an "Add to shopping cart" button for each item).

AdRotator Web Server Control

The **AdRotator** Web Server Control cycles through a series of clickable ad banners, and allows some ads to be weighted more heavily than others. Ads can either be linked to the control using an XML file with a predefined schema or by creating your own custom logic.

The following topics provide information that you will find useful in learning to work with the **AdRotator** control.

In This Section

Introduction to the AdRotator Web Server Control
Provides a general description of the **AdRotator** control.
Adding AdRotator Controls to a Web Forms Page
Gives directions for placing **AdRotator** controls on a Web Forms page.
Creating an Ad List for the Ad Rotator Web Server Control
Describes how to create an ad list using the XML designer.
Selecting Ads in an AdRotator Web Server Control at Run Time
Gives directions on how to select ads in code.
Tracking Ad Responses for the AdRotator Control
Describes a strategy for gathering statistics about which ads users click.

Related Sections

Adding Web Server Controls to a Web Forms Page
Gives directions for including Web server controls on your Web Form.
Setting ASP.NET Server Control Properties
Suggests additional topics of interest concerning server control properties.
Setting Web Server Control Properties Programmatically
Gives directions for setting Web server control properties in code.

Introduction to the AdRotator Web Server Control

Advertisements on Web pages typically take the form of ad banners — .gif files or similar images — that when clicked redirect the user to the advertisers' Web pages. In order to generate ads on your Web page you will need the image files and accompanying URLs.

Using the AdRotator Web server control you can display and cycle through a series of ad banners. The **AdRotator** automates the cycling process, changing the displayed ad each time the page is refreshed. Ads can be weighted to control the priority level of banners making it possible to have certain ads display more often than others. You can also write custom logic that cycles through the ads.

Use the **AdRotator** control to retrieve and display ads with one of two methods:

- Create an XML file containing references to ad banners and their associated properties.
- Write your own logic to select an ad banner in the **AdCreated** event.

XML File Format

One method of storing ad banner image locations, URLs for redirection, and associated properties is to put the information in an XML file. By using the XML file format you can create and maintain a list of advertisements without having to change the code in your application when a change is made to an advertisement. The XML file can be created using the **Ad Rotator Schedule File** template in the XML designer. For details, see Creating an Ad List for the Ad Rotator Web Server Control.

Filtering Ads by Keyword

In the XML file you can assign categories to ads and then configure the **AdRotator** control to show ads according to filter criteria you specify. For example, if a specific **AdvertisementFile** contains categories for both banks and hardware stores you may only want the page to display ads related to banks. By setting the **KeywordFilter** property to "banks" the hardware store ads will be filtered out.

If a filter is specified then one of two situations can result:

- If the ad source is an XML advertisement file, then ads with a matching keyword will be selected.
- If the ad source is an XML advertisement file, but no match exists in the ad file, then the **AdRotator** will display a blank image and a trace warning will be generated.

Tracking Ad Response

The **AdRotator** control does not inherently provide a way to gather statistics about which ads users have clicked, how many times an ad has been clicked, and so on. However, because you have control over the URL to which the ad points, you can add tracking logic to your application for the ads. A typical way to do this is to point all ads to a tracking page that first gathers the statistics you want, and then jumps to the ad's destination page. For details, see Tracking Ad Responses for the AdRotator Control.

See Also

Adding AdRotator Controls to a Web Forms Page | Creating an Ad List for the Ad Rotator Web Server Control | Selecting Ads in an AdRotator Web Server Control at Run Time

Adding AdRotator Controls to a Web Forms Page

First you add the **AdRotator** control to a page, and then link it to the ads you wish to display. Link ads to the control either by using a separate XML file containing the ad information or by linking the ads to the control within an event handler at run time.

To add an AdRotator control to a Web Forms page

1. In Design view, drag an **AdRotator** control from the **Web Forms** tab of the Toolbox onto the page.
2. Link the **AdRotator** to an XML advertisement file:
   a. Select the **AdRotator**, open the Properties window, and then click the ellipsis button (...) next to the **AdvertisementFile** property.

      The **Select Advertisement XML File** dialog box appears.

   b. Navigate to the location of the XML file holding the information about your ads and click **Open**.

      Note If the **AdvertisementFile** property is set to an XML file, then the **AdCreated** event occurs after the ad has been selected.

      Note Alternately, you can set the ad properties in the **AdCreated** event. For details, see Selecting Ads in an AdRotator Web Server Control at Run Time.

See Also

Introduction to the AdRotator Web Server Control | Creating an Ad List for the Ad Rotator Web Server Control | Selecting Ads in an AdRotator Web Server Control at Run Time

Creating an Ad List for the Ad Rotator Web Server Control

You can create an ad list for the **AdRotator** control in the XML designer using the **Ad Rotator Schedule File** as the target schema. The following attributes can be included in the files:

- **ImageUrl** The URL of the image to display
- **NavigateUrl** The URL of the page to navigate to when the **AdRotator** control is clicked.
- **AlternateText** The text to display if the image is unavailable.
- **Keyword** The category of the ad that can be used to filter for specific ads.
- **Impressions** A numeric value that indicates the likelihood of how often the ad is displayed. The total of all impressions values in the XML file may not exceed 2,048,000,000 - 1

All attributes are optional.

To create an ad list using the XML designer

1. From the **Project** menu select **Add New Item**.
2. Select **XML file** from the **Templates** window and give it a unique name.
3. From the **TargetSchema** property in the Properties window, select **Ad Rotator Schedule File** from the list.
4. Insert a new `<Ad>` element within the `<Advertisements>` element for each ad you want to include in the ad list.
5. Set the necessary properties for each ad from among the following attributes:

```
<ImageUrl>...</ImageUrl>
<NavigateUrl>...</NavigateUrl>
<AlternateText>...</AlternateText>
<Keyword>...</Keyword>
<Impressions>...</Impressions>
```

6. Save the file.

See Also

Adding AdRotator Controls to a Web Forms Page | Introduction to the AdRotator Web Server Control | XML Designer | Tracking Ad Responses for the AdRotator Control

Selecting Ads in an AdRotator Web Server Control at Run Time

You can dynamically select ads by creating custom logic within an **AdCreated** event handler. If you have more than one **AdRotator** control in your project it is better to create a global event handler that responds to all of your **AdRotator** controls. In this way the logic for rotating the ads in each control can be handled by the same event.

To select ads using the AdCreated event

1. Create an **AdCreated** event handler for the control.

   The second parameter passed in the handler contains a reference to the ad being created.

2. Set properties of the ad object to specify the image and navigation URLs of the ad to deploy.

   The following example shows how you can set the **ImageUrl**, **NavigateUrl**, and **AlternateText** properties of an **AdRotator** control (AdRotator1) in the **AdCreated** event handler:

```
' Visual Basic
Public Sub AdRotator1_AdCreated(ByVal sender As Object, ByVal e As System.Web.UI.WebContr
ols.AdCreatedEventArgs) Handles AdRotator1.AdCreated
    e.ImageUrl = "images/msft_ad.gif"
    e.NavigateUrl = "http://www.microsoft.com/"
    e.AlternateText = "Where do you want to go today?"
End Sub

// C#
protected void AdRotator1_AdCreated (object sender,
   System.Web.UI.WebControls.AdCreatedEventArgs e)
{
   e.ImageUrl = "images/msft_ad.gif";
   e.NavigateUrl = "http://www.microsoft.com/";
   e.AlternateText = "Where do you want to go today?";
}
```

If you have several **AdRotator** controls in your project, you can create a single global event handler in the Global.asax file. This event handler can then be used by each control.

To create a global event handler to handle more than one AdRotator control

1. In the Global.asax file, create an event handler for the **GlobalAdCreated** event.

   The following example retrieves information from an imaginary commercial ad server called **AcmeAdServer** by adding an XML Web service (**AdService**) as a Web reference to the project. The AdService has a method called **GetNextAd** that contains the logic for specifying which ad is shown.

   The **GlobalAdCreated** event passes the **KeywordFilter** of the **AdRotator** control as a keyword into the **AdService.GetNextAd()** method, which returns the **ImageUrl**, **NavigateUrl**, and **AlternateText** from the **AdInfo** (a structure declared within the **AdService**) for the next ad to be displayed. The **GlobalAdCreated** event is called each time the page refreshes.

```
' Visual Basic
Public Shared Sub GlobalAdCreated(ByVal sender As Object, ByVal e As System.Web.UI.WebCon
trols.AdCreatedEventArgs)

    Dim ai As AcmeAdServer.AdInfo
    Dim ws As New AcmeAdServer.AdService()
    Dim keyword As String

    keyword = CType(sender, AdRotator).KeywordFilter
    ai = ws.GetNextAd(keyword)
```

```
    e.ImageUrl = ai.ImageUrl
    e.NavigateUrl = ai.NavigateUrl
    e.AlternateText = ai.AltText

End Sub

// C#
public static void GlobalAdCreated(Object sender, System.Web.UI.WebControls.AdCreatedEven
tArgs e)
{
   AcmeAdServer.AdInfo ai;
   AcmeAdServer.AdService1 ws = new AcmeAdServer.AdService1();
   String keyword;

   keyword = ((AdRotator)sender).KeywordFilter;
   ai = ws.GetNextAd(keyword);

   e.ImageUrl = ai.ImageUrl;
   e.NavigateUrl = ai.NavigateUrl;
   e.AlternateText = ai.AltText;
}
```

2. In the Web Form's load event, link the **AdRotator** controls' **AdCreated** events to the event handler defined in Global.asax.

```
' Visual Basic
Protected Sub Page_Load(ByVal Sender As System.Object, ByVal e As System.EventArgs) Handl
es MyBase.Load
    AddHandler AdRotator1.AdCreated, AddressOf Global.GlobalAdCreated
    AddHandler AdRotator2.AdCreated, AddressOf Global.GlobalAdCreated
End Sub

// C#
protected void Page_Load(object sender, EventArgs e)
{
        AdRotator1.AdCreated += new System.Web.UI.WebControls.AdCreatedEventHandler (Glob
al.GlobalAdCreated);
        AdRotator2.AdCreated += new System.Web.UI.WebControls.AdCreatedEventHandler (Glob
al.GlobalAdCreated);
}
```

See Also

Adding AdRotator Controls to a Web Forms Page | Introduction to the AdRotator Web Server Control | Tracking Ad Responses for the AdRotator Control

Tracking Ad Responses for the AdRotator Control

The **AdRotator** control does not inherently provide a way to gather statistics about which ads users have clicked, how many times an ad has been clicked, and so on. Ads displayed by the **AdRotator** control are effectively nothing more than hyperlinked images, with no special features for gathering statistics.

However, because you have control over the URL to which the ad points, you can add tracking logic to your application for the ads. A typical way to do this is to point all ads to a tracking page that first gathers the statistics you want, and then jumps to the ad's destination page.

Redirecting Ads to a Tracking Page

When you redirect to a tracking page, not only must you jump to the redirection page, but you must also be sure that the redirection page has the right information for gathering statistics. An easy way to do this is to include the necessary information in a query string.

> **Note** Query strings are appended to the URL of the redirection page and can be seen (and possibly manipulated) by users. If it is important that users not see this information (for example, for security reasons), you should encode the query string or use an alternative way to pass information about the ad to the redirection page. For information about persisting data between pages, see Web Forms State Management.

To redirect ads to a tracking page

- In the **NavigateURL** property of an individual ad, specify a URL that includes:
  - The name of your redirection page
  - Information that allows you to identify the ad
  - The page to which the ad ultimately points

  The following example shows an entry for a single ad from an XML ad list file. In this example, the redirection page is AdRedirector.aspx. The ad identifier and ultimate target page are specified as a query string.

  > **Note** Because the ampersand (&) is a reserved character in XML, the query string includes the entity reference
  > &
  > instead.

```
<Ad>
<ImageUrl>Widgets.gif</ImageUrl>
<NavigateUrl>AdRedirector.aspx?ad=Widgets&amp;target=http://msdn.microsoft.com/widgets/</
NavigateUrl>
<AlternateText>Microsoft Widgets</AlternateText>
<Keyword>computers</Keyword>
<Impressions>1</Impressions>
</Ad>
```

Gathering Statistics and Jumping to the Destination Page

The tracking page you specify performs two functions: it gathers statistics about the ads that users have clicked, then it redirects again to the ad's ultimate destination page. This generally means that the tracking page has no UI of its own.

How you gather statistics depends on your application requirements. As a minimum, you will probably count the number of times an ad has been clicked and store this information in a database.

To gather statistics and jump to the destination page

1. In the `Page_Load` handler of the tracking page, extract information about the ad that has been clicked and the destination page.
2. Store or otherwise process the ad information, as required by your application.
3. Call the Response.Redirect method to jump to the destination page.

   The following example shows a simple `Page_Load` handler that counts the number of times an ad has been clicked and stores the count in the Application state object, using the name of the ad as the key. (In your applications, you are likelier to

store this information in a database.) After updating the count, the code jumps to the destination page.

```
' Visual Basic
Private Sub Page_Load(ByVal sender As System.Object, _
      ByVal e As System.EventArgs) Handles MyBase.Load
   Dim source As String = Request.QueryString("ad")
   Dim redirect As String = Request.QueryString("target")
   Dim adcounter As Integer
   If Application(source) Is Nothing Then
      adcounter = 0
   Else
      adcounter = CType(Application(source), Integer)
   End If
   adcounter += 1
   Application(source) = adcounter
   Response.Redirect(redirect)
End Sub

// C#
private void Page_Load(object sender, System.EventArgs e)
{
   String source = Request.QueryString["ad"];
   String target = Request.QueryString["target"];
   int adcounter;
   if(Application[source] == null)
   {
      adcounter = 0;
   }
   else
   {
      adcounter = (int) Application[source];
   }
   adcounter += 1;
   Application[source] = adcounter;
   Response.Redirect(target);
}
```

See Also

Adding AdRotator Controls to a Web Forms Page | Creating an Ad List for the Ad Rotator Web Server Control | Selecting Ads in an AdRotator Web Server Control at Run Time

Button Web Server Controls

Buttons in a Web Forms page allow the user to send a command. They submit the form to the server and cause it to be processed along with any pending events.

Web server controls include three types of buttons: a standard command button (Button control), a hyperlink-style button (LinkButton control), and a graphical button (ImageButton control). All three provide similar features, but each offers a different look.

The following topics provide information that you will find useful in learning to work with the button Web controls.

In This Section

Introduction to the Button Web Server Controls
Provides background information on the types of buttons available among Web server controls and how you work with them.
Adding Button Web Server Controls to a Web Forms Page
Gives directions for placing buttons on a Web Forms page.
Creating Graphical Button Web Server Controls
Gives directions for displaying a graphic as a button.
Determining Coordinates in an ImageButton Web Server Control
Gives directions for using a graphical button as an image map.
Responding to Button Web Server Control Events
Gives directions for generating a response when users click a button.

Related Sections

Adding Web Server Controls to a Web Forms Page
Gives directions for including Web server controls on a form.
Setting ASP.NET Server Control Properties
Gives directions for setting Web server control properties at design time.
Setting Web Server Control Properties Programmatically
Gives directions for setting Web server control properties in code.
Customizing the Appearance of ASP.NET Server Controls Using Styles
Gives directions for changing the look of your Web server controls on the page.
Server Event Handling in Web Forms Pages
Gives directions for handling Web server control events.

Introduction to the Button Web Server Controls

Buttons on a Web Forms page allow users to indicate that they are finished with the form or want to perform a specific command. Web server controls include three kinds of buttons with different appearances.

Types of Buttons

You can create these types of Web server control buttons:

Control	Description
Button Web server control	Presents a standard command button, rendered as an HTML submit button.
LinkButton Web server control	Renders as a hyperlink in the page. However, it contains client-side script that causes the form to be posted back to the server. (You can create a true hyperlink using the HyperLink Web server control.)
ImageButton Web server control	Allows you to specify a graphic as a button. This is useful for presenting a rich button appearance. **ImageButton** controls also pinpoint where in the graphic a user has clicked, which allows you to use the button as an image map.

> **Note** You can also use the HtmlButton and HtmlInputButton controls to create buttons on the page that are programmable in server code. For details about the differences between HTML and Web server controls, see Introduction to ASP.NET Server Controls.

Button Events

All three types of buttons cause a form to be submitted to the server when the user clicks them. This causes the page to be processed and any pending events to be raised in server-based code. The buttons can also raise their own Click events that you can write event handlers for.

Buttons in Container Controls

Button Web server controls are often used in other controls, such as in the DataList, DataGrid, and Repeater list controls. When they are, you generally respond to their events differently than when they are standing alone on a form. When a user clicks a button in a container control, the event message is bubbled to the container control, where it raises a container-specific event. For example, in the **DataList** control, a button often raises the **DataList** control's ItemCommand event (rather than a **Button** control's Click event).

Because the list Web server controls can contain many different buttons, you can specify an argument to pass along with the event when the event is bubbled to the container control. You can then test for this argument to see which button was clicked.

Binding Data to the Controls

You can bind the Web server button controls to a data source in order to control their property settings dynamically. For example, you can set the buttons' Text property via data binding.

For details, see Data Binding Single-Value Web Server Controls at Design Time.

See Also

Adding Button Web Server Controls to a Web Forms Page | Responding to Button Web Server Control Events | Creating Graphical Button Web Server Controls

Adding Button Web Server Controls to a Web Forms Page

Adding buttons to a page is a simple process. Because buttons execute commands, it can be useful to define an access key for a button.

Adding an **ImageButton** control to a page is slightly more involved than working with other button types. For details, see Creating Graphical Button Web Server Controls.

To add button Web server controls to a Web Forms page

1. In **Design** view, drag a **Button** or **LinkButton** control from the **Web Forms** tab of the Toolbox onto the page.
2. Set the caption by setting the control's Text property.

To make it easier for users to click a button, you can define an access key for the button. Users can then press ALT plus the access key to choose the button.

> **Note** Access keys are not supported in all browsers.

To define a button's access key

- Set the AccessKey property to a single character. For example, to make the access key ALT+B, specify B as the value of the **AccessKey** property.

  > **Note** In Windows applications, access keys are typically indicated on a button using an underlined character. This marking is not available for button Web server controls due to restrictions in HTML. If you want to be able to display the access key for a button, you can use the HtmlButton control.

See Also

Setting Web Server Control Properties Programmatically | Responding to Button Web Server Control Events

Creating Graphical Button Web Server Controls

An **ImageButton** control allows you to present a picture as a clickable control.

To create a graphical button

1. In **Design** view, drag an **ImageButton** Web server control to the page.
2. Set the button's **ImageUrl** property to point to an image file (.gif, .jpg, and so on).

   You can point to an image file anywhere using a URL or UNC path. However, only files that are part of the project are deployed with the application. Therefore, it is recommended that you add the image file for your button to the project.

See Also

Determining Coordinates in an ImageButton Web Server Control

Determining Coordinates in an ImageButton Web Server Control

You can use an **ImageButton** Web Server control as an image map — that is, you can capture the precise x and y coordinates where a user clicked within the image. This allows you to perform different tasks based on where the user clicked.

Coordinate information is sent as part of the event-argument object for the **ImageButton** control's Click event.

To determine the coordinates where a user clicks

1. Create an event handler for the **ImageButton** control's **Click** event. The event-argument object for the method must be of type ImageClickEventArgs Class.
2. In the control's **Click** event handler, get the X and Y properties of the event-argument object.

   The x and y coordinates are expressed in pixels, where the upper left corner of the image is (0,0).

The following example shows how you can determine where the user clicked in a graphic that is 100 by 100 pixels. The code gets the x and y coordinates of the user's click. It then compares them against predetermined values to see if the user clicked in a particular quadrant. The results are displayed in a Label control.

```
' Visual Basic
Private Sub ImageButton1_Click(ByVal sender As System.Object, _
      ByVal e As System.Web.UI.ImageClickEventArgs) _
      Handles ImageButton1.Click
   Dim msg as String = ""
   Dim x As Integer = e.X
   Dim y As Integer = e.Y

   ' The button graphic is assumed to be 100x100 pixels.
   ' This checks coordinates against predetermined values that
   ' make up quadrants of the picture.
   If x >= 50 And y >= 50 Then
      msg = "Southeast"
   ElseIf x >= 50 And y < 50 Then
      msg = "Northeast"
   ElseIf x < 50 And y >= 50 Then
      msg = "Southwest"
   ElseIf x < 50 And y < 50 Then
      msg = "Northwest"
   End If
   Label1.Text = msg ' Display results
End Sub

// C#
private void ImageButton1_Click(object sender, System.Web.UI.ImageClickEventArgs e)
{
   // Label text
   string msg = "";
   // X coordinate from click argument
   int x = e.X;
   // Y coordinate from click argument
   int y = e.Y;

   // The button graphic is assumed to be 100x100 pixels.
   // This checks coordinates against predetermined values that
   // make up quadrants of the picture.
   if(x>=50 && y>= 50)
   {
   msg = "Southeast";
   }
   else if(x >=50 && y < 50)
   {
   msg = "Northeast";
   }
   else if(x < 50 && y >= 50)
```

```
    {
    msg = "Southwest";
    }
    else if(x < 50 && y < 50)
    {
    msg = "Northwest";
    }
    // Display results.
    Label1.Text = msg;
  }
```

See Also

Creating Graphical Button Web Server Controls

Responding to Button Web Server Control Events

When a **Button**, **LinkButton**, or **ImageButton** Web server control is clicked, the current page is submitted to the server, where it is processed.

To respond to a button event

- Create an event handler for one of the following events:
  - The page's **Page_Load** event. Because the button always posts the page to the server, this method will always run. Use the **Page_Load** event when it is not important which button was clicked, only that the form was submitted.
  - The button's Click event. Write an event handler for this event when it is important to know which button was clicked.

    > **Note** If you are working with an **ImageButton** control and want to know the x and y coordinates of the user's click, you must create an event handler for this event. For details, see Determining Coordinates in an ImageButton Web Server Control.

The following example shows how you can respond when a user clicks a **Button** Web server control. The method displays a message in a Label Web server control.

```
' Visual Basic
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Han
dles Button1.Click
   Label1.Text="You clicked a button"
End Sub

// C#
public void Button1_Click (object sender, System.EventArgs e)
{
   Label1.Text="You clicked a button.";
}
```

The following example shows how you can respond to a button click in a **Page_Load** event handler. The method tests the page's IsPostBack property to determine whether this is the first time the page has been processed or whether it was submitted by a button click.

```
' Visual Basic
Private Sub Page_Load(ByVal Sender As System.Object, ByVal e _
    As System.EventArgs) Handles MyBase.Load
    If Not IsPostback Then
        ' This is called the first time the page has loaded.
        ' The user will not have been able to click any buttons yet.
    Else
        ' This is called if the form has been posted back, possibly
        ' by a button click.
        Me.Label1.Text = "You clicked a button."
    End If
End Sub

// C#
private void Page_Load(object sender, System.EventArgs e)
{
   if (!IsPostBack)
   {
      // Evals true first time browser hits the page.
   }
   else
   {
```

```
      // This is called if the form has been posted back, possibly
      // by a button click.
      this.Label1.Text = "You clicked a button.";
   }
}
```

The following example shows a simple four-function integer calculator. By binding all the buttons (Add, Subtract, Multiply, and Divide) to the same method, you can handle all the calculations in one place and avoid repetitive code. Binding the buttons to the **Calculate** method is accomplished by using the **AddHandler** method in Visual Basic, and by using the += operator in C#. To ensure that the input values are integers, you could add error-handling code to the **Calculate** method or use the validation controls available for Web Forms.

```
' Visual Basic
' Set the CommandName property of the buttons to "Add",
' "Subtract", "Multiply", and "Divide".
Private Sub Page_Load(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles
MyBase.Load
   AddHandler Me.btnAdd.Click, AddressOf Calculate
   AddHandler Me.btnSubtract.Click, AddressOf Calculate
   AddHandler Me.btnMultiply.Click, AddressOf Calculate
   AddHandler Me.btnDivide.Click, AddressOf Calculate
End Sub

Public Sub Calculate(ByVal sender As Object, ByVal e As System.EventArgs)
   Dim op1 As Integer = CType(Me.TextBox1.Text, Integer)
   Dim op2 As Integer = CType(Me.TextBox2.Text, Integer)
   Dim result As Integer
   Select Case CType(sender, Button).CommandName
      Case "Add"
         result = op1 + op2
      Case "Subtract"
         result = op1 - op2
      Case "Multiply"
         result = op1 * op2
      Case "Divide"
         ' Divide two numbers and return an integer result.
         If op2 > 0 Then
               result = op1 \ op2
         Else
               result = 0
         End If
      Case Else
         ' Error handling code here.
   End Select
   Label1.Text = result.ToString()
End Sub

// C#
// Set the CommandName property of the buttons to "Add", _
// "Subtract", "Multiply", and "Divide".
protected void Page_Load(object sender, EventArgs e)
{
   btnAdd.Click += new System.EventHandler(this.Calculate);
   btnSubtract.Click += new System.EventHandler(this.Calculate);
   btnMultiply.Click += new System.EventHandler(this.Calculate);
   btnDivide.Click += new System.EventHandler(this.Calculate);
}
```

```
protected void Calculate (object sender, System.EventArgs e)
{
   int op1 = Convert.ToInt16(TextBox1.Text);
   int op2 = Convert.ToInt16(TextBox2.Text);
   int result = 0;
   switch(((Button)sender).CommandName)
   {
      case "Add" :
         result = op1 + op2;
         break;
      case "Subtract" :
         result = op1 - op2;
         break;
      case "Multiply" :
         result = op1 * op2;
         break;
      case "Divide" :
         // Integer division.
         if (op2 > 0)
               result = op1 / op2;
         else
               result = 0;
         break;
      default:
         // Error handling code here.
         break;
   }
   Label1.Text = result.ToString();
}
```

See Also

ASP.NET Server Control Event Model

Calendar Web Server Control

The Calendar Web server control displays a traditional one-month calendar on your Web Forms pages. Users can use the calendar to view and select dates.

The following topics provide information about the Calendar Web server control and how to use it.

In This Section

Introduction to the Calendar Web Server Control
Provides a general description of the **Calendar** control.
Controlling Month Navigation in a Calendar Web Server Control
Gives directions for specifying whether users can move from month to month.
Controlling User Date Selection in a Calendar Web Server Control
Gives directions for specifying whether users can select a day, a week, or the entire month.
Customizing Calendar Web Server Control Appearance
Gives directions for changing the fonts, colors, size, and other appearance features of the calendar as a whole.
Customizing Individual Days in a Calendar Web Server Control
Gives directions for changing the content or look of individual days programmatically, for example, by creating appointment information.
Formatting Web Calendar Elements Using Styles
Gives directions for customizing the look of specific areas of the calendar, such as today's date, weekend days, and the title bar.
Responding to Date Selection in a Calendar Web Server Control
Gives directions for running code in response to the user selecting one or more dates.
Selecting Dates Programmatically in a Calendar Web Server Control
Gives directions for setting and clearing selected dates in code.
Setting Today's Date Programmatically in a Calendar Web Server Control
Gives directions for adjusting the current date to accommodate different time zones on the server and client.

Related Sections

Adding Web Server Controls to a Web Forms Page
Gives directions for including controls on your Web Form.
Setting ASP.NET Server Control Properties
Suggests additional topics of interest concerning server control properties.
Setting Web Server Control Properties Programmatically
Gives directions for setting server control properties in code.
Customizing the Appearance of ASP.NET Server Controls Using Styles
Gives directions for applying styles to change how controls are displayed.

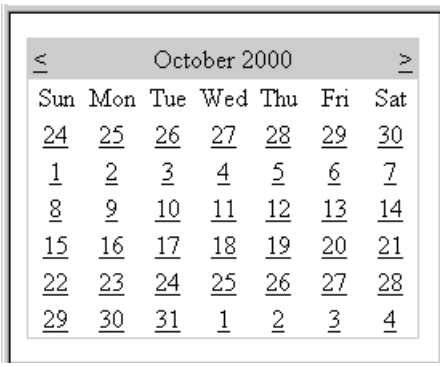
Introduction to the Calendar Web Server Control

The **Calendar** Web server control can serve these purposes:

- **Display and select dates** The control displays a calendar through which users can navigate to any day in any year. Setting the SelectedDate Property property causes a specific date to be highlighted in the control. Optionally, users can move to arbitrary dates by clicking a day or moving from month to month. The calendar can be configured to allow users to select multiple dates, either an entire week or an entire month.

  Typical uses are to graphically display the current date on a daily news Web page, or to allow users to select dates in scheduling applications to order airline reservations, concert tickets, and so on.

  A representation of the Calendar control displaying the month of October

- **Display appointment or other information in a calendar grid** The **Calendar** control can display day-specific details for individual days, such as a to-do list, a schedule of events, or similar information. This feature allows you to display day information from a database.

The **Calendar** control is based around the .NET framework DateTime object, and therefore supports the full range of dates allowed by that object. Effectively, you can display any date between the years 0 and 9999 A.D.

> **Note** In Visual Basic the **Date** data type is equivalent to the .NET framework's DateTime data type.

The **Calendar** control is rendered as an HTML table when the Web Forms page runs. Therefore, a number of the control's properties pertain to various aspects of table formatting. A few of these properties are not fully supported in some older browsers, so some formatting features will not be available in those browsers.

Developing with the Calendar Web Server Control

The following sections contain details about how the **Calendar** control works and an overview of tasks you can accomplish with it.

Displaying and Selecting Dates

The **Calendar** control displays a month of dates at a time, with a total of six weeks appearing at once. Within the control, there are several types of dates that you can work with:

Type of date	Description
Today's date	By default, the current date on the server. However, you can adjust it so that the date appears correctly for a different locale.
Visible date	This date determines what month appears in the calendar. The user can move from month to month on the calendar, thereby changing the visible date without affecting today's date. You can navigate between months by setting the visible date programmatically.
Selected date or dates	This is the date or date range that the user chooses. In the control, the user can choose a single day, week, or month, but can only select contiguous dates. You can also programmatically set the selected dates; in that case, you can set non-contiguous selected dates.

Enabling Date Selection

By default, the calendar allows users to click an individual date to select it. If you are using the control as a read-only calendar, you can disable all date selection.

If day selection is enabled, each day of the calendar contains a LinkButtoncontrol that raises an event when clicked. If you enable week or month selection, a column of links is added to the left side of the calendar to enable the user to specify what week to select.

Customizing the Calendar's Appearance

You can set calendar properties to change the colors, size, text, and other visual features of the calendar. There are a number of ways to do this:

Method	Description
Setting properties	You can set properties to display grid lines, change what day is displayed as the first day of the week, and change the appearance of the month and day names.
Setting extended style properties	You can use properties derived from the Style object to set the appearance of particular elements in the calendar, such as the current date or the title bar containing the month and navigation links. These style properties are supported in browsers that can use cascading style sheets; a reduced set of appearance styles is supported for earlier browsers.
Customizing the rendering of individual days	As the control renders individual days, it raises an event you can handle to modify the stream being rendered to the browser. This is useful not just for changing the appearance of days, but for including custom content on each day as well.

Capturing User Interaction with the Calendar Web Server Control

The **Calendar** control raises an event when the user selects an individual date or range of dates or navigates to a new month. By creating methods for these events, you can determine what day or dates the user has selected and respond appropriately. One response may be to customize the display of that date. In creating a custom display, you can include any HTML elements in the calendar, including other server controls.

Displaying Information from a Database in the Calendar Control

A common scenario is to display information from a database in the calendar. For example, an events calendar is often based out of information kept in a database.

The **Calendar** control does not directly support databinding — that is, you do not bind the calendar as a whole to a data source. Instead, you create a method for the control's **DayRender** event, which is raised as each day in the current calendar month (view) is being rendered. In the method for this event, you can extract information from a data source and add it to the stream being sent to the browser.

See Also

Customizing Calendar Web Server Control Appearance | Setting Web Server Control Properties Programmatically | Customizing the Appearance of ASP.NET Server Controls Using Styles

Controlling Month Navigation in a Calendar Web Server Control

By default, the calendar displays the month containing today's date. Users can move from month to month in the **Calendar** control by clicking month navigation links (actually, LinkButton Web server controls) in the title bar of the calendar. You can control whether these links are available for the user to click.

You can also navigate programmatically, which is useful if you want to provide an alternate way for users to display a specific month. Finally, you can respond to an event raised when the user navigates.

To enable or disable user month navigation

- Set the **Calendar** control's ShowNextPrevMonth property to **true** or **false**. If **false**, the control does not display the **LinkButton** controls in the title that allow a user to move between months.

  Tip You can change the appearance of the month navigation hyperlinks by setting the NextMonthText, PrevMonthText, NextPrevFormat, and NextPrevStyle properties.

To navigate months programmatically

- Set the control's VisibleDate property to a date in the month you want to display. The date can be any day within that month, but is typically set to the first day of the month. If the **VisibleDate** property is empty (it is set to **DateTime.Empty**), the currently visible month is derived from the value of the TodaysDate property.

  The following example navigates to February, 2000:

  ```
  ' Visual Basic
  Calendar1.VisibleDate = New Date(2000, 2, 1)

  // C#
  Calendar1.VisibleDate = new DateTime(2000, 2, 1);
  ```

  Changing the **VisibleDate** property has no effect on the values of the **TodaysDate**, SelectedDayStyle, or SelectedDates properties.

If month navigation is enabled, the control raises an event when the user moves to another month. You can handle this event in order to replace or amend the default month navigation. For example, if you are using two **Calendar** controls in a trip-planning page, you can prevent the user from setting the start-date month later than the end-date month.

To respond to a month navigation event

- Create a method for the control's VisibleMonthChanged event. When this event is raised, the control has already changed the **VisibleMonth** property by one.

  The **VisibleMonthChanged** event takes a single argument of type **MonthChangedEventArgs**. You can use the following properties of this argument to determine or override what the user is doing:

Property	Description
PreviousDate	The value of the month that was displayed before the user clicked a month-navigation button. You can compare the value of this property to that of the control's **VisibleMonth** property to determine what direction the user is navigating. To cancel the effect of the user's click, set the control's **VisibleDate** property to this value.
NewDate	The value of the month that the user has navigated to. The **VisibleDate** property is updated to this value before the **VisibleMonthChanged** event is raised.

```
' Visual Basic
Public Sub Calendar1_VisibleMonthChanged(ByVal sender As Object, _
   ByVal e As System.Web.UI.WebControls.MonthChangedEventArgs) _
   Handles Calendar1.VisibleMonthChanged

   ' Select the 10th and 25th of each month.
   Calendar1.SelectedDates.Clear()
   Calendar1.SelectedDates.Add(New DateTime(e.NewDate.Year, e.NewDate.Month, 10))
```

```
   Calendar1.SelectedDates.Add(New DateTime(e.NewDate.Year, e.NewDate.Month, 25))
End Sub

// C#
private void Calendar1_VisibleMonthChanged (object sender,
   System.Web.UI.WebControls.MonthChangedEventArgs e)
{
     // Select the 10th and 25th of each month.
     Calendar1.SelectedDates.Clear();
     Calendar1.SelectedDates.Add(new DateTime(e.NewDate.Year, e.NewDate.Month, 10));
     Calendar1.SelectedDates.Add(new DateTime(e.NewDate.Year, e.NewDate.Month, 25));
}
```

See Also

Controlling User Date Selection in a Calendar Web Server Control |
Responding to Date Selection in a Calendar Web Server Control

Controlling User Date Selection in a Calendar Web Server Control

You can specify that users can select individual days, individual weeks, or an entire month in the **Calendar** control. Alternatively, you can disable day selection altogether.

To control user date selection

- Set the **Calendar** control's SelectionMode property to one of the values defined in the **SelectionMode** enumeration: **Day**, **DayWeek**, or **DayWeekMonth**. To disable all date selection, set the property to **None**.

  > **Note** Users can only select a single day, a single week, or the whole month. However, in code you can select any arbitrary set of dates. For details, see Selecting Dates Programmatically in a Calendar Web Server Control.

When selection is enabled, whenever the user selects a day, week, or month, the form is submitted and an event is raised for which you can create a method. For details, see Responding to Date Selection in a Calendar Web Server Control.

To allow the user to select a date, the calendar is displayed with links. Individual days contain a link with the date number. If you set the calendar to allow users to select a week or the whole month, an extra column with selection links is added to the left of the calendar.

> **Note** You can also control date selection for individual days. For example, you might specify that users can click specific days of the month, such as holidays. For details about customizing days in this way, see Customizing Individual Days in a Calendar Web Server Control.

You can specify the text for the week and month selection links.

To specify week and month selection link text

- Set the SelectWeekText and SelectMonthText properties to the text you want to use as a navigation link.

  You can also set the font, color, and border style of the column containing the week and month selection links and set styles for days. For details, see Formatting Web Calendar Elements Using Styles.

See Also

Controlling Month Navigation in a Calendar Web Server Control | Responding to Date Selection in a Calendar Web Server Control

Customizing Calendar Web Server Control Appearance

Because the **Calendar** control is a complex control made up out of many individual elements, it offers you a wide variety of options for customizing its appearance. In this topic, you can learn about the following customization options:

- Changing General Calendar Appearance
- Changing Day-of-Week Display
- Changing the Month Display and Navigation Links

Changing General Calendar Appearance

You can change the overall appearance of the **Calendar** control, which establishes a default look for the calendar on the page. If you set appearance properties for individual calendar elements, such as today's date, the selected date, and so on, the individual settings are based on the default settings you make for the calendar as a whole. You can also select from predefined Calendar formats.

> **Note** By default, there is no special formatting to indicate today's date on the **Calendar** control. For details about how to set this format, see Formatting Web Calendar Elements Using Styles.

To change the general appearance of the Calendar control

- Set properties as listed in the following table.

To specify	Set this property
Font, text and background color	Font, ForeColor, BackColor
Overall control size	Height, Width
Margins and spacing around individual days	CellPadding, CellSpacing
Color, size, and style (for example, dotted) of bor der and grid lines	• BorderColor • BorderStyle (set to a **BorderStyle** enumeration value: **Dotted**, **D ashed**, **Solid**, and so on). • ShowGridLines (true or false)

To make it easier to establish the appearance of the **Calendar** control, you can select from a variety of predefined Calendar formats.

To select a predefined Calendar control format

1. In **Design** view, select the **Calendar** control.
2. At the bottom of the Properties window, click the **AutoFormat** link to display the **Calendar Auto Format** dialog box.
3. Select a format from the **Name** list.

   You can further customize the predefined format by changing the properties set by the automatic format process.

Changing Day-of-Week Display

By default, the names of the days of the week are displayed in a row at the top of the calendar. You can control the format of this row.

To change the day of week display

- Set properties as listed in the following table.

To	Set this property
Enable or disable the row that deploys days of t he week	ShowDayHeader (true or false)
Specify the format of day names (for example, f ull day name or abbreviation)	DayNameFormat (use a value in the DayNameFormat enumeration: **Full**, **S hort**, **FirstLetter**, **FirstTwoLetters**).
Specify which day starts the week (default is Su nday)	FirstDayOfWeek (use a day defined in the **FirstDayOfWeek** enumeration.
Set color, font, border style and width, and so o n	DayHeaderStyle, which incorporates style information. For details, see Formatting Web Calendar Elements Using Styles.

Changing the Month Display and Navigation Links

The title bar by default displays the name of the current month and a set of navigation links (LinkButton controls) that allow users to move to the next and previous months.

To change month display and navigation links

- Set properties as listed in the following table.

To	Set this property
Remove the title bar altogether (which disables month navigation).	ShowTitle (true or false)
Specify the format of the name of the month (for example, full name or abbreviation) and optionally add a year.	• **MonthNameFormat** (use a value in the **MonthNameFormat** enumeration: **Full** or **Short**). • TitleFormat (set to a value in the **TitleFormat** enumeration: **Month** or **MonthYear**).
Specify the format and text of month navigation links	• NextPrevFormat (use a value in the **NextPrevFormat** enumeration: **Month** or **CustomText**). • NextMonthText and PrevMonthText (set to strings)
Set color, font, border style and width, and so on	TitleStyle, which incorporates style information. For details, see Formatting Web Calendar Elements Using Styles. **Note** You can format the navigation links separately using the **NextPrevStyle** property.

Note You can use HTML tags within the property string.

See Also

Formatting Web Calendar Elements Using Styles | Controlling User Date Selection in a Calendar Web Server Control

Customizing Individual Days in a Calendar Web Server Control

By default, days in the **Calendar** control are simply displayed as numbers. (If day selection is enabled, the numbers appear as links. For details, see Controlling User Date Selection in a Calendar Web Server Control.) However, you can customize the appearance and content of individual days, which allows you to:

- Programmatically highlight certain days, for example, showing holidays in a different color.
- Programmatically specify whether an individual day can be selected.
- Add information into the day display, such as appointment or event information.

When the **Calendar** control is creating the output to send to the browser, it raises a DayRender event you can handle. The control calls your method for each day as it is preparing the day for display, and you can then programmatically examine which date is being rendered and customize it appropriately.

The **DayRender** event method takes two arguments, a reference to the control raising the event (the **Calendar** control) and an object of type **DayRenderEvent**. The **DayRenderEvent** object provides access to two additional objects:

- **Cell**, a TableCell object that you can use to set the appearance of an individual day.
- **Day**, which you can use to query information about the day being rendered, control whether the day can be selected, and add content to a day. The **Day** object supports various properties you can use to learn about the day (for example, **IsSelected**, **IsToday**, and so on). It also supports a **Controls** collection that you can manipulate to add content to the day.

To customize the appearance of an individual day

1. Create a method for the **Calendar** control's **DayRender** event. The event should have the following signature:

```
' Visual Basic
Private Sub Calendar1_DayRender(ByVal sender As Object, ByVal e As System.Web.UI.WebContr
ols.DayRenderEventArgs) Handles Calendar1.DayRender
' Additional code here
End Sub

// C#
private void Calendar1_DayRender (object sender, System.Web.UI.WebControls.DayRenderEvent
Args e)
{
   // Additional code here
}
```

2. In the method, set properties of the **Cell** object available via the DayRenderEvent argument, as in the following example:

```
' Visual Basic
   If (e.Day.IsToday) Then
      e.Cell.BackColor = System.Drawing.Color.Red
   End If

// C#
   if (e.Day.IsToday)
   {
      e.Cell.BackColor = System.Drawing.Color.Red;
   }
```

The following example shows a simple but complete method that illustrates how to change the appearance of individual days. The method causes vacation days in the calendar to be rendered in yellow, while weekend days are rendered in green.

```
' Visual Basic
Public Sub Calendar1_DayRender(ByVal sender As Object, ByVal e As System.Web.UI.WebControls.D
ayRenderEventArgs) Handles Calendar1.DayRender
   ' Display vacation dates in yellow boxes with purple borders.
   Dim vacationStyle As New Style()
```

```
    With vacationStyle
       .BackColor = System.Drawing.Color.Yellow
       .BorderColor = System.Drawing.Color.Purple
       .BorderWidth = New Unit(3)
    End With

    ' Display weekend dates in green boxes.
    Dim weekendStyle As New Style()
    weekendStyle.BackColor = System.Drawing.Color.Green

    ' Vacation is from Nov 23, 2000 to Nov 30, 2000.
    If ((e.Day.Date >= New Date(2000, 11, 23)) _
       And (e.Day.Date <= New Date(2000, 11, 30))) Then
       e.Cell.ApplyStyle(vacationStyle)
    ElseIf (e.Day.IsWeekend) Then
       e.Cell.ApplyStyle(weekendStyle)
    End If
 End Sub

 // C#
 private void Calendar1_DayRender (object sender, System.Web.UI.WebControls.DayRenderEventArgs
 e)
 {
    // Display vacation dates in yellow boxes with purple borders.
    Style vacationStyle = new Style();
    vacationStyle.BackColor = System.Drawing.Color.Yellow;
    vacationStyle.BorderColor = System.Drawing.Color.Purple;
    vacationStyle.BorderWidth = 3;

    // Display weekend dates in green boxes.
    Style weekendStyle = new Style();
    weekendStyle.BackColor = System.Drawing.Color.Green;


    if ((e.Day.Date >= new DateTime(2000,11,23)) &&
       (e.Day.Date <= new DateTime(2000,11,30)))
    {
       // Apply the vacation style to the vacation dates.
       e.Cell.ApplyStyle(vacationStyle);
    }
    else if (e.Day.IsWeekend)
    {
       // Apply the weekend style to the weekend dates.
       e.Cell.ApplyStyle(weekendStyle);
    }
 }
```

To specify that an individual day can be selected

- In a method for the **Calendar** control's **DayRender** event, determine what day is being rendered by getting information from the **Day** object in the **DayRenderEvent** argument. Specifically, query the **Date** property of the **Day** object. Then set that day's **IsSelectable** property to true. In the following example, the date September 10, 2001 is made selectable; all other dates are made not selectable.
- July 4, 2000 is made selectable if it is one of `myHoliday`'s days.

```
 ' Visual Basic
 Public Sub Calendar1_DayRender(ByVal sender As Object, _
 ByVal e As System.Web.UI.WebControls.DayRenderEventArgs) _
 Handles Calendar1.DayRender
    Dim myAppointment As Date = New Date(2000, 9, 10)
    If (e.Day.Date = myAppointment) Then
       e.Day.IsSelectable = True
    Else
       e.Day.IsSelectable = False
    End If
```

```
End Sub

// C#
private void Calendar1_DayRender(object sender, System.Web.UI.WebControls.DayRenderEventA
rgs e)
{
   DateTime myAppointment = new DateTime(2001,9,10);
   if (e.Day.Date == myAppointment)
   {
      e.Day.IsSelectable = true;
   }
   else
   {
      e.Day.IsSelectable = false;
   }
}
```

To add content to an individual day

- In a method for the **Calendar** control's **DayRender** event, add any valid Web Forms page element (including HTML) to the **Controls** collection of the **Day** object from the **DayRenderEvent** argument.

  The following example displays holidays. The holidays are created as a two-dimensional array during page load; holiday descriptions are loaded into the elements corresponding to their date. In the **DayRender** event method, each day is compared against the holiday array. If the corresponding element of the holiday array contains something, a label control is created with the holiday text and added into the **Controls** collection for that day.

```
' Visual Basic
' Add to the declarations section of the form class.
Dim holidays(13, 32) As String

' Add to form load.
holidays(1, 1) = "Birthday"
holidays(2, 14) = "Anniversary"

Public Sub Calendar1_DayRender(ByVal sender As Object, _
   ByVal e As System.Web.UI.WebControls.DayRenderEventArgs) _
   Handles Calendar1.DayRender

   If e.Day.IsOtherMonth Then
      e.Cell.Controls.Clear()
   Else
      Dim aDate As Date = e.Day.Date
      Dim aHoliday As String = holidays(aDate.Month, aDate.Day)
      If (Not aHoliday Is Nothing) Then
         Dim aLabel As Label = New Label()
         aLabel.Text = "<br>" & aHoliday
         e.Cell.Controls.Add(aLabel)
      End If
   End If
End Sub

// C#
// Add this to the class members of the form.
string[,] holidays = new String[13,32];

// Add this to the Page_Load event.
holidays[1,1] = "Birthday";
holidays[2,14] = "Anniversary";
```

```
// Add this to the DayRender event.
string aHoliday;
DateTime theDate = e.Day.Date;
aHoliday = holidays[theDate.Month,theDate.Day];
if (aHoliday != null)
{
   Label aLabel = new Label();
   aLabel.Text = " <br>" + aHoliday;
   e.Cell.Controls.Add(aLabel);
}
```

See Also

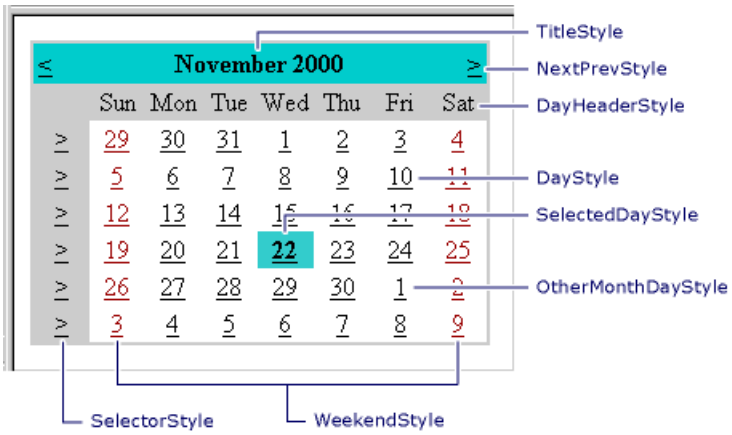
Customizing Calendar Web Server Control Appearance | Controlling User Date Selection in a Calendar Web Server Control | Controlling Month Navigation in a Calendar Web Server Control

Formatting Web Calendar Elements Using Styles

The **Calendar** control, like all Web server controls, allows you to use styles to specify the appearance of the control, including aspects such as font, color, borders, and so on. For general information about using styles with Web server controls, see Customizing the Appearance of ASP.NET Server Controls Using Styles.

> **Note** Styles are not fully supported in all browsers.

To allow you flexibility in customizing the appearance of the calendar output, the **Calendar** control supports styles for a number of discrete elements that make up the calendar grid. The following diagram displays a calendar with all optional elements displayed.



The following table describes each of the elements that you can control using styles.

Calendar Element	Description
DayStyle	Days of the current month. Weekends, today, and the selected day can have different styles (see below).
DayHeaderStyle	Row above the calendar where the names of the days appear.
NextPrevStyle	Sections at the left and right ends of the title bar where the month navigation LinkButtons are.
OtherMonthDayStyle	Days from the previous and next month that appear on the current month view.
SelectedDayStyle	The day selected by the user.
SelectorStyle	The column to the left containing links for selecting a week or the whole month.
TitleStyle	Title bar at the top of the calendar containing the month name and month navigation links. If **NextPrevStyle** is set, it overrides the extreme ends of the title bar.
TodayDayStyle	Today's date.
WeekendDayStyle	Weekend days.

To set calendar element styles in the Web Forms Designer

1. In **Design** view, select the **Calendar** control.
2. In the Properties window, locate the style element to set, such as **DayStyle**, and then expand it to display all style settings.

   > **Tip** If you display properties by category, all the style elements appear together under **Style**.

3. Change the style settings you want to change. Many style settings, such as color and border settings, allow you to select settings from a drop-down list.

You can also set style elements at run time in code.

To set calendar element styles programmatically

- Set properties of individual style elements using the following syntax:

  ```
  Calendar1.StyleName.Property = setting
  ```

  The following examples illustrate some typical style settings for the **Calendar** control:

  ```
  ' Visual Basic
  Calendar1.DayStyle.Font.Size = New FontUnit(10)
  ```

```
Calendar1.TitleStyle.ForeColor = System.Drawing.Color.Red

// C#
Calendar1.DayStyle.Font.Size = new FontUnit(10);
Calendar1.TitleStyle.ForeColor = System.Drawing.Color.Red;
```

Note For details about setting properties, see Setting ASP.NET Server Control Properties.

See Also

Customizing the Appearance of ASP.NET Server Controls Using Styles |
Customizing Individual Days in a Calendar Web Server Control

Responding to Date Selection in a Calendar Web Server Control

If the **Calendar** control's SelectionMode property is set to anything other than **None**, the user can select a day or a range of dates. You can detect and respond to the user's choice.

To respond to a date selection

- Create a method for the **Calendar** control's SelectionChanged event with the following signature:

```
' Visual Basic
Private Sub Calendar1_SelectionChanged(ByVal sender As System.Object, ByVal e As System.E
ventArgs) Handles Calendar1.SelectionChanged

// C#
private void Calendar1_SelectionChanged (object sender, System.EventArgs e)
```

> **Note** The event is raised only if the date selection is changed by user action in the control. For instance, if the user clicks the same date twice, the second click does not raise an event. The event is also not raised if you set a date range programmatically.

Information about date selection is available in these properties:

Property	Description
SelectedDate	A single date. If the user has selected a single date, this property contains that date. If the user has selected multiple dates, this property contains the first date in the range.
SelectedDates	A collection containing all the dates selected. The dates in this collection are ordered and unique. Because the **Calendar** control does not allow the user to select multiple individual dates, the dates in the collection are also sequential.

To determine how many dates are selected

- Get the value of the **Count** property of the **SelectedDates** collection, as shown in the following example.

```
' Visual Basic
Public Sub Calendar1_SelectionChanged(ByVal sender As Object, _
   ByVal e As System.EventArgs) _
   Handles Calendar1.SelectionChanged

   Text1.Text = "You selected " _
      & Calendar1.SelectedDates.Count.ToString() _
      & " date(s)."
End Sub

// C#
private void Calendar1_SelectionChanged (object sender, System.EventArgs e)
{
   Text1.Text = String.Format("You selected {0} date(s).",
      Calendar1.SelectedDates.Count);
}
```

If you've determined that the user has selected multiple dates, you can get the range.

To get the date range of a multi-date selection

1. Get the count of selected dates using the Count property of the **SelectedDates** property.
2. Get the first date in the collection, and then get the last date by extracting the date at the index of the count -1. The following example displays the first and last dates in TextBox controls on the page.

```
' Visual Basic
Public Sub Calendar1_SelectionChanged(ByVal sender As Object, _
   ByVal e As System.EventArgs) _
```

```
   Handles Calendar1.SelectionChanged

   With Calendar1.SelectedDates
      If .Count > 1 Then
         Dim firstDate As Date = .Item(0).Date
         Dim lastDate As Date = .Item(.Count - 1).Date
         TextBox1.Text = firstDate.ToString()
         TextBox2.Text = lastDate.ToString()
      End If
   End With
End Sub

// C#
private void Calendar1_SelectionChanged (object sender,
   System.EventArgs e)
{
   SelectedDatesCollection theDates = Calendar1.SelectedDates;
   if (theDates.Count > 1)
   {
      DateTime firstDate = theDates[0];
      DateTime lastDate = theDates[theDates.Count-1];
      TextBox1.Text = firstDate.ToString();
      TextBox2.Text = lastDate.ToString();
   }
}
```

To get the time span of a multi-date selection

- Create a TimeSpan object and set it to the difference between the last and first dates in the **SelectedDates** collection, and then get the **TimeSpan** object's Days property.

```
' Visual Basic
Public Sub Calendar1_SelectionChanged(ByVal sender As Object, _
   ByVal e As System.EventArgs) Handles Calendar1.SelectionChanged
   With Calendar1.SelectedDates
      Dim days As Integer
      Dim firstDate As Date = .Item(0).Date
      Dim lastDate As Date = .Item(.Count - 1).Date
      ' The Subtract method returns a TimeSpan object.
      days = lastDate.Subtract(firstDate).Days + 1
      TextBox1.Text = "You have selected " & days.ToString() & " day(s)."
   End With
End Sub

// C#
private void Calendar1_SelectionChanged (object sender,
   System.EventArgs e)
{
   SelectedDatesCollection theDates = Calendar1.SelectedDates;
   TimeSpan timeSpan = theDates[theDates.Count-1] - theDates[0];
   TextBox1.Text =
      String.Format("You have selected {0} day(s).", timeSpan.Days + 1);
}
```

See Also

Setting Today's Date Programmatically in a Calendar Web Server Control |
Selecting Dates Programmatically in a Calendar Web Server Control

Selecting Dates Programmatically in a Calendar Web Server Control

You can set date selections in your own code, either a single date or a range of dates. In contrast to user selection in the control on a page, you can select multiple non-sequential dates in code.

Note Setting a date programmatically does not raise the SelectionChanged event.

To select a single date

- Set the control's SelectedDate property to an expression of type **DateTime**.

```
' Visual Basic
Calendar1.SelectedDate = Date.Today

// C#
Calendar1.SelectedDate = DateTime.Today;
```

Note If you set the **SelectedDate**, all the dates in SelectedDates are effectively cleared.

To select a range of dates

- Call the **Add** method of the control's **SelectedDates** collection. You can add dates in any order, because the collection will order them for you. The collection also enforces uniqueness and will therefore ignore a date you add if the date is already in the collection.

  The following example sets the selection to every Wednesday in the month of February, 2000.

```
' Visual Basic
Private Sub Button1_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles Button1.Click

   With Calendar1.SelectedDates
      .Clear()
      .Add(New Date(2000, 2, 2))
      .Add(New Date(2000, 2, 9))
      .Add(New Date(2000, 2, 16))
      .Add(New Date(2000, 2, 23))
   End With
End Sub

// C#
public void Button1_Click (object sender, System.EventArgs e)
{
   SelectedDatesCollection theDates = Calendar1.SelectedDates;
   theDates.Clear();
   theDates.Add(new DateTime(2000,2,2));
   theDates.Add(new DateTime(2000,2,9));
   theDates.Add(new DateTime(2000,2,16));
   theDates.Add(new DateTime(2000,2,23));
}
```

  The following example selects a sequence of seven dates.

```
'Visual Basic
Dim today As Date = Date.Today
Dim i As Integer
```

```
With Calendar1.SelectedDates
   .Clear()
   For i = 0 To 6
      .Add(today.AddDays(i))
   Next
End With

// C#
DateTime aDate = DateTime.Today;
SelectedDatesCollection theDates = Calendar1.SelectedDates;
theDates.Clear();
for (int i = 0;i <= 6;i++)
{
      theDates.Add(aDate.AddDays(i));
}
```

To clear a date selection

- Call the **Clear** method of the control's **SelectedDates** collection, as in the following example:

```
' Visual Basic

Calendar1.SelectedDates.Clear()

// C#

Calendar1.SelectedDates.Clear();
```

See Also

Setting Today's Date Programmatically in a Calendar Web Server Control |
Responding to Date Selection in a Calendar Web Server Control

Setting Today's Date Programmatically in a Calendar Web Server Control

By default, the value of "today" in the **Calendar** control is set to match the date of the server on which the Web Forms page runs. However, you might need to adjust the date to accommodate users who are viewing the page from a different time zone.

To set today's date programmatically

- Set the **Calendar** control's TodaysDate property to a **DateTime** value.

  The following example sets **TodaysDate** to tomorrow and then set the **SelectedDate** to **TodaysDate**. In the browser, the date for tomorrow will be highlighted.

```
' Visual Basic
Dim tomorrow As Date = Date.Today.AddDays(1)
Calendar1.TodaysDate = tomorrow
Calendar1.SelectedDate = Calendar1.TodaysDate

// C#
DateTime tomorrow = DateTime.Today.AddDays(1);
Calendar1.TodaysDate = tomorrow;
Calendar1.SelectedDate = Calendar1.TodaysDate;
```

  The following example shows how you can prompt the user for the correct local date and then use that information to adjust the value of today's date.

```
' Visual Basic
' Add this the declarations section of the page class.
Dim today As Date
Dim tomorrow As Date
Dim yesterday As Date
' Add this to the form Load method.
' The user then selects the current date from the DropDownList.
If Not IsPostback Then
   today = System.Date.Today
   yesterday = Today.AddDays(-1)
   tomorrow = Today.AddDays(1)
   DropDownList1.items.Add(String.Format("{0:dd MMM yyyy}", _
      today))
   DropDownList1.items.Add(String.Format("{0:dd MMM yyyy}", _
      yesterday))
   DropDownList1.items.Add(String.Format("{0:dd MMM yyyy}", _
      tomorrow))
End If

' Add this code to set TodaysDate to the users selected date.
Calendar1.TodaysDate = Date.Parse(DropDownList1.SelectedItem.Text)

// C#
// Add this to the declarations section of the page class.
DateTime today;
DateTime yesterday;
DateTime tomorrow;

// Add this to the Page_Load method.
if (!IsPostBack)
{
   today = System.DateTime.Today;
```

```
    yesterday = today.AddDays(-1);
    tomorrow = today.AddDays(1);
    DropDownList1.Items.Add(String.Format("{0:dd MMM yyyy}",
       today));
    DropDownList1.Items.Add(String.Format("{0:dd MMM yyyy}",
       yesterday));
    DropDownList1.Items.Add(String.Format("{0:dd MMM yyyy}",
       tomorrow));
}

// Add this code to set TodaysDate to the users selected date.
Calendar1.TodaysDate = DateTime.Parse(DropDownList1.SelectedItem.Text);
```

See Also

Selecting Dates Programmatically in a Calendar Web Server Control |
Responding to Date Selection in a Calendar Web Server Control

CheckBox and CheckBoxList Web Server Controls

The **CheckBox** and **CheckBoxList** Web server controls provide a way for users to switch between true/false, yes/no, or on/off options.

Because there are two different controls, and because they function somewhat differently, it's important to understand their different purposes. You can refer to the following table to determine which topics pertain specifically to the control you want to work with.

> **Note** Some of the topics listed below for the **CheckBoxList** control do not pertain solely to that control; instead, they are topics that apply to all list Web server controls (ListBox, DropDownList, RadioButtonList, and CheckBoxList).

In This Section

Introduction to the CheckBox and CheckBoxList Web Server Controls
Explains what you can do with **CheckBox** and **CheckBoxList** controls and what the differences are between them.
Adding CheckBox Web Server Controls to a Web Forms Page
Gives directions for adding an individual **CheckBox** control to a page.
Adding CheckBoxList Controls to a Web Forms Page
Gives directions for adding a **CheckBoxList** control (rather than individual **CheckBox** controls) to a page.
Getting and Setting a CheckBox Web Server Control Value Programmatically
Gives directions for programmatically selecting and clearing an individual **CheckBox** control.
Responding to User Selection in a CheckBox Web Server Control
Gives directions for writing an event handler that responds directly when the user clicks an individual **CheckBox** control.
Setting Layout in a CheckBoxList Web Server Control
Gives directions for specifying how many columns of check boxes the **CheckBoxList** control displays and how they are ordered.

Related Sections

Adding Web Server Controls to a Web Forms Page
Gives directions for including Web server controls on your Web Forms page.
Setting ASP.NET Server Control Properties
Suggests additional topics of interest concerning server control properties.
Setting Web Server Control Properties Programmatically
Gives directions for setting Web server control properties in code.
Customizing the Appearance of ASP.NET Server Controls Using Styles
Gives directions for changing the look of Web server controls on a page.
Adding Items in a List Web Server Control
Gives directions for defining individual check boxes in the **CheckBoxList** control, either at design time or programmatically at run time.
Populating a List Web Server Control from a Database
Gives directions for binding a **CheckBoxList** control to a database in order to display bound check boxes.
Determining the Selection in a List Web Server Control
Gives directions for checking in code what the user has selected.
Setting the Selection in a List Web Server Control
Gives directions for preselecting items in a **CheckBoxList** control.
Responding to Changes in a List Web Server Control
Gives directions for checking in code which check box the user has clicked in a **CheckBoxList** control.

Introduction to the CheckBox and CheckBoxList Web Server Controls

You can use two types of Web server controls to add check boxes to a Web Forms page: individual CheckBox controls or a CheckBoxList control. Both controls provide a way for users to input Boolean data: true or false, yes or no.

> **Note** You can also use the HtmlInputCheckBox server control to add check boxes to a Web Forms page. For details about the differences between Web and HTML server controls, see Introduction to ASP.NET Server Controls.

CheckBox Control versus CheckBoxList Control

You add individual **CheckBox** controls to a page and work with them singly. Alternatively, the **CheckBoxList** control is a single control that acts as a parent control for a collection of check box list items. It derives from a base ListControl class, and therefore works much like the ListBox, DropDownList, and RadioButtonList Web server controls. For that reason, many of the procedures for working with the **CheckBoxList** control are the same as they are for the other list Web server controls.

Each type of control has advantages. By using individual **CheckBox** controls, you get more control over the layout of the check boxes on the page. For example, you can include text (that is, non-check-box text) between each check box. You can also control the font and color of check boxes individually.

The **CheckBoxList** control is a better choice if you want to create a series of check boxes out of data in a database. (You can still bind an individual **CheckBox** control to data.)

CheckBox Events

Events work slightly differently between individual **CheckBox** controls and the **CheckBoxList** control.

Individual CheckBox Controls

Individual **CheckBox** controls raise an event — CheckedChanged — when users click the control. By default, this event does not cause the page to be posted to the server, but you can have the control force an immediate post by setting the AutoPostBack property to **true**. For details about responding to this event directly, see
Responding to User Selection in a CheckBox Web Server Control.

> **Note** The ability of a **CheckBox** control to post to the server when it is checked requires that the browser support ECMAScript (JScript, JavaScript) and that scripting be enabled on the user's browser.

Whether a **CheckBox** control posts to the server or not, you might not need to create an event handler for the **CheckedChanged** event. You can test which check box is selected when the form has been posted to the server by a control such as a **Button** control. Typically, you create an event handler for the **CheckedChanged** event only if you need to know that the check box was changed.

CheckBoxList Control

In contrast, the **CheckBoxList** control raises a SelectedIndexChanged event when users select any check box in the list. By default, the event does not cause the form to be posted to the server, although you can specify this option by setting the **AutoPostBack** property to true. For details, see Responding to Changes in a List Web Server Control.

As with individual **CheckBox** controls, it is more common to test the state of the **CheckBoxList** control after the form has been posted some other way. For details, see Determining the Selection in a List Web Server Control.

Binding Data to the Control

As with any Web server control, you can bind an individual **CheckBox** control to a data source, and you can bind any property of the **CheckBox** control to any field of the data source. For example, it is typical to set the control's Checked property from information in a database.

You can also bind a **CheckBoxList** control to a data source. In that case, the check boxes each represent a different record in the data source.

For general information about data binding Web server controls, see Introduction to Data Binding in Web Forms Pages. For details about binding the **CheckBoxList** control to data, see Populating a List Web Server Control from a Database.

See Also

Adding Web Server Controls to a Web Forms Page | Introduction to Validating User Input in Web Forms |
Setting ASP.NET Server Control Properties | Customizing the Appearance of ASP.NET Server Controls Using Styles |
ASP.NET Server Control Event Model | CheckBox Class | CheckBoxList Class | Introduction to Web Forms State Management

Adding CheckBox Web Server Controls to a Web Forms Page

You can add a check box to a Web Forms page in two ways:

- By adding an individual **CheckBox** Web server control.
- By adding a **CheckBoxList** control and adding individual list items to the control.

For details about the differences in these approaches, see Introduction to the CheckBox and CheckBoxList Web Server Controls.

> **Note** This topic addresses how to add individual **CheckBox** controls to the page. For details about adding a **CheckBoxList** control, see Adding CheckBoxList Controls to a Web Forms Page.

To add an individual CheckBox control to a Web Forms page

1. In **Design** view, drag a **CheckBox** control from the **Web Forms** tab of the Toolbox onto the page.
2. Specify a caption by setting the Text property.
3. Optionally, change the orientation of the caption by setting the TextAlign property.

See Also

Adding Web Server Controls to a Web Forms Page | Adding CheckBoxList Controls to a Web Forms Page | Adding Controls to a Web Forms Page Programmatically | Customizing the Appearance of ASP.NET Server Controls Using Styles | Setting ASP.NET Server Control Properties

Adding CheckBoxList Controls to a Web Forms Page

When you use a **CheckBoxList** control, you add a single control to the page. Adding items to the control is a separate process that depends on whether you will display a static list in the control or a list generated dynamically at run time. For details, see Adding Items in a List Web Server Control.

> **Note** The procedure for working with individual **CheckBox** Web server controls is different. For details, see Adding CheckBox Web Server Controls to a Web Forms Page.

To add a CheckBoxList control to a Web Forms page

1. In **Design** view, drag the **CheckBoxList** control from the Toolbox onto the page.
2. Optionally, change the orientation of the caption by setting the **TextAlign** property.
3. Optionally, change the layout of the control to display multiple columns. For details, see Setting Layout in a CheckBoxList Web Server Control.

See Also

Introduction to the CheckBox and CheckBoxList Web Server Controls | Adding CheckBox Web Server Controls to a Web Forms Page | Adding Items in a List Web Server Control | Setting Layout in a CheckBoxList Web Server Control | Determining the Selection in a List Web Server Control | Setting the Selection in a List Web Server Control

Getting and Setting a CheckBox Web Server Control Value Programmatically

You can programmatically read or set the value of a **CheckBox** Web server control.

> **Note** If you are working with a **CheckBoxList** control, the procedure for getting and setting the value of a check box is different. For details, see Determining the Selection in a List Web Server Control and Setting the Selection in a List Web Server Control.

To get or set CheckBox Web server control selection

- Get or set the **CheckBox** control's **Checked** property. A value of **true** means the check box is selected.

  > **Note** Testing the value of a check box does not tell you if the user changed the value of the control, only if it is selected. For details about how to check for a change in the control, see Responding to User Selection in a CheckBox Web Server Control.

  In the following example, if the check box `chkCanWeSendEmail` is not selected, the check box `chkSendInHTMLFormat` is automatically cleared as well.

```
' Visual Basic
If chkCanWeSendEmail.Checked = false Then
   chkSendInHTMLFormat.Checked = false
End If

// C#
if(chkCanWeSendEmail.Checked == false){
   chkSendInHTMLFormat.Checked = false;
}
```

See Also

CheckBox Class | CheckBoxList Class | Setting ASP.NET Server Control Properties

Responding to User Selection in a CheckBox Web Server Control

When the user selects a **CheckBox** control, the control raises an event that you can respond to.

> **Note** The **CheckBoxList** control raises events differently than individual **CheckBox** controls. For details, see Responding to Changes in a List Web Server Control.

You might not need to respond directly to the check event of a **CheckBox** control at all. You respond to the event only if it's important to know when the user has changed the check box selection.

If you are only interested in the state of the check box when selected, and not whether it has changed, you can simply test the check box after the form has been posted to the server by a control such as a **Button** control. For details about determining whether a **CheckBox** control is selected, see Getting and Setting a CheckBox Web Server Control Value Programmatically.

To respond to a selection in a CheckBox control

- Create an event handler for the control's CheckedChanged event.

  By default, the CheckedChanged event does not immediately cause the Web Forms page to be posted to the server. Instead, the event is raised in server code the next time the form is posted, as when a **Button** Web server control is clicked. To have the **CheckedChanged** event cause an immediate posting, set the **CheckBox** control's AutoPostBack property to **true**.

  > **Note** The ability of a **CheckBox** control to post to the server when it is checked requires that the browser support ECMAScript (JScript, JavaScript) and that scripting be enabled on the user's browser.

  The following example displays "True" or "False" in a **Label** Web server control to reflect the latest change in a **CheckBox** control.

```
' Visual Basic
Private Sub CheckBox1_CheckedChanged(ByVal sender As _
   System.Object, ByVal e As System.EventArgs) Handles _
   CheckBox1.CheckedChanged
   Label1.Text = "CheckBox1 selection is: " & CheckBox1.Checked
End Sub

// C#
private void CheckBox1_CheckedChanged(object sender, EventArgs e)
{
   Label1.Text = "CheckBox1 selection is: " + CheckBox1.Checked;
}
```

See Also

ASP.NET Server Control Event Model | CheckBox Class | CheckBoxList Class | Specifying Whether Controls Post to the Server

Setting Layout in a CheckBoxList Web Server Control

By default, the **CheckBoxList** control displays a single column of check boxes. However, you can specify any number of columns, and within the columns, you can specify how the items are ordered: vertically (the default) or horizontally. Vertical layout in three columns results in a layout such as the following:

```
A     D     G
B     E     H
C     F
```

Horizontal layout for the same items results in the following layout:

```
A     B     C
D     E     F
G     H
```

To specify column count and ordering

1. Set the **CheckBoxList** control's RepeatColumns property to the number of columns you want.
2. Set the **RepeatDirection** property to **Vertical** or **Horizontal**. If you are setting the property in code, use the RepeatDirection enumeration, as shown in the following example:

```
' Visual Basic
CheckBoxList1.RepeatDirection = RepeatDirection.Horizontal

// C#
CheckBoxList1.RepeatDirection = RepeatDirection.Horizontal;
```

See Also

Adding CheckBoxList Controls to a Web Forms Page | Adding Items in a List Web Server Control | Determining the Selection in a List Web Server Control | CheckBox Class | CheckBoxList Class | Setting the Selection in a List Web Server Control

DataGrid Web Server Control

The **DataGrid** Web server control is a multi-column, data-bound grid. Extensive customization is available at design-time using the Property Builder dialog box. At design-time you can build columns that display and edit data, columns that contain Edit, Update, Cancel, and Select buttons, columns that contain custom buttons, and template columns. You can then lay out the template columns in template-editing mode.

The following topics provide information that you will find useful in learning to work with the **DataGrid** control.

In This Section

Introduction to the DataGrid Web Server Control
Background information on the **DataGrid** Web server control, including what you can do with it and an overview of its features.
DataGrid Columns
Describes the different types of columns you can add to a **DataGrid** control.
Adding DataGrid Web Server Controls to a Web Forms Page
Gives directions for adding a **DataGrid** Web server control to a Web Forms page and setting the required properties to make it work.
Adding Template Columns to a DataGrid Web Server Control
Gives directions for creating columns with a custom-defined combination of HTML text and controls.
Adding Bound Columns to a DataGrid Web Server Control
Gives directions for selecting which data fields are displayed as columns.
Adding Button Columns to a DataGrid Web Server Control
Gives directions for adding buttons to individual items that invoke custom functionality for list items.
Adding Hyperlink Columns to a DataGrid Web Server Control
Gives directions for creating a column that displays data as hyperlinks to other pages.
Adding Sorting to a DataGrid Web Server Control
Gives directions for allowing users to sort the data based on a particular column.
Allowing Users to Delete Items in a DataGrid Web Server Control
Gives directions for allowing users to delete either individual items or a batch of items.
Allowing Users to Edit Items in a DataGrid Web Server Control
Gives directions for allowing users to put a **DataGrid** control row into edit mode.
Allowing Users to Select Items in a DataGrid Web Server Control
Gives directions for providing a Select button that users can click to highlight an item.
Specifying Grid Item Format in a DataGrid Web Server Control
Gives directions for dynamically controlling the contents and appearance of items.
Paging Behavior in DataGrid Web Server Controls
Describes the methods available for implementing paging of the grid.
Specifying Paging Behavior in a DataGrid Web Server Control
Gives directions for adding paging to the grid.
Creating Templates Programmatically in the DataGrid Control
Gives directions for creating a template class that you can use to add template columns to the DataGrid control at run time.

Related Sections

Programming Web Forms
Suggests additional topics on working with Web Forms in code.
Introduction to ASP.NET Server Controls
Provides background information on working with controls that you can use on Web Forms.

Introduction to the DataGrid Web Server Control

The DataGrid Web server control displays data in a tabular layout. By default, the **DataGrid** displays data in read-only mode, but the **DataGrid** is also capable of automatically displaying the data in editable controls at run-time. The **DataGrid** control can also create the Select, Edit, Update, and Cancel buttons and programming structure for you through the Property Builder dialog box. Additionally, the **DataGrid** supports paging, though you may also use the custom navigation features of the control to improve performance by controlling the amount of data sent to the client browser.

> **Note** To have complete control over the HTML rendered by the list, use the **Repeater** Web server control. Use the **DataList** control to display data in a non-tabular layout. For more information, see Controls for Displaying Lists On Web Pages.

The following sections introduce the features of the **DataGrid** control.

Binding Data to the Control

The **DataGrid** Web server control must be bound to a data source via its **DataSource** property or it will not be rendered on the page. Typical data sources for the **DataGrid** are the **DataSet** and data readers. You can use data sources available from the Toolbox, such as the **DataSet** or **DataView** class. You can also bind to data sources in code, such as a data reader or an array. The control can use as a data source any class that supports the IEnumerable interface.

When data binding, you specify a data source for the **DataGrid** control as a whole. The grid displays one row — one item — for every row in the data source. By default, the **DataGrid** control generates a bound column for each field in the data source. However, you can pick which fields in the data source generate columns in the grid. For more information on data binding, see Introduction to Data Access in Web Forms Pages.

When the Web page runs, your code must call the control's **DataBind** method to load the grid with data. You call the method again if the data changes (for example, in an event handler) to refresh the grid.

As part of the data binding, you can specify a DataKeyField property. This property allows you to specify information that uniquely identifies each item in the grid. The information does not have to be part of the information displayed in the grid. It can consist of the name of a field in the data source (such as a primary key). You can use this information to update a specific item in the data source. For example, if you set the **DataKeyField** to be **customerId**, you could later update the record where `customerID=DataGrid1.DataKeys[DataGrid1.SelectedIndex]`.

As with other server controls, data binding in the **DataGrid** control is one-way; that is, data binding is read-only. If you are using the grid to allow users to edit the data, you must create your own code to update the data source. After updating the data source, you again bind the data to the data source. The update logic often takes advantage of the **DataKeyField** to locate the correct record using the key value.

For more details about updating data using the **DataGrid** control, see "Editing Items" below. For general information about data binding in server controls, see Data Access in Web Forms Pages.

Format Items

The **Format** tab of the Property Builder dialog box controls the layout of the rows of the **DataGrid**. These settings specify the color, font, and alignment of the rows. The actual control, text, and data contents of the row are specified in the **Columns** tab of the Property Builder. The following items can be configured:

- **DataGrid** These settings are applied to the entire grid. Other item formats override these settings.
- **Header** The Header is the top row of the **DataGrid**, and does not display the row data of the data source. You must indicate in the General tab of the Property Builderwhether you want the Header row to be displayed. It is displayed by default.
- **Footer** The **Footer** is the bottom row of the **DataGrid** and does not display the row data of the data source. You must indicate in the General tab of the Property Builderwhether you want the Footer row to be displayed. It is not displayed by default.
- **Pager** If you are displaying only a few rows of your data source at a time, using the paging abilities of the **DataGrid**, use this item to control the appearance of the paging controls.
- **Items** These four items specify how the data rows will appear. They are:
  - **Normal Items** This format is applied to all the data rows of the table, unless overridden by another item.
  - **Alternating Items** This format is applied to alternating items in the control.
  - **Selected Items** This format is applied to the selected item in the control.
  - **Edit Mode Items** This format is applied to the item being edited.

- **Columns** This item allows you to specify the format of the columns you create in the Columns tab of the Property Builder.

For more information on using item formats, see Specifying Grid Item Format in a DataGrid Web Server Control and Specifying Paging Behavior in a DataGrid Web Server Control.

Columns

The **DataGrid** control allows you to specify the columns it displays in a variety of ways. By default, the columns are generated automatically based on fields in the data source. However, in order to control the content and layout of columns more precisely, you can define the following types of columns:

Type of Column	Description
Bound column	Allows you to specify which data source field to display. Allows you to specify the format of that field, using a .NET formatting expression. For details see Adding Bound Columns to a DataGrid Web Server Control.
Hyperlink column	Displays information as hyperlinks. A typical use is to display data (such as a customer number or product name) as a hyperlink that users can click to navigate to a separate page that provides details about that item. For details see Adding Hyperlink Columns to a DataGrid Web Server Control.
Button column	Allows you to add a button for each item in the grid and define custom functionality for that button. For example, you might create a button labeled "Add to Shopping Cart" that runs your custom logic when a user clicks it. You can also add predefined buttons for Select, Edit, Update, Cancel, and Delete functions. For details see Adding Button Columns to a DataGrid Web Server Control.
Edit, Update, Cancel column	Allows you to create in-place editing. For more details, see "Editing Items" below.
Template column	Allows you to create combinations of HTML text and server controls to design a custom layout for a column. The controls within a template column can be data-bound. Template columns give you great flexibility in defining the layout and functionality of the grid contents, because you have complete control over how the data is displayed and what happens when users interact with rows in the grid. For details see Adding Template Columns to a DataGrid Web Server Control.

For more information about columns, see DataGrid Columns.

Events

The **DataGrid** control supports several events. One of them, the **ItemCreated** event, gives you a way to customize the item-creation process. The **ItemDataBound** event also gives you the ability to customize the **DataGrid** items, but after the data is available for inspection. For example, if you were using the **DataGrid** control to display a to-do list, you could display overdue items in red text, completed items in black text, and other tasks in green text.

The remaining events are raised in response to button or **LinkButton** clicks in grid items. They are designed to help you implement common data manipulation tasks. Four events of this type are supported:

- **EditCommand**
- **DeleteCommand**
- **UpdateCommand**
- **CancelCommand**

The Property Builder dialog box of the **DataGrid** control will build the buttons needed to raise these events at run time. When the user clicks one of the buttons (labeled by default Edit, Delete, Update, or Cancel, respectively), the corresponding event is raised.

The **DataGrid** control also supports the **ItemCommand** event that is raised when a user clicks a button that is not one of the predefined buttons above. You can use this event for custom functionality by setting a button's **CommandName** property to a value you need and then testing for it in the **ItemCommand** event handler. (For example, you could use this approach when selecting an item, as documented in Allowing Users to Select Items in a DataList Web Server Control.)

For a complete list of the events supported by the **DataGrid** control, see DataList Events. For more information about events, see ASP.NET Server Control Event Model. For information on supporting these events with the DataGrid, see Allowing Users to Select Items in a DataGrid Web Server Control, Allowing Users to Edit Items in a DataGrid Web Server Control, and Allowing Users to Delete Items in a DataGrid Web Server Control.

Editing Items

You can allow users to edit the contents of individual rows in a grid item. There are two ways to do this:

- Allow in-place editing. This is the easier way to implement editing. Using the Property Builder, you include a special column in the grid known as an Edit, Update, Cancel column. At run time, each row of the column contains a button labeled "Edit". When a user clicks this edit button, the control automatically redisplays the current row with **TextBox** controls for all columns. The column with the edit button is redisplayed with Update and Cancel buttons. You add code to the **UpdateCommand** and **CancelCommand** events to propagate the changes in the editing controls back to your data source.

  For details, see Allowing Users to Edit Items in a DataGrid Web Server Control.

- Create a custom-editing layout with a template column. You can create columns that display the data in editable controls. This approach allows you to choose which columns can be edited and how the user can edit the data. For example, instead of using a **TextBox** control for true-false data, you might prefer a **ListBox** control with "True" and "False" entries. A template column allows you to put more than one control in a DataGrid cell.

  The template column supports four templates:

  - HeaderTemplate
  - ItemTemplate
  - EditItemTemplate
  - FooterTemplate

  For more information, see Allowing Users to Edit Items in a DataGrid Web Server Control and Allowing Users to Delete Items in a DataGrid Web Server Control.

Sorting

The **DataGrid** Web server control provides a way for you to add sorting to the grid. The grid does not directly support sorting — that is, it doesn't sort the contents of the grid for you. Instead, it provides a way for you to add sort options to the grid, such as hyperlinks in the column heads that users click to sort. The grid then notifies you when a user has requested sorting. You then sort the data source and rebind the data source to the **DataGrid**.

The **DataGrid** control provides two levels of sorting support:

- **Default sorting** You specify that you want the grid to support sorting. The column headings are all link buttons; when the user clicks one, you are notified of the sort. This solution is easy, but does not allow you to pick which columns support sorting — they all do.
- **Custom sorting** You define which columns support sorting. This is a more complex option, because you must define the columns using either bound columns or template columns. However, not only does this allow you to specify which columns can be sorted, but you can use whatever control you want in the column header as the sort button. For example, you might use a graphic that you create using an **ImageButton** Web server control.

For details, see Adding Sorting to a DataGrid Web Server Control.

See Also

DataGrid Web Server Control | Adding DataGrid Web Server Controls to a Web Forms Page | DataGrid Class

DataGrid Columns

When you are working with a **DataGrid** Web server control, you can build several different types of columns into the grid:

- Bound columns
- Button columns
- Hyperlink columns
- Template columns

In addition, it is also possible to allow the grid to auto-generate columns based on information in the grid's data source.

Auto-Generated Columns

If the **DataGrid** control's AutoGenerateColumn property is set to **true**, the control will generate and display a bound column for each bindable column in the data source. Bindable column types include primitive types, string, datetime, and decimal. If you have a field in your data source that is not a supported column type, then the control does not create a column for that field. As long as your data source has at least one bindable column, the **DataGrid** control binds to the source and generates the bindable columns, and ignores those that are not. If **AutoGenerateColumns** is **true** and you create explicit columns, the grid will show both your explicit columns and the auto-generated ones.

Using auto-generated columns is a quick and easy way to display information in the grid. Because these columns are automatically generated at run time and displayed in the order in which they occur in the data source, you do not have flexibility in how these columns are laid out. Therefore, using auto-generated columns is recommended primarily for simple data display, prototype pages, and other situations where you do not necessarily need to lay the columns out carefully. You should explicitly define columns when you want to control which columns to display and in what order.

Bound Columns

A bound column (BoundColumn element) displays information from one column of a data source. You typically add one bound column to the grid for every data column you want to display. Use bound columns when you want to choose which data columns to display in the grid. Bound columns also allow you to set a variety of properties to specify features such as:

- The text, font, and color of the column header and footer.
- Column width.
- Data format.
- Whether the column is read-only (that is, whether it will display an editable control when the row is in edit mode).

For details on creating bound columns in the Visual Studio Web Forms Designer, see Adding Bound Columns to a DataGrid Web Server Control. For details on creating bound columns, see DataGrid Web Server Control.

Button Columns

Button columns allow you to add grid columns with buttons that allow users to perform row-specific functions such as selecting the current row, deleting it, putting it into edit mode. You can create buttons for predefined functions or create your own functionality (for example, an "Add to Shopping Cart" button).

You can define two types of button columns:

- An EditCommandColumn element is predefined to contain three buttons. In display mode, the column displays an Edit button that users can click to put the row into edit mode (with editable controls). When the row is in edit mode, the Edit button is replaced with two buttons: an Update button and a Cancel button. You can specify the text (caption) of the buttons.
- A ButtonColumn element can contain any arbitrary button. You can use a **ButtonColumn** element to define certain predefined functionality (such as selecting a row or deleting it) or to create buttons for your own functionality.

When you define a button for a button column, you specify the button's text (caption) and its button type (LinkButton or Button). In addition, for buttons in the **ButtonColumn** element, you specify a special CommandName property. The setting for this property determines what happens when the button is clicked. For example:

Setting	Result
Select	Clicking the button automatically selects the row.
Delete	Clicking the button raises the **DataGrid** control's DeleteCommand.

A custom (non-predefined) value	Clicking the button raises the **DataGrid** control's generic ItemCommand.

Note Settings for the **CommandName** property are case sensitive.

In most cases, the button click raises an event that you then handle. (If the **CommandName** property is set to "Select," the event is raised, but it is also handled automatically by the grid to select the specified rows.) Typically, in your handler you determine which row was affected and the appropriate action — for example, setting that row into edit mode (or canceling it), deleting the data row represented by that grid row, and so on. For custom buttons, the grid raises the **ItemCommand** event and passes the value of the **CommandName** property as part of the **EventArgs** object.

For details on creating button columns in the Visual Studio Web Forms Designer, see Adding Button Columns to a DataGrid Web Server Control. For details on creating button columns, see DataGrid Web Server Control.

For implementing the functionality of button columns, see the following topics:

- Allowing Users to Edit Items in a DataGrid Web Server Control
- Allowing Users to Select Items in a DataGrid Web Server Control
- Allowing Users to Delete Items in a DataGrid Web Server Control

Hyperlink Columns

A hyperlink column (HyperLinkColumn element) displays a link in each row. You can specify static text for the hyperlink or derive the link's text from a data column. Similarly, you can specify a static URL as the link's target or get the URL from a data source.

For details on creating hyperlink columns in the Visual Studio Web Forms Designer, see Adding Hyperlink Columns to a DataGrid Web Server Control. For details on creating button columns as HTML elements, see DataGrid Web Server Control.

Template Columns

A template column allows you to create a custom layout for a column based on any mixture of controls and static text. Template columns are similar to templates used for the **DataList** and **Repeater** controls, with the difference that you are creating the layout for just a column rather than for a row-like item. For details about templates, see Web Server Controls Templates.

Because they give you complete control over the contents of the column, template columns provide the greatest flexibility for column layout. Some examples include:

- You can easily mix static text with controls.
- You can specify what controls are used in edit mode. When a bound column is in edit mode, it displays a text box no matter what type of data is being edited. By using a template control, you can specify a check box, a drop-down list, or other appropriate control instead.

Within the template column, you can create up to four separate template types:

- The HeaderTemplate and FooterTemplate define what will appear in the column header and footer, respectively.
- The ItemTemplate defines how information appears in the column when the row is in display mode. You often add controls such as a Label or Literal control to display data.
- The EditItemTemplate defines the controls (and text, if applicable) that appear in the column when it is in edit mode. Typically, this involves text boxes, check boxes, and drop-down lists.

For details on creating template columns in the Visual Studio Web Forms Designer, see Adding Template Columns to a DataGrid Web Server Control. For details on creating template columns as HTML elements, see DataGrid Web Server Control.

See Also

Introduction to the DataGrid Web Server Control | DataGrid Web Server Control

Adding DataGrid Web Server Controls to a Web Forms Page

You can add and configure a **DataGrid** Web server control in a few steps in order to display data on the page. In many cases, you will refine the grid to format the columns or add editing or sorting.

The following procedure describes the basic steps to create a working **DataGrid** control.

To add a DataGrid Web server control to a Web Forms page

1. Add a data source to the Web Forms page. There are several methods for handling data in a Web application. To choose an appropriate method, see Introduction to Data Access with ADO.NET and Introduction to Data Access in Web Forms Pages.
2. In Design view, drag a **DataGrid** control from the Web Forms tab of the Toolbox onto the page.
3. Set the control's **DataSource** property. In the Properties window, the drop-down list for the **DataSource** property will display all the data sources, such as **DataSet** and **DataView** controls, defined on the page.

   By default, the **DataGrid** control will generate a bound column for each field in the data source. Each field in the data source is rendered in a separate column, in the order it occurs in the data source. Field names appear in the grid's column headers, and values are rendered in text labels.

4. In code, call the control's **DataBind** method. This is typically done the first time the page runs (when `Page.IsPostBack` is false) in the page's **Page_Load** event using code as shown below. The **DataBind** method needs to be called whenever the data changes. On subsequent round trips of the page (when `Page.IsPostBack` is **true**), the **DataBind** method is typically called in the event handler, such as the paging event, **PageIndexChanged**, when the data you wish to display in the grid changes.

```
' Visual Basic
Private Sub Page_Load(ByVal sender As System.Object, _
ByVal e As System.EventArgs) Handles MyBase.Load
   'Put user code to initialize the page here
   If Not Page.IsPostBack Then
   ' Put user code here to initialize the data source
   ' (such as filling a dataset)
      DataGrid1.DataBind()
   End If
End Sub

// C#
private void Page_Load(object sender, System.EventArgs e)
{
   if (!Page.IsPostBack)
   {
      // Put user code here to initialize the data source
      // (such as filling a dataset)
      DataGrid1.DataBind();
   }
}
```

See Also

DataGrid Web Server Control | Introduction to the DataGrid Web Server Control

Adding Template Columns to a DataGrid Web Server Control

By using a template column, you can specify the controls that appear in your column. The template column supports four items — header, footer, item, and edit item. The buttons that you add to a template column bubble their click events to the **DataGrid's ItemCommand** event.

To add a template column to the DataGrid

1. Add a **DataGrid** to the page. For more information, see Adding DataGrid Web Server Controls to a Web Forms Page.
2. In Design view, select the **DataGrid** control, then click the **Property Builder** link at the bottom of the Properties window.
3. In the ***DataGrid* Properties** dialog box, on the **General** tab, select the **DataSource** and **DataMember**.

   There are several methods for handling data in a Web application. To choose an appropriate method, see Introduction to Data Access with ADO.NET and Introduction to Data Access in Web Forms Pages. If you have created a data source using a **DataSet** or **DataView** control in the designer, you will be able to pick the data control from the drop-down list for the **DataSource** property.

4. Switch to the **Columns** tab.
5. In the **Available columns** box, select **Template Column** and click the **Add** button ( ). Click **OK** to close the ***DataGrid* Properties** dialog box.
6. In the designer, right-click the **DataGrid** control, point to **Edit Template**, and choose the template column you created.
7. You can edit four templates — Header, Footer, Item, and EditItem. Add controls to the template as needed.
8. To exit edit mode, right-click the **DataGrid** control and select **End Template Editing** from the menu.
9. Add code to respond to any button-click events from buttons in the template column. For information on responding to button events in the templates, see Responding to Button Events in DataList, Repeater, or DataGrid Items.

See Also

DataGrid Columns | Creating Web Server Control Templates | DataGrid Web Server Control | Introduction to the DataGrid Web Server Control

Adding Bound Columns to a DataGrid Web Server Control

Using bound columns in a **DataGrid** Web server control allows you to specify which columns from the data source to display, customize the header and footer for each column, and control sorting.

To add bound columns to a DataGrid control

1. Add a **DataGrid** to the page. For more information, see Adding DataGrid Web Server Controls to a Web Forms Page.
2. In Design view, select the **DataGrid** control, then click the **Property Builder** link at the bottom of the Properties window.
3. In the ***DataGrid* Properties** dialog box, click the **Columns** tab.
4. Clear the **Create columns automatically at run time** check box.
5. In the **Available columns** box, expand **Data Fields** and select a field to display, then click the **Add** button ( ).
6. Optionally, set the header text and footer text for each column you add. If you want to use an image rather than text for the header, select an image in the **Header image** box.

   Note The header and footer will be displayed only if you have selected **Show header** and **Show footer** in the **General** tab.

7. If you want to be able to sort by the contents of the column, set the **Sort expression** for the column. For details, see Adding Sorting to a DataGrid Web Server Control.
8. If you want the data in the column to be formatted — for example, if you want to specify how a date is displayed — set the **Data formatting expression**. For information about creating a formatting expression, see Composite Formatting.

See Also

DataGrid Columns | Adding Button Columns to a DataGrid Web Server Control |
Adding Hyperlink Columns to a DataGrid Web Server Control | Adding Template Columns to a DataGrid Web Server Control |
DataGrid Web Server Control

Adding Button Columns to a DataGrid Web Server Control

A **ButtonColumn** column in a **DataGrid** Web server control allows you to define custom functionality for items in the grid beyond the edit, delete, select, and hyperlink features of other column types. For example, you can use a **ButtonColumn** to create an "Add to shopping cart" button.

You can specify that the buttons in the **ButtonColumn** be either **PushButton** (pushbutton-style) or **LinkButton** (hyperlink-style) buttons. The button captions can either be static text or text read from a database.

When you define a button column, you specify a command associated with the button. When users click the button, the button's command is passed to the container — the **DataGrid** control — where you can handle it with your custom code.

To add a button column

1. In Design view, select the **DataGrid** control, then click the **Property Builder** link at the bottom of the Properties window.
2. In the ***DataGrid* Properties** dialog box, click the **Columns** tab.
3. In the **Available columns** box, select **Button Column** and click the **Add** button (>).

   Note The **Button Column** node can be expanded to reveal the **Select**, **Edit**, and **Delete** button column types. For a custom button, however, you must select the **Button Column** node.

4. Set the column's header text and footer text. If you want to use an image rather than text for the header, select the image in the **Header image** box. By default, the **DataGrid** displays the name of the field in the data source as the header text.

   Note The header and footer will be displayed only if you have selected **Show header** and **Show footer** in the **General** tab.

5. If you want to be able to sort by the contents of the column, set the **Sort expression** for the column. For details, see Adding Sorting to a DataGrid Web Server Control.
6. If you are creating a button with static text, set the button's caption in the **Text** box. If you are displaying the contents of a database field as the caption of the button, select the field from the **Text field** list. If you are setting the button text using a text field, you can specify a formatting expression for the text using the **Text format string** box. For information about creating a formatting expression, see Composite Formatting.
7. Select whether the button is a link button (hyperlink-style) or push button from the **Button type** list.
8. In the **Command name** box, enter the command that should be sent to the **DataGrid** control when the button is clicked. For details about working with the command text, see the section below, "To respond to button events."

Defining the button column is one part of creating custom buttons in the grid. You must also create an event handler to respond when a user clicks a button.

To respond to button events

1. Create an event handler for the **DataGrid** control's ItemCommand event. This event is raised when the user clicks any button in any button column in the grid. For details, see Creating Event Handlers in Web Forms Pages.
2. In the method, you determine which button was clicked by examining the information passed in the **DataGridCommandEventArgs** argument of the event.
   a. Check the **CommandName** property of the event-argument object (`e.CommandName`) to see what command string was passed.
   b. Use the **Item** property of the argument object to get a reference to the DataGridItem Class instance associated with the row where the user clicked. The **Item** property in turn contains a **Cells** collection (`e.Item.Cells`) containing the contents of the current row. Each cell has a **Text** property containing the contents of the cell.

The following example shows how you can test to see whether the command string `"AddToCart"` was passed to the event handler. If so, the example extracts the value of a column (2) containing a product ID, which is used to add items to the shopping cart.

```
' Visual Basic
Private Sub DataGrid1_ItemCommand(ByVal source As Object, _
ByVal e As System.Web.UI.WebControls.DataGridCommandEventArgs) _
Handles DataGrid1.ItemCommand
   'e.Item is the row of the table where the button was
   ' clicked.
```

```
    Dim productID As String
    productID = e.Item.Cells(2).Text
    If (e.CommandName = "AddToCart") Then
        ' Add code here to add the productID item to the cart.
    End If
    DataGrid1.DataBind()
End Sub

// C#
private void DataGrid1_ItemCommand(object source,
System.Web.UI.WebControls.DataGridCommandEventArgs e)
{
    // e.Item is the row of the table where the button was
    // clicked.
    string productID = e.Item.Cells[2].Text;

    if (e.CommandName == "AddToCart")
    {
        // Add code here to add the productID item to the cart.
    }
    DataGrid1.DataBind();
}
```

See Also

DataGrid Columns | Adding DataGrid Web Server Controls to a Web Forms Page | DataGrid Web Server Control | Introduction to the DataGrid Web Server Control

Adding Hyperlink Columns to a DataGrid Web Server Control

A Hyperlink column allows you to display text as a hyperlink that users can click to navigate to another page. The text of the hyperlink can either be static — the same text for every item — or the contents of a field in the data source. Similarly, the target page for the hyperlink can be a single page for all the hyperlinks, or a URL derived from the data source.

> **Security Note** The URLs associated with a hyperlink can be tampered with by a malicious user. For more information, see Scripting Exploits.

To add a hyperlink column

1. Add a **DataGrid** to the page. For more information, see Adding DataGrid Web Server Controls to a Web Forms Page.
2. In Design view, select the **DataGrid** control, then click the **Property Builder** link at the bottom of the Properties window.
3. In the ***DataGrid* Properties** dialog box, click the **Columns** tab.
4. In the **Available columns** box, select **Hyperlink Column** and click the **Add** button (>).
5. Optionally, do the following:
   - Set the column's header text and footer text. If you want to use an image rather than text for the header, select an image in the **Header Image** box.

     > **Note** The header and footer will be displayed only if you have selected **Show header** and **Show footer** in the **General** tab.

   - If you want to be able to sort by the contents of the column, set the **Sort expression** for the column. For details, see Adding Sorting to a DataGrid Web Server Control.
6. Specify the text to display as a hyperlink using one of the following methods:
   - To specify static text (same text for every row in the grid), enter it into the **Text** box.
   - To create the hyperlink from the data source, select the name of the field from the **Text field** list. In this case, you can specify a formatting expression for the hyperlink text using the **Text format string** box. For more information about format expressions, see Composite Formatting.
7. Specify the target page for the hyperlink using one of the following methods:
   - To specify a static page (same target page for every item), enter it into the **URL** box.
   - Optionally, specify a target frame or window in the **Target** box.
   - To use a data field as the source for the target page URL, select the field name from the **URL field** list. In this case, you can specify a formatting expression for the hyperlink text using the **URL format string** box. For more information about format expressions, see Composite Formatting.

To pass information to the target page

You can use **URL field** to pass data to a target page.

- Enter a value for the **URL field** box specifying the data field you wish to pass and define a format for the URL in the **URL format string** that includes the target page. For example, to pass the value of the **LastName** field in the data source to a page about benefits, the **URL field** would be set to `LastName`, and the **URL format string** would be set to `Benefits.aspx?LastName={0}`.

  If you provide information in the **URL Field** box in this way, it is passed to the target page as a query string. In the target page you can get the value passed from the hyperlink by examining the contents of the **QueryString** parameter. In a Web Forms page, you can do this in the **Page_Load** event. The following example shows how you can retrieve the value passed to the target page.

```
' Visual Basic
Private Sub Page_Load(ByVal sender As System.Object, _
ByVal e As System.EventArgs) Handles MyBase.Load
   ' Add code here to initialize the page.
   Dim lastName As String = Request.QueryString("LastName")
   ' Add code here to process LastName.
```

```
End Sub

// C#
private void Page_Load(object sender, System.EventArgs e)
{
   // Add code here to initialize the page.
   string lastName = Request.QueryString["LastName"];
   // Add code here to process LastName.
}
```

See Also

DataGrid Columns | Adding DataGrid Web Server Controls to a Web Forms Page | DataGrid Web Server Control | Introduction to the DataGrid Web Server Control

Adding Sorting to a DataGrid Web Server Control

The **DataGrid** Web server control provides a way for you to add sorting to the grid using these methods:

- **Default sorting** All columns in the grid can be sorted. The header for each column contains a **LinkButton** control (hyperlink) that users click to sort by that column.
- **Custom sorting** You define which columns can be sorted and what type of button in the column heading users click to sort.

In either case, the grid does not sort the rows. Instead, it notifies you of the sort request by raising an event. You then perform the sort in your own code, usually by sorting and then rebinding to the data source.

Specifying Default Sorting

To use default sorting, you set properties of the **DataGrid** control to enable sorting for all columns. You then create an event handler to perform the sort.

To specify default sorting

1. In Design view, select the **DataGrid** control, then click the **Property Builder** link at the bottom of the Properties window.
2. In the ***DataGrid* Properties** dialog box, click the **General** tab.
3. Set the grid's **AllowSorting** property to **true**.

At run-time, the headers for all columns are automatically rendered as **LinkButton** controls. When the user clicks a column head, the name of the bound field for that column is passed as the sort key.

Specifying Custom Sorting

You can define custom sorting by including bound columns and template columns in the grid and writing code in the **SortCommand** event to sort the data.

To specify custom sorting

1. In Design view, select the **DataGrid** control, then click the **Property Builder** link at the bottom of the Properties window.
2. In the ***DataGrid* Properties** dialog box, click the **General** tab.
3. In the **Behavior** section, select the **Allow sorting** box.
4. Switch to the **Columns** tab.
5. Add any combination of bound columns and template columns to the **DataGrid** control. For details, see Adding Bound Columns to a DataGrid Web Server Control and Adding Template Columns to a DataGrid Web Server Control.
6. To enable sorting for a column, enter a value for the **Sort expression** for that column. This can be any string value, but is typically the name of the data field to which the column is bound.

   Note Columns without a sort expression will not raise the **SortCommand** event.

7. Add an event handler for the **SortCommand** event and continue as described in the section below, "Performing the Sort." For more information on event handlers, see Creating Event Handlers in Web Forms Pages.

If you include template columns, you can specify what type of button a user clicks in the column heading to raise the **SortCommand** event.

To specify a button type for sorting in a template column

1. Select the template column to work with and enter a value for its **Sort expression**. This can be any string value, but is typically the name of the data field to which the column is bound.
2. Edit the header template for that column. For details about editing templates, see Creating Web Server Control Templates.
3. Add a **Button**, **LinkButton**, or **ImageButton** control to the template and set its properties to specify a caption or graphic.
4. Set the **Command** property of the button to **sort** (case-sensitive).
5. Add an event handler for the **SortCommand** event and continue as described in the next section, "Performing the Sort." For more information on event handlers, see Creating Event Handlers in Web Forms Pages.

Performing the Sort

When the user clicks the button or hyperlink in the header that has sorting enabled, the **SortCommand** event for the **DataGrid** is raised. In the event handler, you sort the data and rebind it to the **DataGrid**.

To respond to a sort request

- Create an event handler for the grid's SortCommand event. In the method:

  1. Get the sort key (the value you set in the **Sort expression**) from the **SortExpression** property of the **DataGridSortCommandEventArgs** argument.
  2. Sort the data.
  3. Rebind the data from the data source.

     The following example shows how to sort by handling the **SortCommand** event. The method gets the sort key value and uses it to set the **Sort** property of a **DataView** object. The code then rebinds the grid to the **DataView** object, which automatically returns the data in sorted order.

```
' Visual Basic
Private Sub DataGrid1_SortCommand(ByVal source As Object, _
ByVal e As System.Web.UI.WebControls.DataGridSortCommandEventArgs) _
Handles DataGrid1.SortCommand
   DataView1.Sort = e.SortExpression
   DataGrid1.DataBind()
End Sub

// C#
private void DataGrid1_SortCommand(object source,
System.Web.UI.WebControls.DataGridSortCommandEventArgs e)
{
   dataView1.Sort = e.SortExpression;
   DataGrid1.DataBind();
}
```

See Also

Adding DataGrid Web Server Controls to a Web Forms Page | DataGrid Web Server Control | Introduction to the DataGrid Web Server Control | Paging Behavior in DataGrid Web Server Controls

Allowing Users to Delete Items in a DataGrid Web Server Control

You can set up your **DataGrid** so that a user can delete a particular row by clicking a button. There are three ways to create a Delete button. You can:

- Have the Property Builder create the Delete button
- Create the button yourself in the Property Builder
- Add the button to a template column

In each case, you create a button control that has "Delete" as the **CommandName** property. Then you add code to the **DeleteCommand** event to delete the item from the data source.

To have the Property Builder create a Delete button

1. Add a **DataGrid** to your Web Forms page. For more information, see Adding DataGrid Web Server Controls to a Web Forms Page.
2. In Design view, select the **DataGrid** control, then click the **Property Builder** link at the bottom of the Properties window.
3. In the ***DataGrid* Properties** dialog box, click the **Columns** tab.
4. In the **Available columns** box, expand the **Button Column** item and choose **Delete**.
5. Click the **Add** button (>) to add the **Delete** column to the **Selected Columns** list.
6. The **Command name** property for the button is **Delete**. You must leave this property set to **Delete** if you want the click to raise the **DeleteCommand** event.
7. Set the rest of the button's properties as you want. For details about setting properties for button columns, see Adding Button Columns to a DataGrid Web Server Control.
8. Add code to the **DeleteCommand** event handler to delete the row from the data source and rebind the data to the **DataGrid**. See the section, "To delete the item from the data source."

To create a Delete button using the Property Builder

1. Add a **DataGrid** to your Web Forms page. For more information, see Adding DataGrid Web Server Controls to a Web Forms Page.
2. In Design view, select the **DataGrid** control, then click the **Property Builder** link at the bottom of the Properties window.
3. In the ***DataGrid* Properties** dialog box, click the **Columns** tab.
4. In the **Available columns** box, select the **Button Column** item.
5. Click the **Add** (>) button to add the **Button** column to the **Selected Columns** list.
6. Set the **Command name** property to **Delete**.
7. Set the rest of the buttons as you want. For details about setting properties for button columns, see Adding Button Columns to a DataGrid Web Server Control.
8. Add code to the **DeleteCommand** event handler to delete the row from the data source and rebind the data to the **DataGrid**. See the section below, "To delete the item from the data source."

To add a Delete button to a template column

1. Add a template column to your **DataGrid**. For information, see Adding Template Columns to a DataGrid Web Server Control.
2. In Design view, right-click the **DataGrid** and select **Edit Template**. Choose the template to which you want to add a Delete button.
3. Add a **Button**, **ImageButton**, or **LinkButton** control to the **ItemTemplate**.
4. Set the **CommandName** property of the button control to **Delete**.
5. Add code to the **DeleteCommand** event handler to delete the row from the data source and rebind the data to the **DataGrid**. See the following section, "To delete the item from the data source."

To delete the item from the data source

1. Create an event handler for the control's **DeleteCommand** event. For more information, see Creating Event Handlers in Web Forms Pages.

2. In the event handler, use the **e.Item.ItemIndex** property to identify the row to be deleted. Delete the row from the data source and rebind the data to the control. The code could look similar to the following:

```
' Visual Basic
Private Sub DataGrid1_DeleteCommand(ByVal source As Object, _
ByVal e As System.Web.UI.WebControls.DataGridCommandEventArgs) _
Handles DataGrid1.DeleteCommand
   Dim index As Integer
   index = e.Item.ItemIndex
   ' Add code to delete data from data source.
   DataGrid1.DataBind()
End Sub

// C#
private void DataGrid1_DeleteCommand(object source,
System.Web.UI.WebControls.DataGridCommandEventArgs e)
{
   int rowToDelete = e.Item.ItemIndex;
   // Add code to delete row from data source.
   DataGrid1.DataBind();
}
```

See Also

Adding DataGrid Web Server Controls to a Web Forms Page | DataGrid Web Server Control | Introduction to the DataGrid Web Server Control | Allowing Users to Select Items in a DataGrid Web Server Control

Allowing Users to Edit Items in a DataGrid Web Server Control

You can allow users to do in-place editing of information in grid rows. To do so, you create an **Edit, Update, Cancel** column using the Property Builder, and add code to the **EditCommand**, **UpdateCommand**, and **CancelCommand** events of the **DataGrid**. At run time, this column displays a button labeled "Edit". When the user clicks this Edit button, the row data is displayed in editable controls such as text boxes, and the Edit button is replaced with Update and Cancel buttons. Clicking the Update button raises the **UpdateCommand** event, where you add code to propagate the data changes back to your data source. Clicking the Cancel button raises the **CancelCommand** event, where you add code to rebind the original data to the **DataGrid**.

To add an Edit, Update, Cancel column

1. Add a **DataGrid** control to your Web Forms page. For details, see Adding DataGrid Web Server Controls to a Web Forms Page.
2. In Design view, select the **DataGrid** control, then click the **Property Builder** link at the bottom of the Properties window.
3. In the ***DataGrid* Properties** dialog box, click the **General** tab. If you want to associate data from a key field with each row in the grid, select a field name from the **Data key field** list. You can use this information to update a specific item in the data source. For example, if you set the **DataKeyField** to be **customerId**, you could later update the record using a statement such as the following:

```
' Visual Basic
Dim customerId as Integer
customerId=DataGrid1.DataKeys(DataGrid1.SelectedIndex)

// C#
int customerId;
customerId=(int)DataGrid1.DataKeys[DataGrid1.SelectedIndex];
this.Label1.Text = customerId.ToString();
```

4. Switch to the **Columns** tab and expand the **Button Column** node in the **Available columns** box.
5. Select **Edit, Update, Cancel** and click the **Add** button (>).
6. Set the header, footer, and sort information as needed. For more information, see Adding Button Columns to a DataGrid Web Server Control.
7. Use the **Edit text**, **Cancel text**, and **Update text** properties to specify the text that appears on the buttons.
8. Use the **Button type** property to specify whether the buttons are displayed as **PushButtons** or **LinkButtons**.
9. If you want the row in edit mode to have a special look, set the properties of the **EditItemStyle** object. For details, see Specifying Grid Item Format in a DataGrid Web Server Control.
10. You must create the **EditItemTemplate** for any template columns. For more information, see Adding Template Columns to a DataGrid Web Server Control.

When a user clicks a button, an event is raised on the **DataGrid** control corresponding to the button. You add code to implement the task according to the event as described in the following table:

DataGrid Event	Response
EditCommand	Put current row into edit mode by setting the **EditItemIndex** property.
UpdateCommand	Extract contents of current row and update the data source, cancel the edit mode of the row, and rebind the up dated data to the **DataGrid** control.
CancelCommand	Cancel edit mode of the row, and rebind the original data to the **DataGrid** control.

To create event handlers for Edit, Update, and Cancel events

1. Create an event handler for the control's **EditCommand** event. For more information, see Creating Event Handlers in Web Forms Pages. In the event handler, do the following:
   a. Set the **DataGrid** control's **EditItemIndex** property to the index value of the item to put into edit mode. The index of the item the user clicked is available via the **Item** property's **ItemIndex** property.
   b. Rebind the control to its data source.

   The event handler could look like this:

```
' Visual Basic
```

```
Private Sub DataGrid1_EditCommand(ByVal source As Object, _
ByVal e As System.Web.UI.WebControls.DataGridCommandEventArgs) _
Handles DataGrid1.EditCommand
   DataGrid1.EditItemIndex = e.Item.ItemIndex
   DataGrid1.DataBind()
End Sub

// C#
private void DataGrid1_EditCommand(object source,
System.Web.UI.WebControls.DataGridCommandEventArgs e)
{
   DataGrid1.EditItemIndex = e.Item.ItemIndex;
   DataGrid1.DataBind();
}
```

2. Create a method for the **UpdateCommand** event that reads the values of the editing controls in the current item and writes them back to the data source.

   Security Note User input in a Web Forms page can include potentially malicious client script. By default, the Web Forms page validates that user input does not include script or HTML elements. For more information, see Scripting Exploits and Protecting Against Script Exploits in a Web Application.

   - Bound columns The **e.Item** property contains an indexed **Cells** collection that contains one **Cell** object for each column in the **DataGrid** control. The data is found in the first control in the **Controls** collection of the cell. The following statements retrieve the string data from the **TextBox** control displayed in the second column:

```
' Visual Basic
Dim newData As String
Dim aTextBox As TextBox
aTextBox = CType(e.Item.Cells(1).Controls(0), TextBox)
newData = aTextBox.Text

// C#
string newData;
TextBox aTextBox;
aTextBox = (TextBox)(e.Item.Cells[1].Controls[0]);
newData = aTextBox.Text;
```

   - Template columns The **e.Item** property contains an indexed **Cells** collection that contains one **Cell** for each column in the **DataGrid**. The data is contained in controls you created while editing the template. Use the **FindControl** method and the control's name to retrieve the data. The following statements retrieve the string data from the **TextBox** control with **Name** property "FirstName" in the second column:

```
' Visual Basic
Dim newName As String
Dim box As TextBox
box = CType(e.Item.Cells(1).FindControl("FirstName"), TextBox)
newName = box.text

// C#
string newName;
TextBox box;
box = (TextBox)(e.Item.Cells[1].FindControl("FirstName"));
newName = box.Text;
```

   - After updating the data source, switch the item back out of editing mode by setting the **EditItemIndex** property to -1, and then rebind to the data source. A complete example follows:

```
' Visual Basic
Private Sub DataGrid1_UpdateCommand(ByVal source As Object, _
```

```
ByVal e As System.Web.UI.WebControls.DataGridCommandEventArgs) _
Handles DataGrid1.UpdateCommand
   ' The quantity is in the 7th column.
   Dim quantityCell As TableCell = e.Item.Cells(6)

   ' The TextBox is the 0th element of the Controls collection.
   Dim quantityBox As TextBox = _
      CType(quantityCell.Controls(0), TextBox)

   ' Extract the quantity from the box.
   Dim quantity As Integer = System.Int32.Parse(quantityBox.Text)

   ' Use quantity to update the data source.

   ' Switch out of edit mode.
   DataGrid1.EditItemIndex = -1
   DataGrid1.DataBind()
End Sub

// C#
private void DataGrid1_UpdateCommand(object source,
System.Web.UI.WebControls.DataGridCommandEventArgs e)
{
   // The quantity is in the 7th column.
   TableCell quantityCell = e.Item.Cells[6];

   // The TextBox is the 0th element of the Controls collection.
   TextBox quantityBox = (TextBox)quantityCell.Controls[0];

   // Extract the quantity from the box.
   int quantity = System.Int32.Parse(quantityBox.Text);

   // Use quantity to update the data source.

   // Switch out of edit mode.
   DataGrid1.EditItemIndex = -1;
   DataGrid1.DataBind();
}
```

3. Create a method for the **CancelCommand** event that switches out of editing mode by setting **EditItemIndex** property to –1 and then rebinds to the data source:

```
' Visual Basic
Private Sub DataGrid1_CancelCommand(ByVal source As Object, _
ByVal e As System.Web.UI.WebControls.DataGridCommandEventArgs) _
Handles DataGrid1.CancelCommand
   DataGrid1.EditItemIndex = -1
   DataGrid1.DataBind()
End Sub

// C#
private void DataGrid1_CancelCommand(object source,
System.Web.UI.WebControls.DataGridCommandEventArgs e)
{
   DataGrid1.EditItemIndex = -1;
   DataGrid1.DataBind();
}
```

See Also

Adding DataGrid Web Server Controls to a Web Forms Page | DataGrid Web Server Control | Introduction to the DataGrid Web Server Control | Allowing Users to Select Items in a DataGrid Web Server Control | Allowing Users to Delete Items in a DataGrid Web Server Control

Allowing Users to Select Items in a DataGrid Web Server Control

You can set up your **DataGrid** so that a user can select a particular row by clicking a button. When a row is selected, the **SelectedIndex** property of the **DataGrid** is set to the index of the selected row, and the row is displayed using the **Selected Item** format. There are three ways to create a button control that has "Select" as the **CommandName** property. You can:

- Have the Property Builder create the Select button
- Create the button yourself in the Property Builder
- Add the button to a template column

To have the Property Builder create a Select button

1. Add a **DataGrid** to your Web Forms page. For details, see Adding DataGrid Web Server Controls to a Web Forms Page.
2. In Design view, select the **DataGrid** control, then click the **Property Builder** link at the bottom of the Properties window.
3. In the ***DataGrid* Properties** dialog box, click the **Columns** tab.
4. In the **Available columns** box, expand the **Button Column** item and choose **Select**.
5. Click the **Add** button (>) to add the **Select** column to the **Selected columns** list.
6. The **Command name** property for the button is **Select**. You must leave this property set to **Select** if you want the row to display the **Selected Item** format when the user clicks the button.
7. Set the rest of the buttons as you want. For information about setting properties for Button Columns, see Adding Button Columns to a DataGrid Web Server Control.
8. When the grid is displayed, a column with buttons labeled "Select" will appear. When the user clicks the select button, the corresponding row will be formatted using the selected item format. For more information see, Specifying Grid Item Format in a DataGrid Web Server Control.

In the previous steps, the Property Builder created a button that would allow users to select a row. In the next procedure, you use the Property Builder to create a button, but you specify the properties so that the button implements the selection behavior.

To create a Select button using the Property Builder

1. Add a **DataGrid** to your Web Forms page. For more information, see Adding DataGrid Web Server Controls to a Web Forms Page.
2. In Design view, select the **DataGrid** control, then click the **Property Builder** link at the bottom of the Properties window.
3. In the ***DataGrid* Properties** dialog box, click the **Columns** tab.
4. In the **Available columns** box, click Button Column.
5. Click the **Add** (>) button to add the **Button** column to the **Selected Columns** list.
6. Set the **Command name** property to **Select**.
7. Set the rest of the buttons as you want. For details about setting properties for button columns, see Adding Button Columns to a DataGrid Web Server Control.
8. When the grid is displayed, a column with buttons labeled **Select** appears. When the user clicks the select button, the corresponding row will be formatted using the selected item format. For more information see, Specifying Grid Item Format in a DataGrid Web Server Control.

To add a Select button to a template column

1. Add a template column to your **DataGrid**. For details, see Adding Template Columns to a DataGrid Web Server Control.
2. In Design view, right-click the **DataGrid**, point to **Edit Template**, and choose the template to which you want to add a Select button.
3. Add a **Button**, **ImageButton**, or **LinkButton** control to the **ItemTemplate**.
4. Set the **CommandName** property of the button control to **Select**.
5. When the grid is displayed, a column with buttons labeled "Select" will appear. When the user clicks the select button, the corresponding row will be formatted using the selected item format. For details see Specifying Grid Item Format in a DataGrid Web Server Control.

See Also

DataGrid Columns | Adding DataGrid Web Server Controls to a Web Forms Page | DataGrid Web Server Control | Introduction to the DataGrid Web Server Control | Allowing Users to Edit Items in a DataGrid Web Server Control | Allowing Users to Delete Items in a DataGrid Web Server Control

Specifying Grid Item Format in a DataGrid Web Server Control

You can format items in the **DataGrid** Web server control to customize their appearance. You can:

- Set the color, font, borders, and spacing for the grid as a whole.
- Set the color, font, and alignment for each type of grid item (row) individually, such as item, alternating item, selected item, header, and footer. Changing these settings allows you to override the settings you make for the entire grid.
- Set the color, font, and alignment for individual columns. This is particularly useful for setting the appearance of a special-purpose column such as a button column.

> **Note** You can format the data in individual bound, button, and hyperlink columns by specifying a formatting expression. For details, see Adding Bound Columns to a DataGrid Web Server Control, Adding Button Columns to a DataGrid Web Server Control, and Adding Hyperlink Columns to a DataGrid Web Server Control. For information about creating a formatting expression, see Composite Formatting.

The formatting you set is applied in the order listed above. The grid as a whole is formatted first. If you specify an item format, that is applied next, and if you've specified a column style, that is applied last. For example, suppose you specify a format for the **Item** and **AlternatingItem** types. If you also specify a format for a column (for example, a button column) the button column format takes precedence, and the column is displayed with no difference between item and alternating item rows.

To set the format for the grid as a whole

1. In Design view, select the **DataGrid** control, then click the **Property Builder** link at the bottom of the Properties window.
2. In the ***DataGrid* Properties** dialog box, click the **Format** tab, and then choose **DataGrid** from the **Objects** list.
3. Choose font, color, and alignment options.

You can override the grid-wide settings for each type of item.

To set the format for an individual item

1. In Design view, select the **DataGrid** control, then click the **Property Builder** link at the bottom of the Properties window.
2. In the ***DataGrid* Properties** dialog box, click the **Format** tab, and under **Objects** do one of the following:
   - Select **Header**, **Footer**, or **Pager**.

     -or-

   - Expand the **Items** node and select the type of item to format.
3. Choose font, color, and alignment options for that item, and then choose **Apply**.

To set the format for an individual column

1. In Design view, select the **DataGrid** control, then click the **Property Builder** link at the bottom of the Properties window.
2. In the ***DataGrid* Properties** dialog box, click the **Format** tab, and under **Objects** expand the **Columns** node and select the column to format.
3. Expand the column and select whether you are formatting the column's header, footer, or content. If you do not specify formatting for a portion of the column (for example, you do not format the column's header), that portion is displayed with the defaults inherited from the grid or item.

> **Note** To remove column formatting, clear the formatting values.

See Also

Adding DataGrid Web Server Controls to a Web Forms Page | DataGrid Web Server Control | Introduction to the DataGrid Web Server Control

Paging Behavior in DataGrid Web Server Controls

The **DataGrid** Web server control allows you to add paging to the grid using these methods:

Method	Description
Default paging and default navigation buttons	The **DataGrid** control displays navigation buttons as part of the **DataGrid** control. These controls are either Next and Previous buttons or page numbers. When the user clicks these buttons, the grid updates the current page number and raises an event so you can refresh the data.
Default paging and custom navigation controls	You enable paging and set the page size of the **DataGrid** control. Then you place navigation controls, such as Next and Previous buttons, on the Web Forms page. In the click events for these buttons, you manipulate the current page of the **DataGrid** control. The control then selects the appropriate data rows to display.
Custom paging	You provide your own navigation controls, manually set which page to display, and provide the rows for just that page. This option allows you to move any number of pages at a time, jump to a specific page, and so on.

For details on specifying paging behavior, see Specifying Paging Behavior in a DataGrid Web Server Control.

Default versus Custom Paging

The page displayed by the grid is determined by its **CurrentPageIndex** property. The default paging controls set this property automatically; if you are providing custom paging controls, you set it yourself. After the **CurrentPageIndex** is set, the grid is re-bound to the data source. The grid recreates the entire dataset and automatically moves to the appropriate place in the dataset. It then displays enough rows to make one page of the grid.

If you are working with a large dataset, recreating the entire dataset each time users navigate to a new page can degrade performance. By using custom paging, you can retrieve data in page-size "chunks" — that is, retrieve just a page worth of records at a time. To do that, you turn off the automatic paging feature of the grid so that it doesn't assume that it is working with the entire dataset. You then get the correct number of rows to fill the grid. For information on implementing custom paging, see DataGrid.AllowCustomPaging Property.

See Also

Specifying Paging Behavior in a DataGrid Web Server Control | DataGrid Web Server Control | Introduction to the DataGrid Web Server Control

Specifying Paging Behavior in a DataGrid Web Server Control

The **DataGrid** supports three types of paging:

- Default paging with default navigation buttons
- Default paging with custom navigation controls
- Custom paging

For a discussion of these options, see Paging Behavior in DataGrid Web Server Controls.

Using Default Paging with Default Navigation Controls

To enable default paging, use the **Property Builder** to set properties to enable paging, set the page size, and specify the style of the paging controls. Paging controls are **LinkButton** controls. You can choose from these types:

- Next and previous buttons. The button captions can be any text you want.
- Page numbers, which allow users to jump to a specific page. You can specify how many numbers are displayed; if there are more pages, an ellipsis (...) is displayed next to the numbers.

You must also create an event handler that responds when users click a navigation control.

To specify paging using the Property Builder

1. In Design view, select the **DataGrid** control, then click the **Property Builder** link at the bottom of the Properties window.
2. In the ***DataGrid* Properties** dialog box, click the **Paging** tab.
3. Select the **Allow paging** box. A complete set of default paging properties is enabled.
4. Set the **Page size** property to the number of items to show on each page.
5. In the Mode list, choose to display either **Next, Previous buttons** or buttons with **Page numbers**.

   - To display "Next" and "Previous" on the navigation buttons, set the **Next page button text** and **Previous page button text** properties. The default text on the button is ">" for the next button and "<" for previous button. At run-time, the symbols "**<**" and "**>**" are displayed.

     -or-

   - Set the **Numeric buttons** property to the number of pages to be displayed. For example, if you set **Numeric buttons** to 2 (that is, display two page numbers at a time), then the user may select from pages "...", 3, 4, and "..." when page 3 is displayed.
   - If you are using page numbers, set the **PagerStyle** object's **PageButtonCount** property to the number of buttons to display.

Each of the settings in the above procedure, "To specify paging using the Property Builder", corresponds to a property of the **DataGrid**. These properties may be set in the **Page_Load** event.

To specify paging programmatically

1. Use the **Page_Load** event to enable paging and set the mode and button properties. The following example shows how you can set a **DataGrid** control called **DataGrid1** to use simple paging and respond to a navigation request. The code sets the pager style to display five numbers at a time when the page is first displayed. When users click a navigation control, the code simply rebinds to the data source.

```
' Visual Basic
Private Sub Page_Load(ByVal sender As System.Object, _
ByVal e As System.EventArgs) Handles MyBase.Load
   'Put user code to initialize the page here

   With DataGrid1
      ' Enable paging.
      .AllowPaging = True
      ' Display 5 page numbers at a time.
      .PagerStyle.Mode = PagerMode.NumericPages
      .PagerStyle.PageButtonCount = 5
```

```
        .PageSize = 5
    End With

    If Not Page.IsPostBack Then
        DataGrid1.DataBind()
    End If
End Sub

// C#
private void Page_Load(object sender, System.EventArgs e)
{
    // Put user code to initialize the page here
    DataGrid1.AllowPaging = true;
    DataGrid1.PagerStyle.Mode = PagerMode.NumericPages;
    DataGrid1.PagerStyle.PageButtonCount = 5;
    DataGrid1.PageSize = 5;

    if (!Page.IsPostBack)
    {
        DataGrid1.DataBind();
    }
}
```

2. Create an event handler for the grid's **PageIndexChanged** event to respond to a paging request. Use the **e.NewPageIndex** to update the **DataGrid**'s **CurrentPageIndex**, then rebind the data.

```
' Visual Basic
Private Sub DataGrid1_PageIndexChanged(ByVal source As Object, _
ByVal e As System.Web.UI.WebControls.DataGridPageChangedEventArgs) _
Handles DataGrid1.PageIndexChanged
    DataGrid1.CurrentPageIndex = e.NewPageIndex
    DataGrid1.DataBind()
End Sub

// C#
private void DataGrid1_PageIndexChanged(object source,
System.Web.UI.WebControls.DataGridPageChangedEventArgs e)
{
    DataGrid1.CurrentPageIndex = e.NewPageIndex;
    DataGrid1.DataBind();
}
```

Using Default Paging with Custom Navigation Controls

One of the options on the **Pager** tab of the **Property Builder** is **Show navigation buttons**. If this option is not selected, no navigation controls are displayed. In this case, you can provide your own navigation buttons and manipulate the **CurrentPageIndex** property of the **DataGrid**. The **DataGrid** will still take care of breaking the data source into appropriate pages and displaying the selected page.

To provide custom navigation controls

1. Add server controls to the Web Forms page that the user can use to navigate. For example, you may want to create **ImageButton** controls with forward and reverse images on them.
2. In event handlers for your navigation controls, set the **DataGrid** control's **CurrentPageIndex** property to the page to go to, and then rebind the grid to the data source.

   The following code shows how you can create code for a VCR-style navigation panel containing buttons that allow the user to go to the first, last, previous, or next page. The buttons are defined as **Button** controls whose **CommandName** property is set to indicate the navigation direction. All four buttons call the **NavigationButtonClicked** method when they are clicked.

Note In Visual C#, you can use the integrated development environment (IDE) to hook up the events and eliminate the
`+=`
assignments shown below. In this case, create the **NavigationButtonClick** method in the Code Editor. Return to the designer. Selecting each button in turn, select the **Events** tab in the Properties window. Select the **NavigationButtonClick** event from the drop-down list in the **Click** event.

```
' Visual Basic
Private Sub Page_Load(ByVal sender As System.Object, _
ByVal e As System.EventArgs) Handles MyBase.Load
   'Put user code to initialize the page here
   If Not Page.IsPostBack Then
      DataGrid1.DataBind()
   End If
End Sub

Private Sub NavigationButtonClicked(ByVal sender As Object, _
ByVal e As EventArgs) Handles FirstPage.Click, PreviousPage.Click, NextPage.Click, LastPa
ge.Click
   Dim direction As String = CType(sender, Button).CommandName

   Select Case direction.ToUpper()
      Case "FIRST"
         DataGrid1.CurrentPageIndex = 0
      Case "PREVIOUS"
         DataGrid1.CurrentPageIndex = _
            Math.Max(0, DataGrid1.CurrentPageIndex - 1)
      Case "NEXT"
          DataGrid1.CurrentPageIndex = _
             Math.Min(DataGrid1.PageCount - 1, _
             DataGrid1.CurrentPageIndex + 1)
      Case "LAST"
         DataGrid1.CurrentPageIndex = _
             DataGrid1.PageCount - 1
      Case Else
   End Select

   DataGrid1.DataBind()
End Sub

// C#
private void Page_Load(object sender, System.EventArgs e)
{
   // Put user code to initialize the page here.

   // Have all the buttons hook up to the same event.
   First.Click += new System.EventHandler(this.NavigationButtonClick);
   Previous.Click +=
      new System.EventHandler(this.NavigationButtonClick);
   Next.Click += new System.EventHandler(this.NavigationButtonClick);
   Last.Click += new System.EventHandler(this.NavigationButtonClick);
}

private void NavigationButtonClick(object sender, System.EventArgs e)
{
   string direction = ((Button)sender).CommandName;

   switch (direction.ToUpper())
   {
```

```
        case "FIRST" :
           DataGrid1.CurrentPageIndex = 0;
           break;
        case "PREVIOUS" :
           DataGrid1.CurrentPageIndex =
              Math.Max(DataGrid1.CurrentPageIndex -1, 0);
           break;
        case "NEXT" :
           DataGrid1.CurrentPageIndex =
              Math.Min(DataGrid1.CurrentPageIndex + 1,
              DataGrid1.PageCount - 1);
           break;
        case "LAST" :
           DataGrid1.CurrentPageIndex = DataGrid1.PageCount - 1;
           break;
        default :
           break;
     }
     DataGrid1.DataBind();
  }
```

Creating Custom Paging

By using the **AllowCustomPaging** property of the **DataGrid**, you can have complete control over which records are displayed. Custom paging improves performance by reducing the amount of data moved around in the system, since you can retrieve just one page of data at a time from the data source. For information on custom paging, see DataGrid.AllowCustomPaging Property. For an example, see Walkthrough: Creating Paged Data Access Using a Web Forms Page.

See Also

Adding DataGrid Web Server Controls to a Web Forms Page | DataGrid Web Server Control | Introduction to the DataGrid Web Server Control

Creating Templates Programmatically in the DataGrid Control

A **DataGrid** control can contain template columns in which you lay out the column by adding controls and literal text. Template columns provide you with substantially more flexibility than do bound columns, button columns, and hyperlink columns. As with templates for the **DataList** and **Repeater** controls, you can dynamically create templates for the **DataGrid** control. This allows you to define the contents, layout, and data of a column at run time.

> **Note** The procedures for working with a dynamic template column are similar to working with a dynamic template in a **Repeater** or **DataList** control. For general information on creating dynamic templates, see Creating Web Server Control Templates Dynamically.

The following are the differences between using a dynamic template in a **DataGrid** control and in the **Repeater** or **DataList** controls:

- You do not create item templates for the grid itself; instead, you create them for a column in the grid.
- There are slightly different templates for a **DataGrid** column than for a **Repeater** or **DataList** control. A **DataGrid** column does not include an alternating item or separator template. However, like the **DataList** control, it does include an **EditItem** template.

To create a dynamic template column

1. Create a template class that implements the **ITemplate** interface of the System.Web.UI namespace. For details, see Creating Web Server Control Templates Dynamically.
2. Optionally, pass into the class's constructor a value that the class can use to determine what type of template to create (**ItemTemplate**, **EditItemTemplate**, and so on).

   Typically, the template type is passed using a value from the **ListItemType** enumeration already defined in the **DataGrid** control.

3. In the class, implement the **InstantiateIn** method (the only member of the **ITemplate** interface). This method provides a way to insert an instance of text and controls into the specified container.

   The following example shows a template class for a dynamic template column. The constructor accepts two parameters: the first parameter specifies the template type to create and the second parameter allows you to pass in the name of the column you are creating. Because this is a template for a DataGrid control, the class includes code to create an **EditItem** template containing a **Textbox** control.

```
' Visual Basic
Private Class DataGridTemplate
   Implements ITemplate
   Dim templateType As ListItemType
   Dim columnName As String

   Sub New(ByVal type As ListItemType, ByVal ColName As String)
      templateType = type
      columnName = ColName
   End Sub

   Sub InstantiateIn(ByVal container As Control) _
      Implements ITemplate.InstantiateIn
      Dim lc As New Literal()
      Select Case templateType
         Case ListItemType.Header
            lc.Text = "<B>" & columnName & "</B>"
            container.Controls.Add(lc)
         Case ListItemType.Item
            lc.Text = "Item " & columnName
            container.Controls.Add(lc)
         Case ListItemType.EditItem
            Dim tb As New TextBox()
            tb.Text = ""
```

```
            container.Controls.Add(tb)
         Case ListItemType.Footer
            lc.Text = "<I>Footer</I>"
            container.Controls.Add(lc)
      End Select
   End Sub
End Class

// C#
public class DataGridTemplate : ITemplate
{
   ListItemType templateType;
   string columnName;

   public DataGridTemplate(ListItemType type, string colname)
   {
      templateType = type;
      columnName = colname;
   }

   public void InstantiateIn(System.Web.UI.Control container)
   {
      Literal lc = new Literal();
      switch(templateType)
      {
         case ListItemType.Header:
            lc.Text = "<B>" + columnName + "</B>";
            container.Controls.Add(lc);
            break;
         case ListItemType.Item:
            lc.Text = "Item " + columnName;
            container.Controls.Add(lc);
            break;
         case ListItemType.EditItem:
            TextBox tb = new TextBox();
            tb.Text = "";
            container.Controls.Add(tb);
            break;
         case ListItemType.Footer:
            lc.Text = "<I>" + columnName + "</I>";
            container.Controls.Add(lc);
            break;
      }
   }
}
```

After you have created the class for a dynamic template column, you can use it to assign columns to the **DataGrid** control at run time.

To use dynamic template columns

1. Create an instance of the TemplateColumn class.
2. Create an instance of your dynamic template, passing it an item type value if appropriate.
3. Assign the instance to one of the template properties of the **TemplateColumn** object you created in Step 1, such as **ItemTemplate**, **EditItemTemplate**, **HeaderTemplate**, and so on.

   The following example shows how to use the dynamic template column to add two columns to the **DataGrid** control. In this example, the templates are instantiated during the page load and before the control is bound to its data source.

```
' Visual Basic
Private Sub Page_Load(ByVal sender As System.Object, _
   ByVal e As System.EventArgs) Handles MyBase.Load
   Dim tc1 As New TemplateColumn()
   tc1.HeaderTemplate = New _
      DataGridTemplate(ListItemType.Header, "Column1")
   tc1.ItemTemplate = New DataGridTemplate(ListItemType.Item, _
      "Column1")
   tc1.EditItemTemplate = New _
      DataGridTemplate(ListItemType.EditItem, "Column1")
   tc1.FooterTemplate = New _
      DataGridTemplate(ListItemType.Footer, "Column1")
   DataGrid1.Columns.Add(tc1)

   Dim tc2 As New TemplateColumn()
   tc2.HeaderTemplate = New _
      DataGridTemplate(ListItemType.Header, "Column2")
   tc2.ItemTemplate = New _
      DataGridTemplate(ListItemType.Item, "Column2")
   tc2.EditItemTemplate = New _
      DataGridTemplate(ListItemType.EditItem, "Column2")
   tc2.FooterTemplate = New _
      DataGridTemplate(ListItemType.Footer, "Column2")
   DataGrid1.Columns.Add(tc2)
   SqlDataAdapter1.Fill(DsCategories1)
   DataGrid1.DataBind()
End Sub

// C#
private void Page_Load(object sender, System.EventArgs e)
{
   TemplateColumn tc1 = new TemplateColumn();
   tc1.HeaderTemplate = new
      DataGridTemplate(ListItemType.Header, "Column1");
   tc1.ItemTemplate = new
      DataGridTemplate(ListItemType.Item, "Column1");
   tc1.EditItemTemplate = new
      DataGridTemplate(ListItemType.EditItem, "Column1");
   tc1.FooterTemplate = new
      DataGridTemplate(ListItemType.Footer, "Column1");
   DataGrid1.Columns.Add(tc1);

   TemplateColumn tc2 = new TemplateColumn();
   tc2.ItemTemplate = new
      DataGridTemplate(ListItemType.Item, "Column2");
   tc2.HeaderTemplate = new
      DataGridTemplate(ListItemType.Header, "Column2");
   tc2.EditItemTemplate = new
      DataGridTemplate(ListItemType.EditItem, "Column2");
   tc2.FooterTemplate = new
      DataGridTemplate(ListItemType.Footer, "Column2");
   DataGrid1.Columns.Add(tc2);
   sqlDataAdapter1.Fill(dsCategories1);
   DataGrid1.DataBind();
}
```

See Also

DataList Web Server Control

The **DataList** Web server control displays rows of database information in customizable format. The format in which the data is displayed is defined in item, alternating item, selected item, and edit item templates. Header, footer, and separator templates are also available to let you customize the overall appearance of the **DataList**. By including **Button** Web server controls in the templates, you can connect the list items to code that allows users to switch between display, selection, and editing modes.

The following topics provide information that you will find useful in learning to work with the **DataList** control.

In This Section

Introduction to the DataList Web Server Control
Provides background information on the **DataList** Web server control, including what you can do with it and an overview of its features.

Adding DataList Controls to a Web Forms Page
Gives directions for adding a **DataList** Web server control to a Web Forms page and setting the required properties to make it work.

Allowing Users to Delete Items in a DataList Web Server Control
Gives directions for allowing users to delete either individual items or a batch of items.

Allowing Users to Edit Items in a DataList Web Server Control
Gives directions for adding editing features to items in the list.

Allowing Users to Select Items in a DataList Web Server Control
Gives directions for providing a Select button that users can click to highlight an item.

Creating Templates Dynamically in a DataList Web Server Control
Gives directions for creating a layout template at run time.

Customizing DataList Items at Run Time
Gives directions for dynamically controlling the contents and appearance of items.

Responding to Button Events in DataList, Repeater, or DataGrid Items
Gives directions for adding buttons to individual items that invoke custom functionality for list items.

Specifying Flow or Table Format in a DataList Web Server Control
Gives directions for displaying the list in either flow or table format.

Specifying Horizontal or Vertical Layout in a DataList Web Server Control
Gives directions for controlling how many columns the list has and how the list items are ordered within columns.

Related Sections

Controls for Displaying Lists On Web Pages
Lists the Web server controls you can use to display lists of data.

Programming Web Forms
Suggests additional topics on working with Web Forms.

Introduction to ASP.NET Server Controls
Provides general information on working with server controls in Web Forms.

Web Server Controls Templates
Provides background information on using templates with the **DataList** control.

Creating Web Server Control Templates

Describes how to format template items for all list controls.

ASP.NET Server Controls and CSS Styles
Provides information on specifying CSS styles in Web server controls.

Introduction to the DataList Web Server Control

The DataList Web server control displays data in a format that you can define using templates and styles. The **DataList** control is useful for displaying rows of data. Optionally, you can configure the **DataList** control to allow users to edit or delete information. You can also customize the control to support other functionality such as selecting rows.

> **Note** To precisely control the HTML used to render the list, use the **Repeater** Web server control. If you want to include paging and sorting automatically, use the **DataGrid** Web server control. For more information, see Controls for Displaying Lists On Web Pages.

The **DataList** control displays rows of data as items in the list. You use templates to define the layout of the items by including HTML text and controls. For example, you might use a **Label** Web server control in an item to display a field from the data source.

The following sections introduce you to the features of the **DataList** control.

Binding Data to the Control

The **DataList** control must be bound to a data source. Typical data sources for the **DataList** are datasets and data readers (OleDbDataReader or SqlDataReader objects). You can use data sources that you can add to your page from the Toolbox, such as the **DataSet** or DataView class. You can also bind to data sources created in code, such as a data reader or an array. The control can use as a data source any class that supports the IEnumerable interface.

When data binding, you specify a data source for the **DataList** control as a whole. When you add other controls to this control — for example, labels or text boxes in list items — you in turn bind them to the container, namely the **DataList** control.

For more information, see Data Access in Web Forms Pages.

Templates for Control Layout

In the **DataList** control, you define the layout of your information using templates.

> **Note** For an overview of using templates, see Web Server Controls Templates.

The **DataList** control supports the following templates:

Template	Description
ItemTemplate	The HTML elements and controls to render once for each row in the data source.
AlternatingItemTemplate	Like the **ItemTemplate** element, but rendered for every other row in the **DataList** control. If you use this template, you typically create a different look for it, such as a different background color than that of the **ItemTemplate**.
SelectedItemTemplate	The elements to render when the user selects an item in the **DataList** control. Typical uses are to visually mark the row using a background or font color. You can also expand the item by displaying additional fields from the data source.
EditItemTemplate	The layout of an item when it is in edit mode. This template typically contains editing controls, such as **TextBox** controls.
HeaderTemplate and **FooterTemplate**	The text and controls to render at the beginning and end of the list.
SeparatorTemplate	The elements to render between each item. A typical example might be a line (using an <HR> element).

Styles

To specify the appearance of items in a template, you can set the template's style. For example, you might specify that:

- Items are rendered with black text on a white background
- Alternating items are rendered with black text on a light gray background
- The selected item is rendered with bold black text on a yellow background
- Items being edited are rendered with black text on a light blue background

Each template supports its own style object whose properties you can set both at design time and at run time.

For an overview, see ASP.NET Server Controls and CSS Styles.

Item Layout

Not only can you use templates to specify the layout of controls and text within individual items, but you can specify how the items are laid out with respect to one another. The **DataList** control offers these options:

Option	Description
Flow versus table	In flow layout, list items are rendered inline, in the fashion of a word-processing document. In table layout, the items are rendered into an HTML table, which gives you more facilities for specifying the look of items, because it allows you to set table cell properties such as gridlines. For details see Specifying Flow or Table Format in a DataList Web Server Control.
Vertical versus horizontal ordering	By default, items in a **DataList** control are displayed in a single vertical column. However, you can specify that the control contain more than one column. If so, you can further specify whether the items are ordered vertically (like newspaper columns) or horizontally (like days in a calendar). For details see Specifying Horizontal or Vertical Layout in a DataList Web Server Control.
Number of columns	Regardless of whether items in a **DataList** control are ordered vertically or horizontally, you can specify how many columns the list will have. This permits you to control the rendered width of the Web page, usually so that horizontal scrolling may be avoided.

Events

The **DataList** control supports several events. One of them, the **ItemCreated** event, gives you a way to customize the item-creation process at run-time. The **ItemDataBound** event also gives you the ability to customize the **DataList** control, but after the data is available for inspection. For example, if you were using the **DataList** control to display a to-do list, you could display overdue items in red text, completed items in black text, and other tasks in green text. Either event may be used to override formatting from the template definition.

The remaining events are raised in response to button clicks in list items. They are designed to help you respond to the most commonly used functionality for the **DataList** control. Four events of this type are supported:

- **EditCommand**
- **DeleteCommand**
- **UpdateCommand**
- **CancelCommand**

When a user clicks a button in an item, the event bubbles to the button's container — the **DataList** control. The exact event that the button raises depends on the **CommandName** property of the button that was clicked. For example, if the **CommandName** property of a button is **edit**, the button causes an **EditCommand** event to be raised. If the **CommandName** property is **delete**, the button causes a **DeleteCommand** event to be raised, and so on.

The **DataList** control also supports the **ItemCommand** event that is raised when a user clicks a button that doesn't have a predefined command such as **edit** or **delete**. You can use this event for custom functionality by setting a button's **CommandName** property to a value you need and then testing for it in the **ItemCommand** event handler. (For example, you could use this approach when selecting an item, as documented in Allowing Users to Select Items in a DataList Web Server Control.)

For a complete list of the events supported by the **DataList** control, see DataList Events. For more information about Web Forms events, see ASP.NET Server Control Event Model.

Editing and Selecting Items

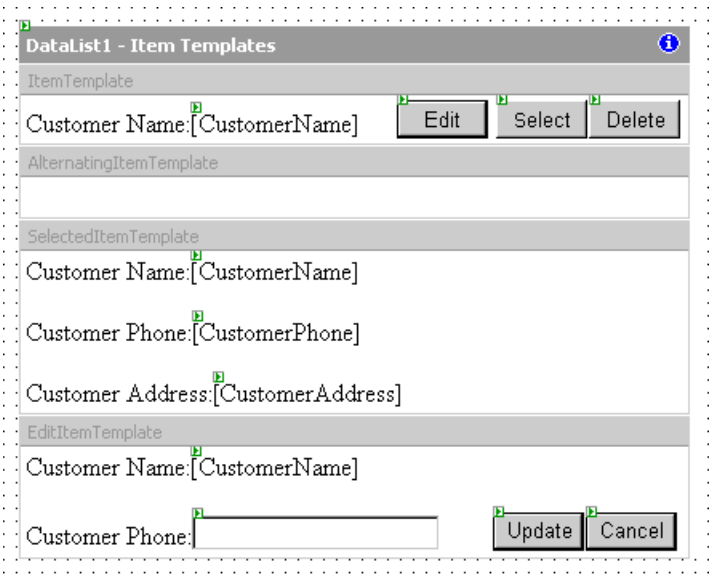
You can allow users to edit individual items in the control. The general strategy is to create an **EditItemTemplate** that provides the appropriate layout and controls for editing. You must also provide a way for users to indicate that they want to edit the item. The most common way is to include a button in the item template (and **AlternatingItemTemplate**, if you are using that) and set the button's **CommandName** property to **edit**. Then when the button is clicked, the **DataList** control automatically raises the **EditCommand** event. In the event handler for that event, you set the item to edit mode, which displays the **EditItemTemplate**.

The **EditItemTemplate** usually includes buttons to allow the user to save the changes or discard them (for example, Update and Cancel buttons). These buttons work similarly to the Edit button — they send a predefined command message (**update** or **cancel**) to the **DataList** control, which raises an **UpdateCommand** or **CancelCommand** event that you can respond to appropriately. For a customer tracking application, the templates might appear as in the following figure.

The process for selecting an item is similar, using the **SelectedIndexChanged** event. Your add a button to the Item template and sets its **CommandName** property to **select**. You then write an event handler for the **SelectedIndexChanged** event. The

SelectedIndexChanged event is raised when the user clicks the Select button. For details see Allowing Users to Select Items in a DataList Web Server Control.

The DataList in Edit Template Mode



See Also

Adding DataList Controls to a Web Forms Page | Specifying Flow or Table Format in a DataList Web Server Control | Specifying Horizontal or Vertical Layout in a DataList Web Server Control | Allowing Users to Select Items in a DataList Web Server Control | Allowing Users to Edit Items in a DataList Web Server Control | Allowing Users to Delete Items in a DataList Web Server Control

Adding DataList Controls to a Web Forms Page

To add a **DataList** Web server control to a page requires several steps. The following procedure describes the minimum that you must do to create a working **DataList** control.

To add a DataList Web server control to a Web Forms page

1. Add a data source to the Web Forms page. There are several methods for handling data in a Web application. To choose an appropriate method, see Introduction to Data Access with ADO.NET and Introduction to Data Access in Web Forms Pages.
2. In Design view, drag a **DataList** control from the **Web Forms** tab of the Toolbox onto the page.
3. Set the control's **DataSource** property. In the Properties window, the drop-down list for the **DataSource** property will display all the data sources, such as **DataSet** and **DataView** controls, defined on the page.
4. In Design view, right-click the **DataList** control, point to **Edit Template**, then click **Item Templates**. The control is displayed in template-editing mode.

   Note For background information about using templates, see Web Server Controls Templates.

5. Add HTML text and other controls (Web server controls or HTML server controls) to the **ItemTemplate**. For details on editing templates, see Creating Web Server Control Templates.

   Note You must add some data-bound content to one of the item templates in order for the **DataList** control to render when the page is browsed.

6. Optionally, define an **AlternatingItemTemplate**, **SelectedItemTemplate**, and **EditItemTemplate**.
7. When you are through editing templates, right-click the **DataList** control and choose **End Template Editing**.

See Also

Creating Templates Dynamically in a DataList Web Server Control | Adding Web Server Controls to a Web Forms Page | Web Server Controls Templates | Creating Web Server Control Templates | Allowing Users to Select Items in a DataList Web Server Control | Allowing Users to Edit Items in a DataList Web Server Control

Allowing Users to Delete Items in a DataList Web Server Control

You can allow users to delete items in a **DataList** control in a variety of ways. One way is to include a Delete button in an item and to delete that item immediately when the user clicks it.

Another way is to include a check box in individual items. Users can then check all the items they want to remove and then click a separate Delete button to delete them in a batch. This method is used in programs such as MSN Hotmail.

To allow users to delete individual items

1. In the **ItemTemplate** (and **AlternatingItemTemplate**, if you are using it), add a **Button** or **LinkButton** Web server control. Set the **CommandName** property for this button to **delete** (case-sensitive).
2. Create an event handler for the **DeleteCommand** event. For details, see Creating Event Handlers in Web Forms Pages. In the method:
   a. Delete the item from the data source and then rebind the **DataList** control. The index of the item the user clicked is available via the **Item** object's **ItemIndex** property. To get the value of individual controls, use the **FindControl** method of the **Item** event-argument object.
   b. Rebind the control to its data source.

   The event handler might look like this, if the data source were a **DataTable** object:

```
' Visual Basic
Private Sub DataList1_DeleteCommand(ByVal source As Object, _
ByVal e As System.Web.UI.WebControls.DataListCommandEventArgs) _
Handles DataList1.DeleteCommand
   ' Code to delete the item from the data source.
   Dim aTable As DataTable
   aTable = CType(DataList1.DataSource, DataTable)
   aTable.Rows(e.Item.ItemIndex).Delete()
   ' Bind the data after item is deleted.
   DataList1.DataBind()
End Sub

// C#
private void DataList1_DeleteCommand(object source,
System.Web.UI.WebControls.DataListCommandEventArgs e)
{
   // Code to delete the item from the data source.
   DataTable aTable = (DataTable)DataList1.DataSource;
   aTable.Rows[e.Item.ItemIndex].Delete();
   // Bind the data after the item is deleted.
   DataList1.DataBind();
}
```

To allow users to delete multiple items at once

1. In the **ItemTemplate** (and **AlternatingItemTemplate**, if you are using that), add a **CheckBox** Web server control and set its **ID** property to "Delete". Make sure that the **AutoPostBack** property of the **CheckBox** control is **false**. Turn off template editing by right-clicking the **DataList** control and selecting **End Template Editing** from the menu.
2. Add a **Button** Web server control to the Web Forms page. Set the **Text** property to **Delete All** and the ID property to **DeleteAll**. This button is not added to one of the **DataList** templates.
3. Create a method for the **Click** event of the **Delete All** button. In the method:
   a. Loop through the **Items** collection of the **DataList** control, extracting each item in turn.
   b. Within the item, use the item's **FindControl** method to get the **CheckBox** control from Step 1 and test its **Checked** property.
   c. If the box is checked, delete the corresponding item from the data source.

d. Rebind the **DataList** control to its data source.

The following example shows an event handler for the **DeleteAll** button that deletes items in a batch using the procedure outlined above.

```
' Visual Basic
Private Sub DeleteAll_Click(ByVal sender As Object, _
ByVal e As System.EventArgs) Handles DeleteAll.Click
   Dim isDeleted As Boolean
   Dim anItem As DataListItem

   ' Check each box and see if the item should be deleted.
   For Each anItem In DataList1.Items
      isDeleted = CType(anItem.FindControl("Delete"), CheckBox).Checked
      If isDeleted Then
         ' Add code here to delete the item, using anItem.ItemIndex.
      End If
   Next
   DataList1.DataBind()
End Sub

// C#
protected void DeleteAll_Click(object sender, System.EventArgs e)
{
   bool isDeleted;

   // Check each box and see if the item should be deleted.
   foreach (DataListItem anItem in DataList1.Items)
   {
      isDeleted =
         ((CheckBox)anItem.FindControl("Delete")).Checked;
      if (isDeleted)
      {
         // Add code here to delete the item, using anItem.ItemIndex.
      }
   }
   DataList1.DataBind();
}
```

See Also

Introduction to the DataList Web Server Control | Allowing Users to Edit Items in a DataList Web Server Control | Allowing Users to Select Items in a DataList Web Server Control | Creating Event Handlers in Web Forms Pages | Responding to Button Events in DataList, Repeater, or DataGrid Items

Allowing Users to Edit Items in a DataList Web Server Control

You can allow users to edit individual items in the **DataList** Web server control. When an individual item is put into edit mode, its editable values are usually displayed in text boxes or other controls that the user can change values in.

To allow users to edit items in a DataList control

1. Create an **ItemTemplate** (and **AlternatingItemTemplate**, if you are using that) and add a **Button** Web server control to it. Set the **CommandName** property for this button to **edit** (case-sensitive). For details see Creating Web Server Control Templates.

   Note You can use a **LinkButton** or an **ImageButton** in any step that calls for a **Button**.

2. Create an **EditItemTemplate** for the **DataList** control that includes the following:

   - Controls for all the values that the user can change. For example, include **TextBox** controls for all character and numeric data. Use the **DataBinding** property to set the data source for these controls. For information, see Data Access in Web Forms Pages.
   - A **Button** control with the **Text** property "Update" whose **CommandName** property is set to **update** (case-sensitive).
   - A **Button** control with the **Text** property "Cancel" whose **CommandName** property is set to **cancel** (case-sensitive).

   The Update button will allow users to specify that they are finished editing. The Cancel button will allow them to quit editing without saving changes.

3. Create an event handler for the control's **EditCommand** event. For details, see Creating Event Handlers in Web Forms Pages. In the event handler, do the following:
   a. Set the **DataList** control's **EditItemIndex** property to the index value of the item to put into edit mode. The index of the item the user clicked is available via the **Item** object's **ItemIndex** property.
   b. Bind the control to its data source.

   The event handler might look like the following:

```
' Visual Basic
Private Sub DataList1_EditCommand(ByVal source As Object, _
ByVal e As System.Web.UI.WebControls.DataListCommandEventArgs) _
Handles DataList1.EditCommand
   DataList1.EditItemIndex = e.Item.ItemIndex
   DataList1.DataBind()
End Sub

// C#
private void DataList1_EditCommand(object source,
System.Web.UI.WebControls.DataListCommandEventArgs e)
{
   DataList1.EditItemIndex = e.Item.ItemIndex;
   DataList1.DataBind();
}
```

4. Create a method for the **UpdateCommand** event that reads the values of the editing controls in the current item and writes them back to the data source. To get the value of a specific control in the item, use the Control.FindControl method of the **Item** event-argument object. After updating the data source, switch the item out of editing mode by setting the **EditItemIndex** property to -1, and then rebind to the data source.

   Security Note User input in a Web Forms page can include potentially malicious client script. By default, the Web Forms page validates that user input does not include script or HTML elements. For more information, see Scripting Exploits and Protecting Against Script Exploits in a Web Application.

```
' Visual Basic
Private Sub DataList1_UpdateCommand(ByVal source As Object, _
ByVal e As System.Web.UI.WebControls.DataListCommandEventArgs) _
```

```
Handles DataList1.UpdateCommand
   Dim quantity As TextBox
   quantity = CType(e.Item.FindControl("quantity"), TextBox)
   Dim newQuantity As String = quantity.Text
   ' Add code here to update the Quantity value of the data source
   ' using the newQuantity variable. Note that newQuantity is a
   ' string and would have to be converted to a number.
   DataList1.EditItemIndex = -1
   DataList1.DataBind()
End Sub

// C#
private void DataList1_UpdateCommand(object source,
System.Web.UI.WebControls.DataListCommandEventArgs e)
{
   TextBox quantity;
   quantity = (TextBox)(e.Item.FindControl("Quantity"));
   string newQuantity = quantity.Text;
   // Add code here to update the Quantity value of the data source
   // using the newQuantity variable. Note taht newQuantity is a
   // string and would have to be converted to a number.
   DataList1.EditItemIndex = -1;
   DataList1.DataBind();
}
```

5. Create a method for the **CancelCommand** event that sets the **EditItemIndex** property to -1 and then rebinds to the data source:

```
' Visual Basic
Private Sub DataList1_CancelCommand(ByVal source As Object, _
ByVal e As System.Web.UI.WebControls.DataListCommandEventArgs) _
Handles DataList1.CancelCommand
   ' Switch out of edit mode.
   DataList1.EditItemIndex = -1
   DataList1.DataBind()
End Sub

// C#
private void DataList1_CancelCommand(object source,
System.Web.UI.WebControls.DataListCommandEventArgs e)
{
   // Switch out of edit mode.
   DataList1.EditItemIndex = -1;
   DataList1.DataBind();
}
```

See Also

Introduction to the DataList Web Server Control | Allowing Users to Delete Items in a DataList Web Server Control | Allowing Users to Select Items in a DataList Web Server Control | Creating Event Handlers in Web Forms Pages | Responding to Button Events in DataList, Repeater, or DataGrid Items

Allowing Users to Select Items in a DataList Web Server Control

You can specify that users can select individual items in the **DataList** Web server control. Typically, selecting an item highlights it visually. In addition, you might display different information for a selected item.

To allow users to select items in a DataList control

1. Set the control's **SelectedItemStyle** properties. For details, see Customizing the Appearance of ASP.NET Server Controls Using Styles.
2. Optionally, create a **SelectedItemTemplate** to display different text and controls for the selected item. For details, see Creating Web Server Control Templates.
3. In the **ItemTemplate** (and **AlternatingItemTemplate**, if you are using it), add a **Button** or **LinkButton** Web server control.
4. Set the **CommandName** property of the button from Step 3 to **select** (case-sensitive).
5. If you have created a separate **SelectedItemTemplate**, you need to rebind the list upon selection. Create an event handler for the control's **SelectedIndexChanged** event. For details, see Creating Event Handlers in Web Forms Pages. In the event handler, call the control's **DataBind** method to refresh the information in the control. The complete code would look something like this:

```
' Visual Basic
Private Sub DataList1_SelectedIndexChanged( _
ByVal sender As System.Object, ByVal e As System.EventArgs) _
Handles DataList1.SelectedIndexChanged
   DataList1.DataBind()
End Sub

// C#
private void DataList1_SelectedIndexChanged (object sender, System.EventArgs e)
{
   DataList1.DataBind();
}
```

6. To cancel the selection, set the control's **SelectedIndex** property to -1. To accomplish this, you could add a **Button** Web server control to the **SelectedItem** template and set its **CommandName** property to **unselect**. The **Click** event for this button would also be bubbled to the **DataGrid** control's **ItemCommand** event. The complete code might look something like the following:

```
' Visual Basic
Private Sub DataList1_ItemCommand(ByVal source As Object, _
ByVal e As System.Web.UI.WebControls.DataListCommandEventArgs) _
Handles DataList1.ItemCommand
   If e.CommandName = "unselect" Then
      DataList1.SelectedIndex = -1
   End If
   DataList1.DataBind()
End Sub

// C#
private void DataList1_ItemCommand(object source, System.Web.UI.WebControls.DataListComma
ndEventArgs e)
{
   if (e.CommandName == "unselect")
   {
      DataList1.SelectedIndex = -1;
   }
   DataList1.DataBind();
}
```

See Also

Introduction to the DataList Web Server Control | Allowing Users to Delete Items in a DataList Web Server Control | Allowing Users to Edit Items in a DataList Web Server Control | Creating Event Handlers in Web Forms Pages | Responding to Button Events in DataList, Repeater, or DataGrid Items

Creating Templates Dynamically in a DataList Web Server Control

Templates do not have to be assigned at design time. In some situations, you might be able to lay out the template at design time, but changes you know you will make during run time are so extensive that loading a new template at run time simplifies the programming. In other cases, you may have several possible templates, but want to change the template at run-time.

One strategy is to create a template file with the basic template layout in it — a template definition file. To load a template at run time, use the **Page.LoadTemplate** method. This method reads a template definition from a file and creates an ITemplate Interface object. You can then assign this object to any of the templates in a **DataList**.

To create a template definition file

1. On the **Project** menu, click **Add New Item**.
2. In the **Add New Item** dialog box, select **Text File**. Name the file with the ".ascx" extension.
3. Add template definition statements to the template file and save it.

   The following listing creates a template that displays the string "Name:" followed by the "LastName" field of the data source that is bound to the **DataList**.

```
<!-- Visual Basic -->
<%@ Language = "VB" %>
Name: <b> <%# DataBinder.Eval(Container, "DataItem.LastName") %> </b>

<!-- Visual C# -->
<%@ Language = "C#" %>
Name: <b> <%# DataBinder.Eval(Container, "DataItem.LastName") %> </b>
```

To create templates dynamically

- Add code to your Web Forms page to load the template using the **Page.LoadTemplate** method.

  The following code uses the **Page_Load** event handler to load a template created as shown above and named NewTemplate.ascx.

```
' Visual Basic
Private Function CreateDataSource() As DataTable
   Dim dt As DataTable = New DataTable()
   dt.Columns.Add("LastName", Type.GetType("System.String"))
   dt.Rows.Add(New Object() {"Smith"})
   dt.Rows.Add(New Object() {"Jones"})
   dt.Rows.Add(New Object() {"Clark"})
   Return dt
End Function

Protected Sub Page_Init(ByVal Sender As System.Object, _
ByVal e As System.EventArgs) Handles MyBase.Init
   InitializeComponent()
   If (Not Page.IsPostBack) Then
      DataList1.AlternatingItemTemplate = _
      Page.LoadTemplate("NewTemplate.ascx")
   End If
End Sub

Private Sub Page_Load(ByVal sender As System.Object, _
ByVal e As System.EventArgs) Handles MyBase.Load
   'Put user code to initialize the page here
   If (Not Page.IsPostBack) Then
      DataList1.DataSource = CreateDataSource()
```

```
      DataList1.DataBind()
   End If
End Sub

// C#
private DataTable CreateDataSource()
{
   DataTable dt = new DataTable();
   dt.Columns.Add("LastName", Type.GetType("System.String"));
   dt.Rows.Add(new object[] {"Smith"});
   dt.Rows.Add(new object[] {"Jones"});
   dt.Rows.Add(new object[] {"Clark"});
   return dt;
}

private void Page_Init(object sender, EventArgs e)
{
   InitializeComponent();
   DataList1.AlternatingItemTemplate =
   Page.LoadTemplate("NewTemplate.ascx");
}

private void Page_Load(object sender, System.EventArgs e)
{
   // Put user code to initialize the page here
   if (!Page.IsPostBack)
   {
      DataList1.DataSource = CreateDataSource();
      DataList1.DataBind();
   }
}
```

See Also

TemplateControl.LoadTemplate Method | Web Server Controls Templates | DataList Web Server Control | Customizing DataList Items at Run Time

Customizing DataList Items at Run Time

During page processing, the **DataList** control creates individual items that it will render to the page. To allow you to customize the items as they are being created, the **DataList** control raises two events:

- The **ItemDataBound** event is raised only for items, alternating items, and selected items. Use this event if the customization depends on the data.
- The **ItemCreated** event is raised for all items, including the header and footer. The event is raised for these items even if you have not created templates for them, in case you want to create them dynamically. For information on dynamically creating templates, see Creating Templates Dynamically in a DataList Web Server Control.

By responding to these events, you can change the items. A typical use is to change the appearance of the item based on the data being displayed in it. For example, if a numeric value you read from a database is negative, you might set the item's background color to red.

To customize items at run time using the ItemCreated event

1. Create an event handler for the **DataList** control's **ItemCreated** event. For details, see Creating Event Handlers in Web Forms Pages.
2. In the event handler, get the current item using the **Item** object from the *e* event-argument object. The index of the current item is available via the **Item** object's **ItemIndex** property, and its type is available via its **ItemType** property (which is defined using the **ListItemType** enumeration).

   > **Note** The **ItemIndex** property returns a positive value only for **Item**, **AlternatingItem**, and **SelectedItem** objects. The index value for header, footer, and separator items is -1.

   The following example shows how you can modify the background color of an item conditionally. The example uses the **ItemType** property and the **ListItemType** enumeration to set the background colors of the items. Items are set to **LightGoldenrodYellow**, alternating items are set to **DarkGoldenRod**, and the selected item is set to **Yellow**.

```
' Visual Basic
Private Sub DataList1_ItemCreated(ByVal sender As Object, _
ByVal e As System.Web.UI.WebControls.DataListItemEventArgs) _
Handles DataList1.ItemCreated
   Select Case e.Item.ItemType
      Case ListItemType.Item
         e.Item.BackColor = Color.LightGoldenrodYellow
      Case ListItemType.AlternatingItem
         e.Item.BackColor = Color.DarkGoldenrod
      Case ListItemType.SelectedItem
         e.Item.BackColor = Color.Yellow
      Case Else
         ' Add code here to hand the header, footer, and
         ' separator templates.
   End Select
End Sub

// C#
private void DataList1_ItemCreated(object sender,
System.Web.UI.WebControls.DataListItemEventArgs e)
{
   switch (e.Item.ItemType)
   {
      case System.Web.UI.WebControls.ListItemType.Item :
         e.Item.BackColor = Color.LightGoldenrodYellow;
         break;
      case System.Web.UI.WebControls.ListItemType.AlternatingItem :
         e.Item.BackColor = Color.DarkGoldenrod;
         break;
      case System.Web.UI.WebControls.ListItemType.SelectedItem :
```

```
            e.Item.BackColor = Color.Yellow;
            break;
        default :
            // Add code here to handle the header, footer, and
            // separator templates.
            break;
        }
    }
```

At the time that the **ItemCreated** event is raised, the individual controls in the templates have not yet been bound to data. However, the data is available in the **ItemDataBound** event.

To customize items at run time using the ItemDataBound event

1. Create an event handler for the **DataList** control's **ItemDataBound** event. For details, see Creating Event Handlers in Web Forms Pages.
2. In the event handler, use the **e.Item.DataItem** property to test the data that will be bound to the control and base a comparison on that.

   The following example shows how you can make a conditional test based on data. The example extracts the third field, Quantity field, from the **DataItem**. If the quantity is less than 10, the item in the **DataList** control is colored red.

```
' Visual Basic
Private Sub DataList1_ItemDataBound(ByVal sender As Object, _
ByVal e As System.Web.UI.WebControls.DataListItemEventArgs) _
Handles DataList1.ItemDataBound
   Dim drv As DataRowView = CType(e.Item.DataItem, DataRowView)
   Dim quantity As Integer = _
      Integer.Parse(drv.Row("Quantity").ToString())
   If (quantity < 10) Then
      e.Item.BackColor = Color.Red
   End If
End Sub

// C#
private void DataList1_ItemDataBound(object sender,
System.Web.UI.WebControls.DataListItemEventArgs e)
{
   DataRowView drv = (DataRowView)(e.Item.DataItem);
   int quantity = int.Parse(drv.Row["Quantity"].ToString());
   if (quantity < 10)
   {
      e.Item.BackColor = Color.Red;
   }
}
```

See Also

Introduction to the DataList Web Server Control | Creating Event Handlers in Web Forms Pages

Responding to Button Events in DataList, Repeater, or DataGrid Items

If your **DataList**, **Repeater**, or **DataGrid** control templates include **Button**, **LinkButton**, or **ImageButton** Web Server controls, then these buttons can bubble their click events to the containing **DataList**, **Repeater**, or **DataGrid** control. This allows you to include buttons for functions not already defined for the **DataList** and **DataGrid** controls (edit, delete, update, and cancel) and to define functionality for the **Repeater** control.

To respond to button events in DataList, Repeater, and DataGrid controls

1. Include a **Button**, **LinkButton**, or **ImageButton** in a control template. Set the button's **CommandName** property to a string that identifies its function, such as "sort" or "copy".
2. Create a method for the **ItemCommand** event of the control. In the method, do the following:
   a. Check the **CommandName** property of the event-argument object's **CommandSource** property to see what string was passed.
   b. Perform the appropriate logic for the button that the user clicked.

   > **Note** If you call the **DataBind** method of the parent control (the **DataList**, **Repeater**, or **DataGrid** control), the **ItemCommand** event is not raised, because the contents of the parent control are reset. Therefore, you generally do not want to call the **DataBind** method on each round trip (that is, in the page initialization without checking for a post back). For details, see Data Binding Multi-Record Web Server Controls and Data Binding Single-Value Web Server Controls at Design Time.

   The following example shows how you can respond to a button click in a **DataList** control. In the example, the **ItemTemplate** contains an **ImageButton** control that displays a shopping cart. The button sends the command **AddToCart**. The **ItemCommand** event handler determines which button was clicked, and — if it was the shopping cart button — performs the appropriate logic.

```
' Visual Basic
Private Sub DataList1_ItemCommand(ByVal source As Object, _
ByVal e As System.Web.UI.WebControls.DataListCommandEventArgs) _
Handles DataList1.ItemCommand
   If (e.CommandName = "AddToCart") Then
      ' Add code here to add the item to the shopping cart.
      ' Use the value of e.Item.ItemIndex to find the data row
      ' in the data source.
    End If
End Sub

// C#
private void DataList1_ItemCommand(object source,
System.Web.UI.WebControls.DataListCommandEventArgs e)
{
   if (e.CommandName == "AddToCart")
   {
      // Add code here to add the item to the shopping cart.
      // Use the value of e.Item.ItemIndex to find the data row
      // in the data source.
   }
}
```

For an example using the **DataList** Web server control, see Allowing Users to Select Items in a DataList Web Server Control.

See Also

Introduction to the DataList Web Server Control | Introduction to the Repeater Web Server Control | Introduction to the DataGrid Web Server Control | Allowing Users to Delete Items in a DataList Web Server Control |

Allowing Users to Edit Items in a DataList Web Server Control | Allowing Users to Select Items in a DataList Web Server Control | Creating Event Handlers in Web Forms Pages

Specifying Flow or Table Format in a DataList Web Server Control

The **DataList** control allows you to display the items in the control in a variety of ways. You can specify that the control renders items in either flow mode (like a Word document) or in table mode (like an HTML table). Flow mode is suitable for simple layouts. Table mode gives you more precise control over layout and allows you to use table attributes (such as **CellPadding**).

To specify flow or table format at design time

- Set the **DataList** control's **RepeatLayout** property to **Flow** or **Table**.

To specify flow or table format programmatically

- Use the RepeatLayout enumeration, as shown in the following example:

```
' Visual Basic
DataList1.RepeatLayout = RepeatLayout.Flow

// C#
DataList1.RepeatLayout = RepeatLayout.Flow;
```

See Also

Introduction to the DataList Web Server Control | Specifying Horizontal or Vertical Layout in a DataList Web Server Control

Specifying Horizontal or Vertical Layout in a DataList Web Server Control

By default, the **DataList** Web server control displays a single column. However, you can specify any number of columns. Within the columns, you can specify how the items are ordered: vertically (the default) or horizontally. Vertical layout in three columns results in a layout such as the following:

```
1     4     7
2     5     8
3     6
```

Horizontal layout for the same items results in the following layout:

```
1     2     3
4     5     6
7     8
```

To specify column count and ordering at design time

1. Set the **DataList** control's **RepeatColumns** property to the number of columns you want.
2. Set the **RepeatDirection** property to **Vertical** or **Horizontal**.

To specify column count and ordering programmatically

- Specify the column count using the **RepeatColumns** property, and specify the ordering using the RepeatDirection enumeration, as shown in the following example:

```
' Visual Basic
DataList1.RepeatColumns = 2
DataList1.RepeatDirection = RepeatDirection.Horizontal

// C#
DataList1.RepeatColumns = 2;
DataList1.RepeatDirection = RepeatDirection.Horizontal;
```

See Also

Introduction to the DataList Web Server Control | Specifying Flow or Table Format in a DataList Web Server Control

DropDownList Web Server Control

The **DropDownList** Web server control enables users to select from a single-selection drop-down list box. The **DropDownList** control is very similar to the ListBox Web server control. It differs in that it shows only the selected item in a box, along with a drop-down button. When users click the button, a list of items is displayed.

The following table lists topics that provide information about how to work with the **DropDownList** control. Because the control is similar in many ways to the **ListBox** control, many of the links below lead you to topics for that control.

In This Section

Introduction to the DropDownList Web Server Control
Explains the basic uses of the **DropDownList** control.
Adding DropDownList Controls to a Web Forms Page
Gives directions for putting a **DropDownList** control onto a Web Form.

Related Sections

Adding Web Server Controls to a Web Forms Page
Gives directions for including controls on your Web Form.
Setting ASP.NET Server Control Properties
Suggests additional topics of interest concerning server control properties.
Setting Web Server Control Properties Programmatically
Gives directions for setting server control properties in code.
Customizing the Appearance of ASP.NET Server Controls Using Styles
Gives directions for changing the look of your server controls on the page.
Adding Items in a List Web Server Control
Gives directions for defining individual items in the **DropDownList** control, either at design time or programmatically at run time.
Populating a List Web Server Control from a Database
Gives directions for using a **DropDownList** control to display database information.
Determining the Selection in a List Web Server Control
Gives directions for checking in code what the user has selected.
Setting the Selection in a List Web Server Control
Gives directions for preselecting items in the control.

Introduction to the DropDownList Web Server Control

The DropDownList Web server control allows users to select one or more items from a predefined list. It differs from the ListBox Web server control in that the list of items remains hidden until users click the drop-down button. In addition, the **DropDownList** control differs from the **ListBox** control in that it does not support multi-selection mode.

DropDownList Control Appearance

You can control the look of the **DropDownList** control by setting its height and width in pixels. Some browsers do not support setting the height and width in pixels and will use the row-count setting.

You cannot specify the number of items that are displayed in the list when users click the drop-down button. The length of the displayed list is determined by the browser.

As with other Web server controls, you can use style objects to specify the appearance of the control. For details, see Customizing the Appearance of ASP.NET Server Controls Using Styles.

Items

The **DropDownList** control is actually a container for the list items. Each item is a separate object with its own properties:

Property	Description
Text	The text displayed in the list.
Value	The value associated with an item. Setting this property allows you to associate a value with a specific item without displaying it. For example, you can set the **Text** property to the name of a U.S. state and the **Value** property to its postal abbreviation.
Selected	A Boolean value indicating whether the item is selected.

To work with items programmatically, you work with the Items collection of the **DropDownList** control. **Items** is a standard collection, and you can add item objects to it, delete items, clear the collection, and so on.

The currently selected item is available in the **DropDownList** control's SelectedItem property.

Binding Data to the Control

You can use a **DropDownList** Web server control to list options that are read from a data source. Each item in the **DropDownList** control corresponds to an item — typically a row — in the data source.

The control displays one field from the source. Optionally, you can bind the control to a second field to set the value (which does not display) of an item.

As with other server controls, you can bind any control properties, such as the color or size of the control, to data. For details, see Web Forms Data Binding.

DropDownList Events

The **DropDownList** control raises an event — SelectedIndexChanged — when users select an item. By default, this event does not cause the page to be posted to the server, but you can cause the control to force an immediate post by setting the AutoPostBack property to **true**.

> **Note** The ability of a **DropDownList** control to post to the server when it is checked requires that the browser support ECMAScript (JScript, JavaScript) and that scripting be enabled on the user's browser.

See Also

Adding DropDownList Controls to a Web Forms Page | Adding Items in a List Web Server Control | Populating a List Web Server Control from a Database | Determining the Selection in a List Web Server Control | Setting the Selection in a List Web Server Control

Adding DropDownList Controls to a Web Forms Page

The process of adding a **DropDownList** Web server control to a page consists primarily of setting options for how items should be displayed and whether users can select multiple items.

To add a DropDownList control to a Web Forms page

- In **Design** view, drag the **DropDownList** control from the Toolbox onto the page.

Adding items to the control is a separate process that depends on whether you will display a static list in the control or a list generated dynamically at run time. For details, see Adding Items in a List Web Server Control.

See Also

Adding Web Server Controls to a Web Forms Page | DropDownList Class

HyperLink Web Server Control

The **HyperLink** Web server control provides a means of creating and manipulating links on a Web page from server code. The following topics provide information that you will find useful in learning to work with the **HyperLink** control.

In this Section

Introduction to the HyperLink Web Server Control
Provides background information on the **HyperLink** control.
Adding HyperLink Web Server Controls to a Web Forms Page
Gives directions for adding a **HyperLink** Web server control to a Web Forms page and setting its appearance.

Related Sections

Adding Controls to a Web Forms Page Programmatically
Gives directions for using code to add **HyperLink** controls to a page.
Adding Hyperlink Columns to a DataGrid Web Server Control
Gives directions for using **HyperLink** controls as part of a data grid.
Setting Web Server Control Properties Programmatically
Gives directions for specifying Web server control properties in code.
Customizing the Appearance of ASP.NET Server Controls Using Styles
Gives directions for applying styles to change how controls are displayed.
Web Forms Data Binding
Provides basic conceptual information about binding data to Web Forms.

Introduction to the HyperLink Web Server Control

The HyperLink Web server control creates links on a Web page that allow users to move from page to page in your application. The primary advantage of using a **HyperLink** control is that you can set link properties in server code. For example, you can dynamically change the link text or target page based on conditions in your page.

> **Security Note** The URLs associated with a hyperlink can be tampered with by a malicious user. For more information, see Scripting Exploits.

Another advantage of using the **HyperLink** control is that you can use data binding to specify the target URL for the link (and parameters to be passed with the link, if necessary). A typical example is to create **HyperLink** controls based on a list of products; the target URL points to a page where the user can read more detail about the product.

> **Tip** If you need to create data-bound **HyperLink** controls, a convenient way to do so is to add them as children of a Repeater, DataList, or DataGrid control.

The **HyperLink** control can display clickable text or an image.

Unlike most Web server controls, the **HyperLink** control does not raise an event in server code when users click it. Instead, the control simply performs navigation.

See Also

Adding HyperLink Web Server Controls to a Web Forms Page | Adding Controls to a Web Forms Page Programmatically | Web Forms Data Binding

Adding HyperLink Web Server Controls to a Web Forms Page

HyperLink controls can render as text or as graphics.

To add a HyperLink control to a Web Forms page

1. Switch to **Design** view in the Web Forms Designer, and then drag a **HyperLink** control from the **Web Forms** tab of the Toolbox onto the page.
2. Specify the format of the link in the page by doing one of the following:
   - To create a text link, set the control's Text property. You can include HTML formatting in the property; for example, you can bold an individual word in the text by placing a <B> element around it in the **Text** property.
   - To create a graphical link, set the control's ImageUrl property to the URL of a .gif, .jpg, or other Web graphic file. For details about adding graphics to your project, see Adding and Removing Files in the File System Editor.

     Note If you set both the **ImageUrl** and **Text** properties, the **ImageUrl** property takes precedence.
3. Set the NavigateUrl property to the URL of the page to link to.

   Security Note The URLs associated with a hyperlink can be tampered with by a malicious user. For more information, see Scripting Exploits.
4. Optionally, set the ID of a target window or frame to display the linked page in. You can either specify a window by name or you can use predefined target values such as `_top`, `_parent`, and so on.

   Tip You can change the appearance of the link text — for example, whether it is underlined — using styles. For details, see Customizing the Appearance of ASP.NET Server Controls Using Styles.

   The following example shows how you can set the properties of a **Hyperlink** control at run time. The method handles a **Button** control's **Click** event and sets the control's link text and target page.

```
' Visual Basic
Public Sub Button1_Click(ByVal sender As Object, ByVal e As System.EventArgs)
   Hyperlink1.Text = "Home"
   Hyperlink1.NavigateUrl = "http://www.microsoft.com/net/"
End Sub

// C#
public void Button1_Click (object sender, System.EventArgs e)
{
   this.HyperLink1.Text = "Home";
   this.HyperLink1.NavigateUrl = "http://www.microsoft.com/net/";
}
```

See Also

Introduction to the HyperLink Web Server Control | Web Forms Data Binding | HyperLink Class

Image Web Server Control

The **Image** Web server control allows you to display images on a Web Forms page and manage the images in your own code. The following topics provide information that you will find useful in learning to work with the **Image** control.

In This Section

Introduction to the Image Web Server Control
Provides background information on the **Image** control.
Adding Image Web Server Controls to a Web Forms Page
Gives directions for placing an **Image** control on a page and setting its appearance.

Related Sections

Setting Web Server Control Properties Programmatically
Gives directions for specifying Web server control properties in code.
Customizing the Appearance of ASP.NET Server Controls Using Styles
Gives directions for applying styles to change how controls are displayed.
Web Forms Data Binding
Provides basic conceptual information about binding data to Web Forms.
Data Binding Single-Value Web Server Controls at Design Time
Gives directions for using data binding to specify the image for an **Image** control.

Introduction to the Image Web Server Control

The Image Web server control allows you to display images on a Web Forms page and manage the images in your own code.

You can specify the graphics file for an **Image** control at design time in the designer or at run time in a program. You can also bind the control's ImageURL property to a data source to display graphics based on database information. For details, see Data Binding Single-Value Web Server Controls at Design Time.

Unlike most other Web server controls, the **Image** control does not support any events; for example, it does not respond to mouse clicks. You can create an interactive image using the ImageButton Web server control.

See Also

Adding Label Web Server Controls to a Web Forms Page | Setting Web Server Control Properties Programmatically | Customizing the Appearance of ASP.NET Server Controls Using Styles

Adding Image Web Server Controls to a Web Forms Page

The minimum action required to display an image is to create an **Image** control and point to a graphics file.

1. In **Design** view, drag an **Image** control from the **Web Forms** tab of the Toolbox onto the page.
2. Set the control's ImageURL property to the URL of a .gif, .jpg, or other Web graphic file. For details about adding graphics to your project, see Adding and Removing Files in the File System Editor.

   Security Note The URL associated with an **Image** control can be tampered with by a malicious user. For more information, see Scripting Exploits.

   Tip You can choose an image using the **Select Image** dialog box. To display it, click the ellipsis button (...) next to the **ImageUrl** property in the Properties window. For help with choosing an image, click **Help** in the dialog box.

   Note The control does not check that the URL you set is valid.

   For information about setting the **ImageUrl** property using data binding, see Data Binding Single-Value Web Server Controls at Design Time.

3. Optionally, set the following **Image** control properties.

Property	Description
Height and **Width**	Reserve space for the graphic on the page and resize the graphic to fit the reserved space. The image will be sized to fit the space you reserve.
ImageAlign	Aligns the image with respect to surrounding text, using values such as **Top**, **Bottom**, **Left**, **Middle**, and **Right**. In code, image alignment is set using the ImageAlign enumeration.
AlternateText	Displays text in place of the graphic if the graphic can't be loaded. In some browsers, this text is also displayed as a ToolTip.

See Also

Setting Web Server Control Properties Programmatically | Customizing the Appearance of ASP.NET Server Controls Using Styles | Image Class

Label Web Server Control

The **Label** Web server control provides a way to display text under programmatic control in a Web Forms page.

> **Note** If you want to display static text, you can present it using HTML; you do not need a **Label** control. Use a **Label** control only if you need to change the contents or other characteristics of the text in server code. If you are in linear layout mode, you can simply type text in. If you are in grid layout mode, you can add a **Linear Layout Panel** element and add static text inside of it.

The following topics provide information that you will find useful in learning to work with the **Label** Web server control.

In this Section

Introduction to the Label Web Server Control
Provides background information on the **Label** control.
Adding Label Web Server Controls to a Web Forms Page
Gives directions for placing a **Label** control on a page.

Related Sections

Setting Web Server Control Properties Programmatically
Gives directions for specifying Web server control properties in code.
Customizing the Appearance of ASP.NET Server Controls Using Styles
Gives directions for applying styles to change how controls are displayed.
Web Forms Data Binding
Provides basic conceptual information about binding data to Web Forms.
Data Binding Single-Value Web Server Controls at Design Time
Gives directions for displaying data in a **Label** control.

Introduction to the Label Web Server Control

The Label Web server control provides you with a way to programmatically set text in a Web Forms page. You typically use the **Label** control when you want to change text in the page at run time, such as in response to a button click. A **Label** control is better for simply displaying text than a **TextBox** control (or another control) because the resulting text is static on the page; users cannot edit it.

> **Note** If you want to display static text, you can present it using HTML; you do not need a **Label** control. Use a **Label** control only if you need to change the contents or other characteristics of the text in server code. If you are in linear layout mode, you can simply type text in. If you are in grid layout mode, you can add a **Linear Layout Panel** element and add static text inside of it.

You can set the text of the **Label** control at design time in the designer or at run time in a program. You can also bind the Text property of a **Label** control to a data source to display database information on a page.

For details about data binding, see Data Binding Single-Value Web Server Controls at Design Time.

Using Labels in List Controls

The **Label** control is often used in list Web server controls (**Repeater**, **DataList**, and **DataGrid** controls) to display read-only information from a database. The usual strategy is to add a **Label** control to a template for one of these controls. You then bind the **Label** control to a data source (generally via the container list control). For details about templates, see Web Server Controls Templates.

See Also

Adding Label Web Server Controls to a Web Forms Page | Adding Web Server Controls to a Web Forms Page | Setting Web Server Control Properties Programmatically | Customizing the Appearance of ASP.NET Server Controls Using Styles

Adding Label Web Server Controls to a Web Forms Page

You can add **Label** Web server controls as self-standing controls on the page. You can also make them child controls of other controls, such as using them in Web server control templates.

> **Note** If you want to display static text, you can present it using HTML; you do not need a **Label** control. Use a **Label** control only if you need to change the contents or other characteristics of the text in server code. If you are in linear layout mode, you can simply type text in. If you are in grid layout mode, you can add a **Linear Layout Panel** element and add static text inside of it.

To add a Label Web server control to a Web Forms page

1. In **Design** view, drag a **Label** control from the **Web Forms** tab of the Toolbox onto the page.
2. Set the control's Text property to the text to display. You can include HTML formatting in the property; for example, you can bold an individual word in the text by placing a `<B>` element around it in the **Text** property.

   The following example shows how you can set the text of a **Label** control at run time. The method handles a **Button** control's **Click** event and displays in the **Label** control whatever the user has typed into a **TextBox** control called **TextBox1**.

   > **Security Note** User input in a Web Forms page can include potentially malicious client script. By default, the Web Forms page validates that user input does not include script or HTML elements. For more information, see Scripting Exploits and Protecting Against Script Exploits in a Web Application.

```
' Visual Basic
Private Sub Button1_Click(ByVal sender as Object, ByVal e as EventArgs)
   Label1.Text = TextBox1.Text
End Sub

// C#
private void Button1_Click(object sender, System.EventArgs e) {
   Label1.Text = TextBox1.Text;
}
```

See Also

Introduction to the Label Web Server Control | Data Binding Single-Value Web Server Controls at Design Time | Setting Web Server Control Properties Programmatically | Customizing the Appearance of ASP.NET Server Controls Using Styles | Label Class

ListBox Web Server Control

The **ListBox** Web server control allows users to select one or more items from a predefined list.

The following topics provide information that you will find useful in learning to work with the **ListBox** control.

In This Section

Introduction to the ListBox Web Server Control
Explains the general use of the **ListBox** control.
Adding ListBox Web Server Controls to a Web Forms Page
Gives directions for putting a **ListBox** control onto a Web Form.
Adding Items in a List Web Server Control
Gives directions for defining individual items in the **ListBox** control, either at design time or programmatically at run time.
Determining the Selection in a List Web Server Control
Gives directions for checking in code what the user has selected.
Populating a List Web Server Control from a Database
Gives directions for using a **ListBox** control to display database information.
Responding to Changes in a List Web Server Control
Creating event handlers that are called when users work with a control.
Setting the Selection in a List Web Server Control
Gives directions for preselecting items in the control.

Related Sections

Adding Web Server Controls to a Web Forms Page
Gives directions for including Web server controls on your Web Form.
Setting ASP.NET Server Control Properties
Suggests additional topics of interest concerning server control properties.
Setting Web Server Control Properties Programmatically
Gives directions for setting Web server control properties in code.
Customizing the Appearance of ASP.NET Server Controls Using Styles
Gives directions for changing the look of Web server controls on a page.

Introduction to the ListBox Web Server Control

The ListBox Web server control allows users to select one or more items from a predefined list. It differs from a DropDownList control in that it can display multiple items at once and it optionally allows the user to select multiple items.

ListBox Control Appearance

The **ListBox** control is typically used to display more than one item at once. You can control the look of the list in these ways:

- Number of rows displayed You can set the control to display a specific number of items. If the control contains more items than that, it displays a vertical scrollbar.
- Height and width You can set the size of the control using pixels. In that case, the control ignores the number of rows you've set and displays as many as will fit into the height of the control. Some browsers do not support setting the height and width in pixels and will use the row count setting.

As with other Web server controls, you can use style objects to specify the appearance of the control. For details, see Customizing the Appearance of ASP.NET Server Controls Using Styles.

Items

The **ListBox** control is actually a container for the list items. Each list item is a separate object with its own properties:

Property	Description
Text	The text displayed in the list.
Value	The value associated with an item. Setting this property allows you to associate a value with a specific item without displaying it. For example, you can set the **Text** property to the name of an employee and the **Value** property to the employee's email alias.
Selected	A Boolean value indicating whether the item is selected. If the **ListBox** is set to allow multiple selections, more than one item can be selected.

To work with items programmatically, you usually access the Items of the **ListBox** control. **Items** is a standard collection, and you can add item objects to it, delete items, clear the collection, and so on. To test which items are selected, you can loop through the collection and test individual items.

For ease of use, the **ListBox** control also supports a property called SelectedItem. If the control is set to single-selection mode, this property returns the one selected item, making it unnecessary to loop through the entire **Items** collection to get at the current selection.

Single versus Multiple Selection

Users can normally select a single item in the list by clicking it. If you set the **ListBox** control to allow multiple selections, users can hold down the CTRL or SHIFT key while clicking to select multiple items.

Binding Data to the Control

You can use a **ListBox** Web server control to list options that are read from a data source. Each item in the **ListBox** control corresponds to an item — typically a row — in the data source.

The control displays one field from the source. Optionally, you can bind the control to a second field to set the value (which does not display) of an item.

As with other Web server controls, you can bind any control properties, such as the color or size of the control, to data. For details, see Web Forms Data Binding.

ListBox Events

The **ListBox** control raises an event — SelectedIndexChanged — when users select an item. By default, this event does not cause the page to be posted to the server, but you can cause the control to force an immediate post by setting the AutoPostBack property to **true**. If the **ListBox** control has been has been set to allow multiple selection and **AutoPostBack** is **true**, the form is posted back to the server with each selection. However, the current selection is maintained during each round trip.

Note The ability of a **ListBox** control to post to the server when it is checked requires that the browser support ECMAScript (JScript, JavaScript) and that scripting be enabled on the user's browser.

See Also

Adding ListBox Web Server Controls to a Web Forms Page | Adding Items in a List Web Server Control | Populating a List Web Server Control from a Database | Determining the Selection in a List Web Server Control | Setting the Selection in a List Web Server Control | Responding to Changes in a List Web Server Control

Adding ListBox Web Server Controls to a Web Forms Page

The process of adding a **ListBox** Web server control consists primarily of setting options for how items should be displayed and whether users can select multiple items.

To add a ListBox control to a Web Forms page

- In **Design** view, drag the **ListBox** control from the Toolbox onto the page, and then set the following properties:

Property	Description
Rows	Set to specify how many items should be displayed at once. **Note** You can also set the Height property to a pixel value. If you do, it takes precedence over t he value of the **Rows** property. However, not all browsers support the **Height** property, so you s hould set the **Rows** property also.
SelectionMode	Set to **Single** or **Multiple** to specify how many items users can select.

Adding items to the control is a separate process that depends on whether you will display a static list in the control or a list generated dynamically at run time. For details, see Adding Items in a List Web Server Control.

See Also

Adding Web Server Controls to a Web Forms Page | ListBox Class

Adding Items in a List Web Server Control

The information in this topic applies to all list Web server controls: ListBox, DropDownList, CheckBoxList, and RadioButtonList.

You can add items to a list Web server control in these ways:

- Add static items at design time.
- Add items programmatically at run time.
- Add items using data binding. For details, see Populating a List Web Server Control from a Database.

When you add a list item, you specify up to three possible properties of the item:

Property	Description
Text	The text displayed in the list.
Value	The value associated with an item. Setting this property allows you to associate a value with a specific item without displaying it. For example, you can set the **Text** property to the name of chemical element and the **Value** property to the symbol of the element.
Selected	A Boolean value indicating whether the item is selected. In a CheckBoxList control and in a multi-selection ListBox control, more than one item can be selected. In a DropDownList control, RadioButtonList control, and single-selection ListBox control, only one item can be selected at a time. If you set more than one selected item in these controls, the browser determines which item is rendered as selected.

To add static items at design time

1. In **Design** view, select the list control. In the Properties window, click the ellipsis button (...) for the Items property to open the **Collection Editor** dialog box.
2. Click **Add** to add a new item.
3. To set the properties for the new item, select it, and then in the **ListItem Properties** list, enter values for its **Text**, **Value**, and **Selected** properties.
4. Repeat Steps 2 and 3 for each item to add, and then choose **OK** to close the **Collection Editor** dialog box.

To add items programmatically

1. Create a new object of type **ListItem** and set its **Text** and **Value** properties.
2. Call the **Add** method of the control's **Items** collection and pass it the new object.

   The following example shows how to add **ListItem** objects to a **ListBox** control, but the procedure is identical for all list Web server controls.

```
' Visual Basic
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Han
dles Button1.Click
   Me.ListBox1.Items.Add(New ListItem("Carbon", "C"))
   Me.ListBox1.Items.Add(New ListItem("Oxygen", "O"))
End Sub

// C#
public void Button1_Click (object sender, System.EventArgs e)
{
   this.ListBox1.Items.Add(new ListItem("Carbon", "C"));
   this.ListBox1.Items.Add(new ListItem("Oxygen", "O"));
}
```

The following example shows how you can create new items programmatically. Using a loop, the code creates three new items in a **ListBox** control.

```
' Visual Basic
Private Sub Button2_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Han
dles Button2.Click
   Dim elements() As String = New String() {"Carbon", "Oxygen", "Hydrogen"}
   Dim i As Integer
   For i = 0 To ubound(elements)
      Me.ListBox1.Items.Add(New ListItem(elements(i)))
   Next
End Sub

// C#
public void Button1_Click (object sender, System.EventArgs e)
{
   string[] elements = {"Carbon", "Oxygen", "Hydrogen"};
   int i;

   for (i = 0;i < elements.GetLength(0);i++)
   {
      this.ListBox1.Items.Add(new ListItem(elements[i]));
   }
}
```

See Also

Populating a List Web Server Control from a Database | Determining the Selection in a List Web Server Control | Setting the Selection in a List Web Server Control | ListBox Class

Determining the Selection in a List Web Server Control

The information in this topic applies to all list Web server controls: ListBox, DropDownList, CheckBoxList, and RadioButtonList.

One of the most common tasks in working with a list Web server control is to determine what item or items users have selected. The procedure varies depending on whether the list control allows single or multiple selections.

Use the following procedure when working with the **DropDownList** control, the **RadioButtonList** control, and a single-selection **ListBox** control.

> **Note** When you data bind a list control (by calling its **DataBind** method), the current list selection is lost. You generally do not want to call the **DataBind** method on each round trip (that is, in the page initialization without checking for a post-back), because doing so replaces the values in controls. For details, see Data Binding Multi-Record Web Server Controls.

To determine the selection in a single-selection list control

- Use one of the following methods:
  - To get the index value of the selected item, read the value of the SelectedIndex property. The index is zero-based. If nothing has been selected, the value of the property is -1.
  - To get the contents of the selected item, get the control's SelectedItem property. This property returns an object of type **ListItem**. You can get the contents of the selected item by getting the **Text** or **Value** property of the object.

    > **Security Note** Controls in a Web Forms page can include potentially malicious client script. By default, the Web Forms page validates that user input does not include script or HTML elements. For more information, see Scripting Exploits and Protecting Against Script Exploits in a Web Application.

  The following example shows how you can test which item is selected in a **RadioButtonList** control. The code first checks that there is a selection at all by reading the value of the **SelectedIndex** property, which is set to -1 until the user selects an item. It then gets the **SelectedItem** object and displays that object's **Text** property.

```
' Visual Basic
    Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs)
Handles Button1.Click

   ' Is anything selected? The index is -1 if nothing is selected.
   If RadioButtonList1.SelectedIndex > -1 Then
      Label1.Text="You chose: " & RadioButtonList1.SelectedItem.Text
   End If
End Sub

// C#
public void Button1_Click (object sender, System.EventArgs e)
{
   // Is anything selected? The index is -1 if nothing is selected.
   if (RadioButtonList1.SelectedIndex > -1) {
      // The contents of the selection is available in the
      // SelectedItem objects, which provides the Text, Value,
      // and Selected properties.
      Label1.Text="You chose: " + RadioButtonList1.SelectedItem.Text;
   }
}
```

If the list control supports multiple selections, you must loop through the control and check for selected items one by one.

To determine the selection in a multi-selection list control

- Loop through the control's Items collection and test the **Selected** property of every individual item.

  The following example shows how you can test selections in a multi-selection **ListBox** control called ListBox1. The code displays a list of selected items in a label.

```
' Visual Basic
Public Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Hand
les Button1.Click
   Dim msg As String
   Dim li As ListItem
   msg = ""
   For Each li In ListBox1.Items
      If li.Selected = True Then
         msg = msg & "<BR>" & li.Text & " selected."
   End If
   Next
   Label1.Text = msg
End Sub

// C#
public void Button1_Click(object sender, System.EventArgs e)
{
   // Text that will display selected items
   string msg = "" ;
   foreach(ListItem li in ListBox1.Items)
   {
      if(li.Selected == true)
         {
            msg += "<BR>" + li.Text + " is selected.";
         }
   }
   Label1.Text = msg;
}
```

See Also

Setting the Selection in a List Web Server Control | ListBox Class

Populating a List Web Server Control from a Database

The information in this topic applies to all list Web server controls: ListBox, DropDownList, CheckBoxList, and RadioButtonList.

You can use a list Web server control to display items that are read from a data source. Each item in the control corresponds to an item — typically a row — in the data source. The control can display one field from the source and can optionally use a second field as the item value.

To populate a list control from a database

1. Set the control's DataSource property to a reference to a data source for the page.
2. Specify which fields in the data source should be used to populate each item by setting the following control properties:

   - DataTextField The name of the field whose value will be displayed in the list.
   - DataTextFormatString A formatting expression for the list item text. For information about creating a formatting expression, see Composite Formatting.
   - DataValueField The name of the field to be used for the **Value** property of each list item.

   If the data source has only one field, you do not have to explicitly set these fields, because the control will simply display the single field.

See Also

Introduction to Data Binding in Web Forms Pages | Multi-Record and Single-Value Data Binding for Web Forms Pages | ListBox Class

Responding to Changes in a List Web Server Control

The information in this topic applies to all list Web server controls: ListBox, DropDownList, CheckBoxList, and RadioButtonList.

When the user makes a selection in list Web server control, the control raises an event that you can respond to directly. Normally you do not need to respond directly to the selection event at all. Instead, it is more typical to test which item is selected after the form has been posted to the server by a control such as a Button control. For details about determining which item is selected, see Determining the Selection in a List Web Server Control.

To respond directly to changes in a list Web server control

- Create an event handler for the control's SelectedIndexChanged event.

  By default, the **SelectedIndexChanged** event does not immediately cause the Web Forms page to be posted to the server. Instead, the event is raised in server code the next time the form is posted (as when a **Button** Web server control is clicked). To have the **SelectedIndexChanged** event cause an immediate posting, set the control's AutoPostBack property to **true**.

  > **Note** The event is raised as soon as the user makes a selection in the list control. If the **AutoPostBack** property is **true**, the form is posted with each selection, but the selected items are preserved with each round trip.

  The following example shows how you can respond to a selection in a **DropDownList** control. The event handler displays the user's selection in a **Label** control.

  > **Security Note** Controls in a Web Forms page can include potentially malicious client script. By default, the Web Forms page validates that user input does not include script or HTML elements. For more information, see Scripting Exploits and Protecting Against Script Exploits in a Web Application.

```
' Visual Basic
Private Sub DropDownList1_SelectedIndexChanged(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles DropDownList1.SelectedIndexChanged
   Label1.Text = "You selected " & DropDownList1.SelectedItem.Text
End Sub

// C#
private void DropDownList1_SelectedIndexChanged(object sender, System.EventArgs e)
{
   Label1.Text = "You selected " + DropDownList1.SelectedItem.Text;
}
```

See Also

ASP.NET Server Control Event Model | Creating Event Handlers in Web Forms Pages | ListBox Class

Setting the Selection in a List Web Server Control

The information in this topic applies to all list Web server controls: ListBox, DropDownList, CheckBoxList, and RadioButtonList.

Normally, users select items in a list Web server control to indicate their choice. However, you might want to preselect items or select items at run time (programmatically) based on some condition.

To set the selection in a list Web server control at design time

1. In **Design** view, select the control. In the Properties window, click the ellipsis button (...) for the Items property to open the **Collection Editor** dialog box.
2. From the **Members** list, choose the member to be selected, and then set its **Selected** property to true.
3. If the control is set to allow multiple selection, repeat Step 2 for each item to select, and then choose **OK** to close the **Collection Editor** dialog box.

To set a single selection in a list Web server control programmatically

- Set the control's SelectedIndex property to the index value of the item to select. The index is zero-based. To set no selection, set **SelectedIndex** to -1.

```
' Visual Basic
ListBox1.SelectedIndex = 2
' Selects the third item

// C#
ListBox1.SelectedIndex = 2;
// Selects the third item
```

To set multiple selections in a list control programmatically

- Loop through the control's **Items** collection and set the **Selected** property of every individual item.

  Note You can only select multiple items if the control's SelectionMode property is set to **Multiple**.

  The following example shows how you can set selections in a multi-selection **ListBox** control called ListBox1. The code selects every other item.

```
' Visual Basic
Protected Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) H
andles Button1.Click
   Dim i As Integer
   Dim li As ListItem
   For Each li In ListBox1.Items
      i += 1
      If (i Mod 2 = 0) Then
         li.Selected = True
      End If
   Next
End Sub

// C#
public void Button1_Click(object sender, System.EventArgs e)
{
   // Counter
   int i = 0;
   foreach(ListItem li in ListBox1.Items)
   {
      if( (i%2) == 0)
      {
         li.Selected = true;
```

```
        }
        i += 1;
    }
}
```

See Also

Setting ASP.NET Server Control Properties | Setting Web Server Control Properties Programmatically | ListBox Class

Literal Web Server Control

The **Literal** control renders static text into a Web page without adding any HTML elements. You can manipulate the text programmatically with server code.

> **Note** If you want to display static text, you can present it using HTML; you do not need a **Literal** control. Use a **Literal** control only if you need to change the contents in server code.

The following topics provide information that you will find useful in learning to work with the **Literal** Web server control.

In This Section

Introduction to the Literal Web Server Control
Provides background information on the **Literal** control.
Adding Literal Web Server Controls to a Web Forms Page
Gives directions for placing a **Literal** control on a page.

Related Sections

Setting Web Server Control Properties Programmatically
Gives directions for specifying Web server control properties in code.
Customizing the Appearance of ASP.NET Server Controls Using Styles
Gives directions for applying styles to change how controls are displayed.
Web Forms Data Binding
Provides basic conceptual information about binding data to Web Forms.

Introduction to the Literal Web Server Control

You can use a Literal Web server control to render static text into a Web page and manipulate that text using server code. Unlike the **Label** Web server control, the **Literal** control does not add any HTML elements to the text.

As an example, you might want to create a simple vertical list of text elements that can be programmatically manipulated. You can do this by setting the **Text** property either of a **Label** control or of a **Literal** control to the HTML content that you want to send to the Web page. If you use a **Label** control, the control is wrapped in an HTML tag. The does not affect the rendering of the page in any noticeable way. If you use a **Literal** control, the tag is not added, which allows your code to be simpler.

You cannot apply styles to the content of the **Literal** control. This means that the **Literal** control cannot be positioned when the Web Forms Designer is in grid mode; as a consequence, the **Literal** control might not be suitable for creating captions. In addition, you cannot address the control with client code.

See Also

Adding Literal Web Server Controls to a Web Forms Page | Literal Web Server Control | Literal Class

Adding Literal Web Server Controls to a Web Forms Page

Add a **Literal** Web server control to your page when you want to set text programmatically without adding extra HTML tags.

> **Note** If you want to display static text, you can present it using HTML; you do not need a **Literal** control. Use a **Literal** control only if you need to change the contents in server code.

To add a Literal control to a Web Forms page

1. In Design view, drag a **Literal** control from the **Web Forms** tab of the Toolbox onto the page.

   The following example shows a simple page that displays a headline at run time. The body of the page, including the **Literal** control, might look like this:

   ```
   <body>
      <form runat="server">
         <H1><asp:Literal id="Headline" runat=server /></H1>
      </form>
   </body>
   ```

2. Add code to your page to set the control's Text property at run time.

   The following example shows how to programmatically set the text of the **Literal** control.

   ```
   ' Visual Basic
   Sub Page_Load(ByVal sender As System.Object, _
         ByVal e As System.EventArgs) Handles MyBase.Load
      Headline.Text = "New Web Site Announced"
   End Sub

   // C#
   private void Page_Load(object sender, System.EventArgs e)
   {
      Headline.Text = "New Web Site Announced";
   }
   ```

See Also

Introduction to the Literal Web Server Control | Literal Web Server Control | Literal Class

Panel Web Server Control

The **Panel** Web server control provides a container within the page for other controls. By putting controls into a **Panel** control, you can control them as a unit — for example, hiding or showing them. You can also use a **Panel** control to create a distinct appearance for a group of controls.

> **Note** The **Panel** control is not required to group controls such as **RadioButton** controls.

The following topics provide information that you will find useful in learning to work with the **Panel** control.

In This Section

Introduction to the Panel Web Server Control
Provides background information on the **Panel** control.
Adding Panel Controls to a Web Forms Page
Gives directions for adding a **Panel** Web server control to a Web Forms page and adding child controls to it.

Related Sections

Adding Controls to a Web Forms Page Programmatically
Gives directions for adding child controls to a **Panel** control at run time.
CheckBox and CheckBoxList Web Server Controls
Gives information on creating grouped check boxes.
RadioButton and RadioButtonList Web Server Controls
Gives information on creating grouped radio buttons.

Introduction to the Panel Web Server Control

The Panel Web server control provides a container control within a Web Forms page that you can use as a parent for static text and for other controls. The **Panel** control is suitable for:

- **Group Behavior** You can manage a group of controls as a unit by putting them in a panel and then manipulating the panel. For example, you can hide or show a group of controls inside a panel by setting the panel's **Visible** property.
- **Dynamic Control Generation** The **Panel** control provides a convenient container for controls that you create at run time.
- **Appearance** The **Panel** control supports appearance properties such as **BackColor** and **BorderWidth** that you can set to create a unique look for local area on a page.

> **Note** The **Panel** control is not required to group controls such as **RadioButton** controls.

See Also

Adding Panel Controls to a Web Forms Page | Adding Controls to a Web Forms Page Programmatically

Adding Panel Controls to a Web Forms Page

To use a **Panel** Web server control, you add it to a page and then add controls and static text to it.

To add a Panel control to a Web Forms page

1. Switch to Design view in the Web Forms Designer, and then drag a **Panel** control from the **Web Forms** tab of the Toolbox onto the page.

   The control is initially displayed with the static text "Panel" in it. To work with static text in the control, see below.

2. Optionally, drag the panel's borders to resize the control.

   > **Note** The control automatically resizes itself to display all its child controls, even if they exceed the height you have set.

3. Optionally, set the following properties of the **Panel** control to specify how the pane interacts with its child controls.

Property	Description
HorizontalAlign	Specifies how the child controls are aligned within the panel (left, right, centered, or justified).
Wrap	Specifies whether content that is too wide for the panel is wrapped to the next line or truncated at the panel's edge.

You can add and remove text directly in **Design** view.

To work with static text in the Panel control at design time

1. Place the pointer where you want to edit in the **Panel** control, and then click the control once to select it. When you see selection handles on the border, click again to put the control into text-editing mode. (If you click too quickly the second time, the designer will treat your click as a double click and switch to the Code Editor).

   When the panel is in text-editing mode, it displays a shaded border and the mouse pointer becomes an insertion point.

2. Edit text within the **Panel** control as you do on the page.

   > **Note** To add static text to the **Panel** control at run time, you create a LiteralControl object and set its **Text** property. You then programmatically add the **LiteralControl** object to the panel the way you would any control. For details on adding controls, see Adding Controls to a Web Forms Page Programmatically.

After adding a **Panel** control to a page, you can add controls. You can do this at design time or at run time.

To add controls to a Panel control at design time

- To add controls, drag them from the Toolbox into the **Panel** control. If necessary, the panel will resize itself to accommodate the control.

  > **Tip** If you want to be able to position controls in the panel using x and y coordinates, first drag a Grid Layout Panel control from the HTML tab of the Toolbox into the panel and size it to fit. You can then drag controls onto the panel and position them with the mouse.

For details about adding controls at run time, see Adding Controls to a Web Forms Page Programmatically.

See Also

Introduction to the Panel Web Server Control | Panel Class

PlaceHolder Web Server Control

The **PlaceHolder** control enables you to place an empty container control in the page and then dynamically add child elements to it at run time.

The following topics provide information that you will find useful in learning to work with the **PlaceHolder** Web server control.

In This Section

Introduction to the PlaceHolder Web Server Control
Provides background information on the **PlaceHolder** control.
Adding PlaceHolder Web Server Controls to a Web Forms Page
Gives directions for placing a **PlaceHolder** control on a page.

Related Sections

Setting Web Server Control Properties Programmatically
Gives directions for specifying Web server control properties in code.
Customizing the Appearance of ASP.NET Server Controls Using Styles
Gives directions for applying styles to change how controls are displayed.
Web Forms Data Binding
Provides basic conceptual information about binding data to Web Forms.

Introduction to the PlaceHolder Web Server Control

The PlaceHolder Web server control enables you to place an empty container control within the page and then dynamically add, remove, or loop through child elements at run time. The control only renders its child elements; it has no HTML-based output of its own.

As an example, you might want to have a variable number of buttons appear on a Web page, depending on options selected by users. That way, users are not confronted with potentially confusing choices that are either unavailable or not relevant to their individual needs.

See Also

Adding PlaceHolder Web Server Controls to a Web Forms Page | PlaceHolder Class | PlaceHolder Web Server Control

Adding PlaceHolder Web Server Controls to a Web Forms Page

Add a **PlaceHolder** Web server control to your page when you want to dynamically add, remove, or loop through controls at run time.

To add a PlaceHolder control to a Web Forms page

- In Design view, drag a **PlaceHolder** control from the **Web Forms** tab of the Toolbox onto the page.

To add child controls to a PlaceHolder control at run time

1. Create an instance of the control you want to add to the **PlaceHolder** control.
2. Call the Add method of the **PlaceHolder** control's Controls property, passing it the instance you created.

   The following example shows how to add three **HtmlButton** controls as children of a **PlaceHolder** control.

```
' Visual Basic
Private Sub Page_Load(ByVal sender As System.Object, _
      ByVal e As System.EventArgs) Handles MyBase.Load
   Dim btnClick As HtmlButton = New HtmlButton()
   btnClick.InnerText = "Button 1"
   PlaceHolder1.Controls.Add(btnClick)
   btnClick = New HtmlButton()
   btnClick.InnerText = "Button 2"
   PlaceHolder1.Controls.Add(btnClick)
   btnClick = New HtmlButton()
   btnClick.InnerText = "Button 3"
   PlaceHolder1.Controls.Add(btnClick)
End Sub

// C#
private void Page_Load(object sender, System.EventArgs e)
{
   HtmlButton btnClick = new HtmlButton();
   btnClick.InnerText = "Button 1";
   PlaceHolder1.Controls.Add(btnClick);
   btnClick = new HtmlButton();
   btnClick.InnerText = "Button 2";
   PlaceHolder1.Controls.Add(btnClick);
   btnClick = new HtmlButton();
   btnClick.InnerText = "Button 3";
   PlaceHolder1.Controls.Add(btnClick);
}
```

See Also

Introduction to the PlaceHolder Web Server Control | PlaceHolder Web Server Control |
Adding Controls to a Web Forms Page Programmatically

RadioButton and RadioButtonList Web Server Controls

The **RadioButton** and **RadioButtonList** Web server controls allow users to select one item from a short, predefined list. The following topics provide information that you will find useful in learning to work with these two controls.

Because there are two different controls, and because they function somewhat differently, it's important to understand their different purposes. You can refer to the following table to determine which topics pertain specifically to the control you want to work with.

> **Note** Some of the topics listed below for the **RadioButtonList** control do not pertain solely to that control; instead, they are topics that apply to all list Web Server controls (ListBox, DropDownList, CheckBoxList, and **RadioButtonList**).

In This Section

Introduction to the RadioButton and RadioButtonList Web Server Controls
Explains what you can do with **RadioButton** and **RadioButtonList** controls and what the differences are between them.
Adding Individual RadioButton Controls to a Web Forms Page
Gives directions for adding individual **RadioButton** controls to a page and grouping them.
Adding RadioButtonList Controls to a Web Forms Page
Gives directions for adding a **RadioButtonList** control (rather than individual **RadioButton** controls) to a page.
Responding to a User Selection in a RadioButton Group
Gives directions for writing an event handler that responds directly when the user clicks an individual **RadioButton** control.
Setting and Getting the Selection in a RadioButton Web Server Control
Gives directions for programmatically selecting and clearing individual **RadioButton** controls.
Setting Layout in a RadioButtonList Web Server Control
Gives directions for specifying how many columns of radio buttons the **RadioButtonList** control displays and how they are ordered.

Related Sections

Adding Web Server Controls to a Web Forms Page
Gives directions for including Web server controls on your Web Form.
Adding Items in a List Web Server Control
Gives directions for defining individual radio buttons in the **RadioButtonList** control, either at design time or programmatically at run time.
Populating a List Web Server Control from a Database
Gives directions for binding a **RadioButtonList** control to a database in order to display bound radio buttons.
Setting the Selection in a List Web Server Control
Gives directions for preselecting items in a **RadioButtonList** control.
Determining the Selection in a List Web Server Control
Gives directions for checking in code which radio button the user has clicked in a **RadioButtonList** selected.
Responding to Changes in a List Web Server Control
Gives directions for writing an event handler to respond directly when the user clicks an item in a **RadioButtonList** control.
Setting ASP.NET Server Control Properties
Suggests additional topics of interest concerning server control properties.
Setting Web Server Control Properties Programmatically
Gives directions for setting Web server control properties in code.
Customizing the Appearance of ASP.NET Server Controls Using Styles
Gives directions for changing the look of Web server controls on a page.

Introduction to the RadioButton and RadioButtonList Web Server Controls

You can use two types of Web server controls to add radio buttons to a Web Forms page: individual RadioButton controls or a RadioButtonList control. Both controls allow users to select from a small set of mutually exclusive, pre-defined choices. The controls allow you to define any number of radio buttons with labels and to arrange them horizontally or vertically.

> **Note** You can also use the HtmlInputRadioButton server control to add radio buttons to a Web Forms page. For details about the differences between Web and HTML controls, see Introduction to ASP.NET Server Controls.

> **Tip** If you want to present users with a longer list of options or a list that might vary in length at run time, use a ListBox or DropDownList Web server control.

RadioButton Control versus RadioButtonList Control

You add individual **RadioButton** controls to a page and work with them singly. Typically, you group two or more individual buttons together. See Adding Individual RadioButton Controls to a Web Forms Page.

In contrast, the **RadioButtonList** control is a single control that acts as a parent control for a collection of radio button list items. It derives from a base ListControl Class, and therefore works much like the **ListBox**, **DropDownList**, and CheckBoxList Web server controls. Therefore, many of the procedures for working with the **RadioButtonList** control are the same as they are for the other list Web server controls.

Each type of control has advantages. Individual **RadioButton** controls give you more control over the layout of the radio button group. For example, you can include text (that is, non-radio-button text) between each radio button.

The **RadioButtonList** control does not permit you to insert text between the buttons, but is far easier to use if you want to bind the buttons to a data source. It is also slightly easier to write code that checks which button has been selected.

Grouping Radio Buttons

Radio buttons are rarely used singly. Instead, they are grouped to provide a set of mutually exclusive options. Within a group, only one radio button can be selected at a time. You can create grouped radio buttons in these ways:

- Add individual RadioButton Web server controls to a page and then manually assign them all to a group. In this case, the group is an arbitrary name; all radio buttons with the same group name are considered part of a single group.
- Add a RadioButtonList Web server control to the page. The list items in the control are automatically grouped.

RadioButton Events

Events work slightly differently between individual **RadioButton** controls and the **RadioButtonList** control.

Individual RadioButton Controls

Individual **RadioButton** controls raise an event — CheckedChanged — when users click the control. By default, this event does not cause the page to be posted to the server, but you can have the control force an immediate post by setting the AutoPostBack property to **true**. For details about responding to this event directly, see Responding to a User Selection in a RadioButton Group.

> **Note** The ability of a **RadioButton** control to post to the server when it is checked requires that the browser support ECMAScript (Jscript, JavaScript) and that scripting be enabled on the user's browser.

Whether a **RadioButton** control posts to the server or not, it is not usually necessary to create an event handler for the **CheckedChanged** event. Instead, it is more common to test which button is selected when the form has been posted to the server by a control such as a **Button** control. For details, see
Setting and Getting the Selection in a RadioButton Web Server Control.

RadioButtonList Control

In contrast, the **RadioButtonList** control raises a SelectedIndexChanged event when users change which radio button in the list is selected. By default, the event does not cause the form to be posted to the server, although you can specify this option by setting the AutoPostBack property to **true**. For details, see Responding to Changes in a List Web Server Control.

> **Note** The ability of a **RadioButtonList** control to post to the server when it is checked requires that the browser support ECMAScript (JScript, JavaScript) and that scripting be enabled on the user's browser.

As with individual **RadioButton** controls, it is more common to test the state of the **RadioButtonList** control after the form has been posted some other way. For details, see Determining the Selection in a List Web Server Control.

Binding Data to the Control

As with any Web server control, you can bind an individual **RadioButton** control to a data source, and you can bind any property of the **RadioButton** control to any field of the data source. For example, you might set the control's Text property from information in a database.

However, because radio buttons are used in groups, binding a single radio button to a data source isn't a common scenario. Instead, it's more typical to bind a **RadioButtonList** control to a data source. In that case, the data source dynamically generates radio buttons (list items) for each record in the data source.

For general information about data binding Web server controls, see
Data Binding Single-Value Web Server Controls at Design Time.

See Also

Adding Web Server Controls to a Web Forms Page | Introduction to Validating User Input in Web Forms |
Setting Web Server Control Properties Programmatically | Customizing the Appearance of ASP.NET Server Controls Using Styles |
Web Forms State Management | Specifying Whether Controls Post to the Server

Adding Individual RadioButton Controls to a Web Forms Page

You can add radio buttons to a Web Forms page in two ways:

- By adding individual **RadioButton** Web server controls and grouping them.
- By adding a **RadioButtonList** control and adding individual list items to the control.

For details about the differences in these approaches, see
Introduction to the RadioButton and RadioButtonList Web Server Controls.

> **Note** This topic addresses how to add individual **RadioButton** controls to the page. For details about adding a **RadioButtonList** control, see Adding RadioButtonList Controls to a Web Forms Page.

When you use individual the **RadioButton** Web server controls, you typically add a set of them to the page and then group them. You can create multiple separate groups of buttons.

To add individual RadioButton controls to a Web Forms page

1. In Design view, drag a **RadioButton** control from the **Web Forms** tab of the Toolbox onto the page.
2. Specify a caption by setting the Text property.
3. Optionally, change the orientation of the caption by setting the TextAlign property.
4. Repeat Steps 1 through 3 for each radio button.

To group individual RadioButton controls

- Set the GroupName property of each control to the same name. You can use any string as the name, but do not include a space. For example, you might assign the string "RadioButtonGroup1" to the **GroupName** property of all the buttons.

  To create multiple groups, use a different group name for each.

  > **Note** You can add **RadioButton** controls to the page and group them with existing groups at any point.

See Also

Setting and Getting the Selection in a RadioButton Web Server Control | Responding to a User Selection in a RadioButton Group | RadioButton Class | RadioButtonList Class

Adding RadioButtonList Controls to a Web Forms Page

When you use a **RadioButtonList** control, you add a single control to the page. Adding items to the control is a separate process that depends on whether you will display a static list in the control or a list generated dynamically at run time. For details, see Adding Items in a List Web Server Control.

> **Note** The procedure for working with individual **RadioButton** Web server controls is different. For details, see Adding Individual RadioButton Controls to a Web Forms Page.

To add a RadioButtonList control to a Web Forms page

1. In Design view, drag a **RadioButtonList** control from the **Web Forms** tab of the Toolbox onto the page.
2. Optionally, change the orientation of the caption by setting the TextAlign property.
3. Optionally, change the layout of the control to display multiple columns. For details, see Setting Layout in a RadioButtonList Web Server Control.

See Also

Introduction to the RadioButton and RadioButtonList Web Server Controls | Adding Individual RadioButton Controls to a Web Forms Page | Setting Layout in a RadioButtonList Web Server Control | Adding Items in a List Web Server Control | Setting the Selection in a List Web Server Control | Determining the Selection in a List Web Server Control | RadioButton Class | RadioButtonList Class

Responding to a User Selection in a RadioButton Group

When the user selects a **RadioButton** control, the control raises an event that you can respond to.

> **Note** The **RadioButtonList** control raises events differently than individual **RadioButton** controls. For details, see Responding to Changes in a List Web Server Control.

You might not need to respond directly to the selection event of a **RadioButton** control at all. You respond to the event only if it's important to know when the user has made a change to the selection in a radio button group.

If you are only interested in knowing which radio button is selected, and not whether selection has changed, you can simply test radio button selection after the form has been posted to the server by a control such as a Button control. For details about determining which **RadioButton** control is selected, see Setting and Getting the Selection in a RadioButton Web Server Control.

Because each **RadioButton** Web server control is a separate control, each can raise an event individually; the radio button group does not raise an event as a whole.

To respond to a selection in a RadioButton control

- Create an event handler for the control's CheckedChanged event.

  By default, the **CheckedChanged** event does not immediately cause the Web Forms page to be posted to the server. Instead, the event is raised in server code the next time the form is posted, as when a **Button** Web server control is clicked. To have the **CheckedChanged** event cause an immediate posting, set the **RadioButton** control's AutoPostBack property to **true**.

  > **Note** The ability of a **RadioButton** control to post to the server when it is checked requires that the browser support ECMAScript (JScript, JavaScript) and that scripting be enabled on the user's browser.

  The following example shows how you can respond when a user selects a **RadioButton** control.

```
' Visual Basic
Sub RadioButton1_CheckedChanged(sender As Object, e As EventArgs) Handles RadioButton1.Ch
eckedChanged
   Label1.Text = "You selected Radio Button " & RadioButton1.Text
End Sub

// C#
public void RadioButton1_CheckedChanged (object sender, System.EventArgs e)
{
   Label1.Text = "You selected Radio Button " + RadioButton1.Text;
}
```

See Also

ASP.NET Server Control Event Model | Specifying Whether Controls Post to the Server |
Adding Individual RadioButton Controls to a Web Forms Page |
Setting and Getting the Selection in a RadioButton Web Server Control | Responding to a User Selection in a RadioButton Group |
RadioButton Class | RadioButtonList Class

Setting and Getting the Selection in a RadioButton Web Server Control

You can set a selected radio button at design time or at run time in code. If the radio button is in a group, setting it clears any other selection in the group.

> **Note** If you are working with a **RadioButtonList** control, the procedure for getting and setting the value of a button is different. For details, see Determining the Selection in a List Web Server Control and Setting the Selection in a List Web Server Control.

To set the selected RadioButton control

- Set the control's Checked property to **true**. If you select more than one **RadioButton** control in a group, the browser determines which button is rendered as selected.

  If you set the property to **false**, it clears the selection but does not select another radio button. Therefore, you can clear all selections by setting the **Checked** property of all radio buttons in a group to **false**.

Determining which **RadioButton** control has been selected is a matter of testing the **Checked** property.

To determine which RadioButton control in a group is selected

- Test the control's **Checked** property.

  > **Note** Testing the value of a radio button does not tell you if the user changed the value of the control, only if it is selected. To check for a change in the control, write an event handler for the control's CheckedChanged event. For details, see Responding to a User Selection in a RadioButton Group.

  To determine which is checked in a group, you have to test each control individually, as in the following example:

```
' Visual Basic
Public Sub Button1_Click(ByVal sender As Object, ByVal e As System.EventArgs) Handles But
ton1.Click
   Dim msg As String = "You selected "
   If RadioButton1.Checked = True Then
      msg = msg & RadioButton1.Text
   ElseIf Radiobutton2.Checked = True Then
      msg = msg & RadioButton2.Text
   ElseIf Radiobutton3.Checked = True Then
      msg = msg & RadioButton3.Text
   End If
   Label1.text = msg
End Sub

// C#
public void Button1_Click (object sender, System.EventArgs e)
{
   if (RadioButton1.Checked) {
      Label1.Text = "You selected " + RadioButton1.Text;
   }
   else if (RadioButton2.Checked) {
      Label1.Text = "You selected " + RadioButton2.Text;
   }
   else if (RadioButton3.Checked) {
      Label1.Text = "You selected " + RadioButton3.Text;
   }
}
```

See Also

Adding Individual RadioButton Controls to a Web Forms Page | Responding to a User Selection in a RadioButton Group | RadioButton Class | RadioButtonList Class

Setting Layout in a RadioButtonList Web Server Control

By default, the **RadioButtonList** Web server control displays a single column of buttons. However, you can specify any number of columns, and within the columns, you can specify how the items are ordered: vertically (the default) or horizontally. Vertical layout in three columns results in a layout such as the following:

```
A     D     G
B     E     H
C     F
```

Horizontal layout for the same items results in the following layout:

```
A     B     C
D     E     F
G     H
```

> **Note** If you are working with individual **RadioButton** Web server controls, you don't set layout as a property of the control. Instead, you set it by simply adding the radio buttons within the page flow. For details about the differences between these controls, see Introduction to the RadioButton and RadioButtonList Web Server Controls.

To specify column count and ordering:

1. In the Properties window, set the **RadioButtonList** control's RepeatColumns property to the number of columns you want.
2. Set the RepeatDirection property to **Vertical** or **Horizontal**. If you are setting the property in code, use the **RepeatDirection** enumeration, as shown in the following example:

```
' Visual Basic
Public Sub Button1_Click(ByVal sender As Object, ByVal e As System.EventArgs) Handles But
ton1.Click
   ' Create five radio buttons, labeled Red, Blue, Green, Yellow, and Orange.
   Dim colors() As String = _
      New String() {"Red", "Blue", "Green", "Yellow", "Orange"}
   RadioButtonList1.Items.Clear()
   Dim i As Integer
   For i = 0 To ubound(colors)
      RadioButtonList1.Items.Add(colors(i))
   Next
   ' Layout the radio buttons horizontally.
   RadioButtonList1.RepeatDirection = RepeatDirection.Horizontal
End Sub

// C#
public void Button1_Click (object sender, System.EventArgs e)
{
   // Create five radio buttons, labeled Red, Blue, Green, Yellow, and Orange.
   string[] colors = {"Red", "Blue", "Green", "Yellow", "Orange"};
   this.RadioButtonList1.Items.Clear();
   for(int i=0;i < colors.GetLength(0);i++){
      this.RadioButtonList1.Items.Add(colors[i]);
   }
   // Layout the radio buttons horizontally.
   this.RadioButtonList1.RepeatDirection = RepeatDirection.Horizontal;
}
```

See Also

Adding RadioButtonList Controls to a Web Forms Page | Setting the Selection in a List Web Server Control | Determining the Selection in a List Web Server Control | Adding Items in a List Web Server Control | RadioButton Class |

Repeater Web Server Control

The Repeater Web server control is a data-bound container control that produces a list of individual items. You define the layout of individual items on a Web page using templates. For an overview, see Web Server Controls Templates. When the page is run, the control repeats the layout for each item in the data source.

> **Note** For information on the differences between the **Repeater**, **DataList**, and **DataGrid** Web server controls, see Controls for Displaying Lists On Web Pages.

The following topics provide information that you will find useful in learning to work with the **Repeater** control.

In This Section

Introduction to the Repeater Web Server Control
Provides background information on the **Repeater** control, including an overview of its features.
Adding Repeater Controls to a Web Forms Page
Gives directions for adding a **Repeater** control to a Web Forms page and setting the required properties to make it work.

Related Sections

Responding to Button Events in DataList, Repeater, or DataGrid Items
Gives directions for adding buttons to individual items that invoke custom functionality for list items.
Introduction to ASP.NET Server Controls
Provides general information on working with server controls in Web Forms.
Programming Web Forms
Suggests additional topics on working with Web Forms in code.
Controls for Displaying Lists On Web Pages
Lists the Web server controls you can use to display lists of data.
Web Server Controls Templates
Provides background information on using templates with the **DataList** control.
ASP.NET Server Controls and CSS Styles
Provides information on specifying CSS styles in Web server controls.

Introduction to the Repeater Web Server Control

The **Repeater** Web server control is a basic container control that allows you to create custom lists out of any data available to the page. The control does not have an inherent look — for example, it does not by default create a vertical list of data or a table. Instead, you provide the layout for the control by creating templates. When the page runs, the **Repeater** control loops through the records in the data source and renders an "item" (or alternating item) for each record.

Because the **Repeater** control has no default look, it is useful for creating virtually any kind of list. Examples of how you can use the **Repeater** control to display information include:

- In a table layout
- In a comma-delimited list (for example, "a, b, c, d, ...")
- In a bulleted list or numbered list

Templates

To use the **Repeater** control, you create templates that define the layout of the control's content. Templates can contain any combination of HTML text and controls that are valid on a Web Forms page. If no templates are defined, or if none of the templates contain elements, the control does not appear on the page when the application is run.

The **Repeater** control supports the following templates:

Template	Description
ItemTemplate	Elements that are rendered once for each row in the data source. To display data in the **ItemTemplate**, you add one or more Web server or HTML server controls to the template and bind the controls to a data source.
AlternatingItemTemplate	Like the **ItemTemplate** element, but rendered for every other row in the **Repeater** control. You can specify a different look for the **AlternatingItemTemplate** element by setting its style properties.
HeaderTemplate and **FooterTemplate**	Elements to render once before and after all data-bound rows have been rendered. For example, you can have an HTML Table span the entire repeater by putting the opening tag, <TABLE>, in the **HeaderTemplate** and putting the closing tag, </TABLE> in the **FooterTemplate**.
SeparatorTemplate	Elements to render between each row, such as line breaks (tags), lines (<HR> tags), or commas.

For more information, see Web Server Controls Templates and Creating Web Server Control Templates.

Working with the Repeater Control at Design Time

The Web Forms Designer provides only limited what-you-see-is-what-you-get (WYSIWYG) editing for the **Repeater** control. You can drag the control onto a page and set its base properties as you do with any other control on the page.

However, to specify templates for the **Repeater** control, you work in the HTML view of the Web Forms Designer. The control allows you to specify incomplete HTML elements for templates — for example, a <TABLE> tag in the **HeaderItem** template and a </TABLE> element in the **FooterItem** template. Because the Design view of the Web Forms Designer always checks the validity of the HTML markup in a page, it will not accept element fragments of this sort. As a consequence, template editing for the **Repeater** control is done in HTML view.

> **Note** Warnings will be displayed in the task list when HTML tags are split across templates. You can safely ignore these warnings. The HTML elements will be displayed correctly when the application is run.

Binding Data to the Control

The **Repeater** control must be bound to a data source via its **DataSource** property or it will not render. The control can use as a data source any class that supports the IEnumerable interface. In practical terms, this means that you can use any type of data source that you can create using the Visual Studio data designers. You can also use data sources that you add to your page from the Toolbox, such as the **DataSet** or **DataView** class. You can also bind to simpler structures, such as an array.

Controls in the templates may be bound to the data source of the **Repeater** control or to a separate data source. Binding the controls to the **Repeater** control guarantees that all the controls will display data items from the same data row. The syntax for binding the controls to the **Repeater** control uses "Container" as the data source, since the **Repeater** is the container for all the controls. The binding command has the following form:

```
<%# DataBinder.Eval(Container, "DataItem.PhoneNumber") %>
```

Alternatively, a control may be bound to a data source that is not the same as the **Repeater** control's data source. This method might be used if you want the control to display a different row of data from the **Repeater** control's data source or if you want the control to have a completely different data source. For example, the **Repeater** item template might display the name and occupation of an employee from one table, while a **ListBox** control contains a selection of occupations from a different table.

> **Note** You cannot bind controls in the header, footer, and separator templates using **DataBinder.Eval** expressions in HTML view. You must bind these controls in the code.

When the page runs, you call the **Repeater** control's **DataBind** method to get the records to be displayed. You call the method again (for example, in an event handler) each time you need to refresh the data being displayed.

For an overview of data binding in Web server controls, see Data Access in Web Forms Pages and Adding Repeater Controls to a Web Forms Page.

Events

The **Repeater** control supports several events. The Repeater.ItemCreated event is called each time the table is rendered and allows you to customize the item-creation process. For example, if you were using a repeater control to display a to-do list, you could display overdue items in red text, completed items in black text, and other tasks in green text.

The Repeater.ItemCommand Event is raised in response to button clicks in individual items. The event is designed to allow you to embed a **Button**, **LinkButton**, or **ImageButton** Web server control in an item template and then be notified if a user clicks the button. When a click occurs, the event is bubbled up to the **Repeater** control. The most common uses for the **ItemCommand** event are to program add, update, and delete behaviors to the **Repeater** control. Since all the button clicks raise the same **ItemCommand** event, there must be way to distinguish which button has been clicked. You can determine which button was clicked by setting each button's **CommandName** property to a unique string value. The **ItemCommand** property of the **RepeaterCommandEventArgs** parameter contains the **CommandName** of the button clicked.

For more information, see Responding to Button Events in DataList, Repeater, or DataGrid Items.

See Also

Adding Repeater Controls to a Web Forms Page | ASP.NET Server Control Event Model | Repeater Class

Adding Repeater Controls to a Web Forms Page

Adding a **Repeater** Web server control to a page requires several steps. The following procedure describes the minimum that you must do to create a working **Repeater** control.

To add a Repeater Web server control to a Web Forms page

1. Add a data source to the Web Forms page. There are several methods for handling data on a page. To choose an appropriate method, see Introduction to Data Access with ADO.NET and Introduction to Data Access in Web Forms Pages.
2. In Design view, drag a **Repeater** control from the **Web Forms** tab of the Toolbox onto the page.
3. Set the control's **DataSource** property. In the Properties window, the drop-down list for the **DataSource** property will display all the data sources, such as **DataSet** and **DataView** controls, defined on the page.
4. Call the control's **DataBind** method. This is typically done in the page's **Page_Load** event using code such as the following, where the Repeater1 control is bound to a **DataSet** object. Since the **DataSet** was bound to the **Repeater** control in the designer, it does not need to be done in the code:

```
' Visual Basic
Protected Sub Page_Load(ByVal sender As System.Object, _
ByVal e As System.EventArgs) Handles MyBase.Load
   ' Put user code to initialize the page, including data, here.
   Repeater1.DataBind()
End Sub

// C#
protected void Page_Load(object sender, EventArgs e)
{
   // Put user code to initialize the page, including data, here.
   Repeater1.DataBind();
}
```

5. Switch to HTML view.
6. Create templates inside the **Repeater** element that contain HTML text and controls to display the data. For details see Web Server Controls Templates. The skeleton for an **ItemTemplate** element might look like this:

```
<asp:repeater id=Repeater1 runat="server" datasource="<%# dataView1 >">
   <ItemTemplate>
   </ItemTemplate>
</asp:repeater>
```

7. Add any combination of HTML elements and Web server controls to the templates to display the data source information.

   You must bind the controls to the data by including a data-binding expression. Typically, you bind them to the container control. An **ItemTemplate** element that displays the "EmployeeName" and "PhoneNumber" fields of each row in the data source would look like this:

```
<ItemTemplate>
   <%# DataBinder.Eval(Container, "DataItem.EmployeeName") %>
   <%# DataBinder.Eval(Container, "DataItem.PhoneNumber") %>
   <br>
</ItemTemplate>
```

   For details, see Data Access in Web Forms Pages.

The following example shows what a complete **Repeater** control definition might look like in HTML view. The **Repeater** control is specified to render as a table. The <table> element begins in the **HeaderTemplate** and ends in the **FooterTemplate**. (The Task List will report errors for this situation, but the errors can be safely ignored and the table will be correctly displayed when the application is run.) Within the body of the **Repeater** control, table cells are used to display columns from the data source. The **AlternatingItemTemplate** element is identical to the **ItemTemplate** item except that the background color for the table cells is different to create a banded effect.

```
<asp:Repeater id=Repeater1 runat="server" DataSource="<%# DataView1%>">
   <HeaderTemplate>
      <table width="100%" style="font: 8pt verdana">
      <tr style="background-color:DFA894">
         <th> First Name </th>
         <th> Last Name </th>
      </tr>
   </HeaderTemplate>
   <ItemTemplate>
      <tr>
         <td><%# DataBinder.Eval(Container, "DataItem.EmployeeName") %>
         </td>
         <td><%# DataBinder.Eval(Container,"DataItem.PhoneNumber") %>
         </td>
      </tr>
   </ItemTemplate>
   <AlternatingItemTemplate>
      <tr>
         <td bgcolor="lightblue">
         <%# DataBinder.Eval(Container, "DataItem.EmployeeName") %> </td>
         <td bgcolor="lightblue">
         <%# DataBinder.Eval(Container,"DataItem.PhoneNumber") %> </td>
      </tr>
   </AlternatingItemTemplate>
   <FooterTemplate> </table> </FooterTemplate>
</asp:Repeater>
```

The following example uses a label control to display values from a database. A division between values in the list is created using the separator template. Note that the value of the Text attribute of the **Label** control is contained in single quotation marks. Attribute values are normally enclosed in double quotation marks, but since the **DataBinder** call contains double quotation marks, the Text attribute value must be enclosed in single quotation marks.

```
<asp:Repeater id="Repeater1" runat="server" DataSource="<%# DataView1%>">
   <ItemTemplate>
      <asp:Label id=Label1 runat="server"
         Text='<%# DataBinder.Eval(Container, "DataItem.EmployeeName")%>'>
      </asp:Label>
   </ItemTemplate>
   <SeparatorTemplate>,</SeparatorTemplate>
</asp:Repeater>
```

See Also

Web Server Controls Templates | Responding to Button Events in DataList, Repeater, or DataGrid Items | Repeater Class | Data Access in Web Forms Pages

Table, TableRow, and TableCell Web Server Controls

The **Table** Web server control creates a general-purpose table on a Web Forms page. Rows in the table are created as **TableRow** Web server controls, and cells within each row are implemented as **TableCell** Web server controls.

The following topics provide information that you will find useful in learning to work with the **Table**, **TableRow**, and **TableCell** controls.

In This Section

Introduction to the Table Web Server Control
Provides background information on the **Table**, **TableRow**, and **TableCell** controls, including how a **Table** control compares to other list Web server controls.

Adding Table Web Server Controls to a Web Forms Page
Gives directions for adding a **Table** Web server control to a Web Forms page and adding rows and cells at design time.

Adding Rows and Cells Dynamically to a Table Web Server Control
Gives directions for creating rows and cells for a **Table** control at run time.

Displaying Database Information Using a Table Web Server Control
Gives directions for using a **Table** control to display data.

Related Sections

Setting Web Server Control Properties Programmatically
Gives directions for specifying Web server control properties in code.

Customizing the Appearance of ASP.NET Server Controls Using Styles
Gives directions for applying styles to change how controls are displayed.

Web Forms Data Binding
Provides basic conceptual information about binding data to Web Forms.

Introduction to the Table Web Server Control

The Table Web server control allows you to create server-programmable tables on your Web Forms page. The TableRow and TableCell Web server controls provide a way to display the actual content for the **Table** control.

Comparing the Table Web Server Control to Other Table Elements

Tables are typically used not only to present tabular information, but as a way to format information on a Web page. There are a number of ways to create tables on your Web Forms page:

- HTML table You can add a <TABLE> element from the HTML tab of the Toolbox.
- HtmlTable **control** This is a <TABLE> HTML element that has been converted to an HTML server control by adding the `runat=server` attribute. You can program this control in server code. For details about HTML server controls, see Introduction to ASP.NET Server Controls.
- **Table Web Server control** A Web control that allows you to create and manipulate tables (for example, adding table rows and cells) using an object model that is consistent with other Web controls.

In general, you use a **Table** Web server control when you intend to add rows and cells (columns) to the table in code at run time. Although you can use it as a static table with predefined rows and columns, it's easier in that case to work with the HTML <TABLE> element.

The **Table** Web server control can be easier to program than the HtmlTable control because it offers an object model with typed properties that is consistent with other Web server controls. (The model is also consistent between the **Table**, **TableRow**, and **TableCell** controls.)

Comparing the Table Web Server Control to Other List Web Server Controls

Some of the functions you might accomplish with a **Table** Web server control can also be accomplished with the list Web server controls: the Repeater, DataList, and DataGrid controls. All of these controls render (or have the option to render) as HTML tables. For a discussion of the differences among these three list controls, see Controls for Displaying Lists On Web Pages.

The differences between the list controls and the **Table** control are these:

- The list controls are data-bound. The list controls work only against a data source, whereas the **Table** control can display any combination of HTML text and controls, whether or not they are data-bound.
- The list controls use templates to specify the layout of their elements. The **Table** control supports the **TableCell** child control, which you can fill as you would any HTML <TD> element.

Table Web Server Control Object Model

The **Table** control acts as a parent control for TableRow controls. The table supports a property called Rows that is a collection of **TableRow** objects. By managing this collection — adding or deleting items in it — you specify the rows for the table. The **TableRow** control in turn supports a collection called Cells of TableCell objects.

The content to be displayed in the table is added to the **TableCell** control. The cell has a **Text** property that you set to any HTML text. Alternatively, you can display controls in the cell by creating instances of, and adding controls to, the cell's **Controls** collection.

The parent **Table** control supports properties to control the appearance of the entire table, such as Font, BackColor, and ForeColor. The **TableRow** and **TableCell** controls support these properties as well, so you can specify the look of individual rows or cells, overriding the parent table appearance.

Binding Data to the Control

Although the **Table** control (unlike the list Web server controls; see above) is not inherently data bound, you can use it to display data from a database.

As with all Web server controls, you can bind any property of a **Table** control to a data source. However, the **Table** control does not support a property that you use to directly display data. Instead, you typically add **TableCell** controls to the table. You can then either bind the **Text** property of individual **TableCell** controls to data, or you can add data-bound controls (such as a **Label** or **TextBox** control) to the cell.

See Also

Adding Rows and Cells Dynamically to a Table Web Server Control |
Displaying Database Information Using a Table Web Server Control | Web Forms Data Binding | Web Server Controls Templates

Adding Table Web Server Controls to a Web Forms Page

Adding a **Table** Web server control consists of two steps: adding the table itself, and separately adding the rows and cells. You can add rows and columns at design time to create a static table, or you can add them at run time in code. For details about adding them at run time, see Adding Rows and Cells Dynamically to a Table Web Server Control.

To add a Table control to a Web Forms page

- Switch to **Design** view in the Web Forms Designer, and then drag a **Table** control from the **Web Forms** tab of the Toolbox onto the page.

  The table is initially displayed as a simple text control with no rows or columns.

To create a static table, you can add rows and cells at design time.

To add rows and cells to a Table Web server control at design time

1. Select the table, and then open the Properties window and click the ellipsis button (...) for the **Rows** property.

   The **TableRow Collection Editor** window is displayed.

2. Click **Add** to create a new row.

   A new row is added, and its properties are displayed in a grid on the right-hand side of the dialog box.

3. Optionally, set display characteristics for the new row by changing its font, color, and so on.

   > **Note** Settings you enter in the **TableRow Collection Editor** dialog box apply to only one row. To apply settings that apply to all rows, set properties for the table as a whole.

4. To add cells to the new row, make sure the row is selected in the **Members** list, and then click the ellipsis button (...) for the **Cells** property.

   The **TableCell Collection Editor** dialog box is displayed.

5. Click **Add** to create a new cell, and set its properties using the grid on the right-hand side of the dialog box. To display text in the cell, set its **Text** property.
6. Repeat Step 5 for each cell you want to add to a row. When you are finished adding cells to a row, click **OK** in the **TableCell Collection Editor** dialog box.
7. Repeat Steps 2 through 6 for each new row and its cells. When you are finished adding rows to the table, click **OK** in the **TableRow Collection Editor** dialog box.

   The **Table** control is displayed in **Design** view with placeholder characters for each row and cell that you have created.

See Also

Introduction to the Table Web Server Control | Adding Rows and Cells Dynamically to a Table Web Server Control | Displaying Database Information Using a Table Web Server Control | Table Class

Adding Rows and Cells Dynamically to a Table Web Server Control

It's common to add rows and cells to a **Table** Web server control at run time. The Table Web Server control is designed specifically for this task.

> **Note** To design a table visually in Design view, use an HtmlTable control. If you also need to programmatically manipulate the contents of the **HtmlTable** control, convert its rows and cells to HtmlTableRow and HtmlTableCell controls by setting their
> runat
> attribute to
> server
> . For details, see Converting HTML Server Controls to HTML Elements.

Rows in a **Table** Web server control are objects of type **TableRow**. The Rows property of the **Table** control supports a collection of **TableRow** objects. To add a row to the table, you add a **TableRow** object to this collection.

Similarly, the **TableRow** object has a Cells property that supports a collection of objects of type **TableCell**. You can add cells to a row by manipulating this collection.

To add rows and cells to a table dynamically

1. To add a row, create a new object of type TableRow:

```
' Visual Basic
Dim tRow As New TableRow()
Table1.Rows.Add(tRow)

// C#
TableRow tRow = new TableRow();
Table1.Rows.Add(tRow);
```

2. To add cells to the row, create one or more objects of type TableCell:

```
' Visual Basic
Dim tCell As New TableCell()
tRow.Cells.Add(tCell)

// C#
TableCell tCell = new TableCell();
tRow.Cells.Add(tCell);
```

3. Add content to the new cell. You can do this in several ways, as shown in the following table.

To add	Do this
Static text	Set the cell's Text property.
Controls	Declare an instance of the control and add it to the cell's **Controls** collection.
Both text and controls in the same cell	Declare the text by creating an instance of the LiteralControl class. Add it to the cell's **Controls** collection as you would other controls.

> **Note** By default, controls that you add dynamically to a Web Forms page are added to the page's view state. If you recreate controls with each round trip, this can result in unexpected behavior when the page is processed, because view state is restored before the controls are recreated. You can avoid problems by setting the EnableViewState property of the container control (for example, of the **Table** control) to **false**. For more information, see Adding Controls to a Web Forms Page Programmatically.

The following example shows you can add rows and cells to a **Table** control. The number of rows and columns is determined by what the user enters into two text boxes. Each cell displays the row and cell number as static text.

```
' Visual Basic
Public Sub Button1_Click(ByVal sender As Object, ByVal e As System.EventArgs) Handles But
```

```
ton1.Click
   ' Total number of rows
   Dim rowCnt As Integer
   ' Current row count
   Dim rowCtr As Integer
   ' Total number of cells (columns)
   Dim cellCtr As Integer
   ' Current cell counter
   Dim cellCnt As Integer

   rowCnt = CInt(Textbox1.Text)
   cellCnt = CInt(Textbox2.Text)

   For rowCtr = 1 To rowCnt
      Dim tRow As New TableRow()
      For cellCtr = 1 To cellCnt
         Dim tCell As New TableCell()
         tCell.Text = "Row " & rowCtr & ", Cell " & cellCtr
         ' Add new TableCell object to row
         tRow.Cells.Add(tCell)
      Next
      ' Add new row to table
      Table1.Rows.Add(tRow)
   Next

End Sub

// C#
public void Button1_Click (object sender, System.EventArgs e){
   // Total number of rows
   int rowCnt;
   // Current row count
   int rowCtr;
   // Total number of cells per row (columns)
   int cellCtr;
   // Current cell counter
   int cellCnt;

   rowCnt = int.Parse(TextBox1.Text);
   cellCnt = int.Parse(TextBox2.Text);

   for(rowCtr=1; rowCtr <= rowCnt; rowCtr++) {
      // Create new row and add it to the table.
      TableRow tRow = new TableRow();
      Table1.Rows.Add(tRow);
      for (cellCtr = 1; cellCtr <= cellCnt; cellCtr++) {
         // Create a new cell and add it to the row.
         TableCell tCell = new TableCell();
         tCell.Text = "Row " + rowCtr + ", Cell " + cellCtr;
         tRow.Cells.Add(tCell);
      }
   }
}
```

The following example is similar to the previous one, but displays static text and a HyperLink control in each cell. The **Hyperlink** control navigates to a mocked-up URL, passing a mock product ID. Because the sample mixes static text and controls, the static text is implemented as a **LiteralControl** object, which is added to the cell's **Controls** collection just like the **Hyperlink** control is.

```
' Visual Basic
Public Sub Button1_Click(ByVal sender As Object, ByVal e As System.EventArgs) Handles But
ton1.Click
   ' Total number of rows
   Dim rowCnt As Integer
   ' Current row count
   Dim rowCtr As Integer
   ' Total number of cells (columns)
   Dim cellCtr As Integer
   ' Current cell counter
   Dim cellCnt As Integer

   rowCnt = CInt(TextBox1.Text)
   cellCnt = CInt(TextBox2.Text)

   For rowCtr = 1 To rowCnt
      Dim tRow As New TableRow()
      For cellCtr = 1 To cellCnt
         Dim tCell As New TableCell()
         ' Mock up a product ID
         Dim prodID As String
         prodID = rowCtr & "_" & cellCtr

         ' Create literal text as control
         Dim s As New LiteralControl()
         s.Text = "Buy: "
         tCell.Controls.Add(s)   ' Add to cell
         ' Create Hyperlink Web Server control and add to cell
         Dim h As New HyperLink()
         h.Text = rowCtr & ":" & cellCtr
         h.NavigateUrl = "http://www.microsoft.com/net"
         tCell.Controls.Add(h)
         ' Add cell to row
         tRow.Cells.Add(tCell) ' Add new TableCell object to row
      Next cellCtr
      Table1.Rows.Add(tRow) ' Add new row to table
   Next rowCtr
End Sub

// C#
public void Button1_Click (object sender, System.EventArgs e){
   // Total number of rows
   int rowCnt;
   // Current row count
   int rowCtr;
   // Total number of cells per row (columns)
   int cellCtr;
   // Current cell counter
   int cellCnt;


   rowCnt = int.Parse(TextBox1.Text);
   cellCnt = int.Parse(TextBox2.Text);

   for(rowCtr=1; rowCtr <= rowCnt; rowCtr++) {
      // Create new row and add it to the table.
      TableRow tRow = new TableRow();
      Table1.Rows.Add(tRow);
      for (cellCtr = 1; cellCtr <= cellCnt; cellCtr++) {
```

```
            // Create a new cell and add it to the row.
            TableCell tCell = new TableCell();
            tRow.Cells.Add(tCell);
            // Mock up a product ID.
            string prodID = rowCtr + "_" + cellCtr;

            // Add a literal text as control.
            tCell.Controls.Add(new LiteralControl("Buy: "));
            // Create Hyperlink Web Server control and add to cell
            System.Web.UI.WebControls.HyperLink h = new HyperLink();
            h.Text = rowCtr + ":" + cellCtr;
            h.NavigateUrl = "http://www.microsoft.com/net";
            tCell.Controls.Add(h);
        }
    }
}
```

See Also

Introduction to the Table Web Server Control | Displaying Database Information Using a Table Web Server Control | Table Class

Displaying Database Information Using a Table Web Server Control

It's possible to display database information using the **Table** Web server control.

> **Tip** For many applications, it is easier to display data-bound information in a table by using the Repeater, DataList, or DataGrid Web server controls.

In the most common scenario for displaying data in a **Table** Web server control, each row of the table represents a data row from the data source, and each cell of the table displays a single field from a row. Unlike the list controls (**Repeater**, **DataList**, and so on), the **Table** control does not have a mechanism for automatically iterating through the rows in a data source. Therefore, you must do this yourself.

For general details about data binding in Web server controls, see Web Forms Data Binding.

To display database information in a Table Web server control

1. Add a dataset to the page. For information on how to do this, see Walkthrough: Creating a Distributed Application, Walkthrough: Simple Data Access in a Windows Form, or Creating Typed Datasets with the Component Designer. For a discussion of options in creating datasets from database tables, see Introduction to ADO.NET Connection Design Tools, Introduction to Data Adapters, and Introduction to Datasets.
2. Add a row to the table for each row in the dataset, and then set the contents of each **TableCell** object in the row to the data to be displayed. Most often you set the cell's **Text** property to an expression that extracts data from a data source. It's also possible to include controls in a **TableCell** control and bind the controls to a source of data. For details about using controls to display information, see Adding Rows and Cells Dynamically to a Table Web Server Control.

   The following example illustrates one way to display data in a **Table** Web server control. The sample shows an event handler for a button. The method loops through the **DefaultView** object of a dataset, creating a **TableRow** control for each row in the dataset. The code then creates a **TableCell** control in each table row and sets its **Text** property to the "name" field of the current data row.

```
' Visual Basic
Private Sub Button1_Click(ByVal sender As System.Object, _
ByVal e As System.EventArgs) Handles Button1.Click
   Dim aDataView As DataView = authorData.Tables(0).DefaultView
   Dim aRow As DataRowView
   For Each aRow In aDataView
      Dim aTableRow As New TableRow()
      Dim aCell As New TableCell()
      aCell.Text = aRow.Item("last").ToString()
      aTableRow.Cells.Add(aCell)
      Table1.Rows.Add(aTableRow)
   Next
End Sub

// C#
private void Button1_Click(object sender, System.EventArgs e)
{
   DataView aDataView = authorData.Tables[0].DefaultView;
   foreach( DataRowView aRow in aDataView)
   {
      TableRow aTableRow = new TableRow();
      TableCell aCell = new TableCell();
      aCell.Text = aRow["last"].ToString();
      aTableRow.Cells.Add(aCell);
      Table1.Rows.Add(aTableRow);
   }
}
```

Note Controls that you add dynamically to a Web Forms page do not automatically become part of the page's view state — neither the controls nor their values are saved when a page performs a round trip to the server. You are therefore responsible for saving the state of any dynamically-generated controls whose values you want to preserve. For details, see Introduction to Web Forms State Management.

See Also

Introduction to the Table Web Server Control | Adding Rows and Cells Dynamically to a Table Web Server Control | Web Forms Data Binding | Table Class

TextBox Web Server Control

The **TextBox** Web server control provides a way for users to type information into a Web Forms page, including text, numbers, and dates.

The following topics provide information that you will find useful in learning to work with the **TextBox** Web server control.

In This Section

Introduction to the TextBox Web Server Control
Provides background information on the **TextBox** control.
Getting and Setting Values in TextBox Web Server Controls
Gives directions for working with the contents of a **TextBox** control at run time.
Responding to Changes in a TextBox Web Server Control
Gives directions for getting notification when a user has entered information into a **TextBox** control.
Setting a TextBox Web Server Control for Password Entry
Gives directions for using a **TextBox** control to accept passwords.
Specifying Multiline Input for a TextBox Web Server Control
Gives directions for allowing users to enter multiple lines of text.

Related Sections

Adding Web Server Controls to a Web Forms Page
Gives directions for including Web server controls on your Web Form.
Setting ASP.NET Server Control Properties
Suggests additional topics of interest concerning server control properties.
Setting Web Server Control Properties Programmatically
Gives directions for setting Web server control properties in code
Introduction to Validating User Input in Web Forms
Explains validating user entries in a TextBox control.
Data Binding Single-Value Web Server Controls at Design Time
Gives directions for displaying data in a **TextBox** control.

Introduction to the TextBox Web Server Control

The TextBox Web server control provides a way for users to type information into a Web Form, including text, numbers, and dates. You can configure the **TextBox** Web server control in several ways:

Configuration	Description
Single-line	Users type information in a single line. You can optionally limit the number of characters that the control accepts.
Password	Like a single-line **TextBox** control, but the characters users type are masked with asterisks (*) to hide them. **Security Note** Using a **TextBox** control set for passwords can help ensure that other people will not be able to determine a user's password if they observe the user entering it. However, the password text entered is not encrypted in any way, and you should protect it as you would any other confidential data. For example, for maximum security when posting a form with a password in it, you might use secure sockets layer (SSL) and encryption. For more information, see Web Application Security at Run Time.
Multi-line	Users type information in a box that displays multiple lines and allows text wrapping.

Security Note User input in a Web Forms page can include potentially malicious client script. By default, the Web Forms page validates that user input does not include script or HTML elements. For more information, see Scripting Exploits and Protecting Against Script Exploits in a Web Application.

The **TextBox** control raises a TextChanged event when the user leaves the control. By default, the event is not raised immediately; instead, it is raised on the server when the Web Forms page is submitted. However, you can specify that the TextBox control should submit the page to the server as soon as the user leaves the field.

Note The **TextBox** Web server control does not raise an event each time the user enters a keystroke, only when the user leaves the control.

See Also

Adding Web Server Controls to a Web Forms Page | Introduction to Validating User Input in Web Forms | Setting Web Server Control Properties Programmatically | Customizing the Appearance of ASP.NET Server Controls Using Styles

Getting and Setting Values in TextBox Web Server Controls

You can get or set the value of a TextBox at run time.

> **Security Note** User input in a Web Forms page can include potentially malicious client script. By default, the Web Forms page validates that user input does not include script or HTML elements. For more information, see Scripting Exploits and Protecting Against Script Exploits in a Web Application.

To get or set values in the TextBox control

- Get or set the value of the control's Text property.

  The following example shows how you can read the value of one TextBox control (called `TextBox1`) and use it to set the value of another (`TextBox2`).

```
' Visual Basic
Public Sub Button1_Click(ByVal sender As Object, ByVal e As System.EventArgs) Handles But
ton1.Click
   Textbox2.Text = TextBox1.Text
End Sub

// C#
public void Button1_Click(object sender, System.EventArgs e)
{
   TextBox2.Text = TextBox1.Text;
}
```

See Also

ASP.NET Server Control Event Model | Setting Web Server Control Properties Programmatically | TextBox Class

Responding to Changes in a TextBox Web Server Control

When the user leaves a **TextBox** Web server control after entering information, the control raises an event that you can respond to.

> **Note** The **TextBox** Web server control does not raise an event each time the user enters a keystroke, only when the user leaves the control.

To respond to changes in the TextBox control

- Create an event handler for the **TextBox** control's TextChanged event.

  By default, the **TextChanged** event does not immediately cause the Web Forms page to be posted to the server. Instead, the event is raised in server code the next time the form is posted, as when a Button Web server control is clicked. To have the **TextChanged** event cause an immediate posting, set the **TextBox** control's AutoPostBack property to **true**.

  > **Note** The ability of a **TextBox** control to post to the server when it is checked requires that the browser support ECMAScript (JScript, JavaScript) and that scripting be enabled on the user's browser.

  The following example shows how you can respond to changes in the **TextBox** control. The code displays the contents of the control in a label.

  > **Security Note** User input in a Web Forms page can include potentially malicious client script. By default, the Web Forms page validates that user input does not include script or HTML elements. For more information, see Scripting Exploits and Protecting Against Script Exploits in a Web Application.

```
' Visual Basic
Public Sub TextBox1_TextChanged(ByVal sender As Object, ByVal e As System.EventArgs) Hand
les TextBox1.TextChanged
   Label1.Text = TextBox1.Text
End Sub

// C#
public void TextBox1_TextChanged(object sender, EventArgs e)
{
   Label1.Text = TextBox1.Text;
}
```

See Also

ASP.NET Server Control Event Model | Specifying Whether Controls Post to the Server | Creating Event Handlers in Web Forms Pages | TextBox Class

Setting a TextBox Web Server Control for Password Entry

A **TextBox** Web server control set for passwords hides a user's entry as it is being typed. Password text boxes can only be single-line text boxes.

> **Security Note** Using a **TextBox** control set for passwords can help ensure that other people will not be able to determine a user's password if they observe the user entering it. However, the password text entered is not encrypted in any way, and you should protect it as you would any other confidential data. For example, for maximum security when posting a form with a password in it, you might use secure sockets layer (SSL) and encryption. For more information, see Web Application Security at Run Time.

To create a password textbox

1. Set the **TextBox** control's TextMode property to **Password**. In code, text modes are set using the TextBoxMode enumeration.
2. Set the size of the control by setting its Width property to a value in pixels or its Columns property to the number of characters to display. If you set both properties, **Width** takes precedence.

   > **Note** The **Width** property does not work in browsers that do not support CSS styles.

3. Optionally, you can limit the number of characters the user is allowed to enter by setting the control's MaxLength property. If the users exceed this number of characters, the control stops accepting more.

   > **Note** You might not want to set the **MaxLength** property, because it can be of use to unauthorized users who are trying to guess the password.

See Also

Introduction to Validating User Input in Web Forms | Setting Web Server Control Properties Programmatically | Data Binding Single-Value Web Server Controls at Design Time | Customizing the Appearance of ASP.NET Server Controls Using Styles | Responding to Changes in a TextBox Web Server Control | TextBox Class

Specifying Multiline Input for a TextBox Web Server Control

A multi-line **TextBox** control allows users to type information into a box that supports word wrap and vertical and horizontal scrolling.

To create a multiline TextBox Web server control

1. Set the **TextBox** control's TextMode property to **MultiLine**. In code, text modes are set using the TextBoxMode enumeration.
2. Set the size of the control by setting the following properties:

   - Width to a value in pixels, or Columns to the number of characters to display.
   - Height to a value in pixels, or Rows to the number of rows.

     Note The **Height** and **Width** property do not work in browsers that do not support CSS styles.

   Setting these properties does not limit the number of characters or rows the user can enter, only how many will display. The control always displays a vertical scrollbar. **Height** and **Width** take precedence over **Rows** and **Columns**.

   Tip By default, a multi-line **TextBox** control includes a vertical scrollbar. To hide the scrollbar, include the style attribute
   overflow: hidden
   . For example, in HTML view, the declarative syntax for the control might look as follows:

```
<asp:TextBox id="TextBox1" style="Z-INDEX: 101; LEFT: 56px; OVERFLOW: hidden; POSITION: a
bsolute; TOP: 72px" runat="server" TextMode="MultiLine"></asp:TextBox>
```

3. Set word-wrap behavior by setting the control's Wrap property. If true, text wraps automatically (without embedding carriage-return/linefeed characters). If false, the text does not wrap and the user must press ENTER to start a new line, which embeds a carriage-return/linefeed character. Setting **Wrap** to **false** causes the control to display a horizontal scrollbar.

   Note The MaxLength property has no effect in a multi-line textbox.

See Also

Introduction to Validating User Input in Web Forms | Setting Web Server Control Properties Programmatically | Data Binding Single-Value Web Server Controls at Design Time | Customizing the Appearance of ASP.NET Server Controls Using Styles | Responding to Changes in a TextBox Web Server Control | TextBox Class

Validation Web Server Controls

There is a set of Web server controls that you can use to validate input on your Web Forms pages. These controls ensure that the information that a user has provided matches the requirements of your application. There are standard controls for different types of validation, such as range checking or pattern matching, and a custom control that you can use to establish your own validation criteria. For more information on validation, see Introduction to Validating User Input in Web Forms.

The validation controls listed below appear on the **Web Forms** tab of the Toolbox.

In the Visual Basic and Visual C# Documentation

RequiredFieldValidator
Ensures that the user does not skip a required field when entering data on a Web Forms page. For details, see Validating Required Entries for ASP.NET Server Controls.

CompareValidator
Compares a user's entry against a constant value, against the property value of another control, or against a database value. For details, see Validating Against a Specific Value for ASP.NET Server Controls or Validating Against a Data Type for ASP.NET Server Controls.

RangeValidator
Ensures that a user's entry is between specified upper and lower bounds. For details, see Validating Against a Range of Values for ASP.NET Server Controls.

RegularExpressionValidator
Ensures that a user's entry matches a pattern defined by a regular expression. For details, see Validating Against Patterns for ASP.NET Server Controls.

CustomValidator
Ensures that a user's entry matches validation logic you create yourself. For details, see Validating with a Custom Function for ASP.NET Server Controls.

ValidationSummary
Provides a central location to display validation error messages. For details, see Controlling Validation Error Message Display for ASP.NET Server Controls.

Related Sections

Web Forms Validation
Lists topics that provide information on how to use the validation controls to verify user input on a Web Forms page.

Introduction to Validating User Input in Web Forms
Provides background information about using validation controls.

Client-Side Validation for ASP.NET Server Controls
Explains how validation works in browsers that support Dynamic HTML (such as Internet Explorer).

Special-Case Validation Results for ASP.NET Server Controls
Describes how validation controls work under specific circumstances and how they interact with one another.

Controlling Validation Error Message Display for ASP.NET Server Controls
Gives directions for specifying where error messages are displayed.

Testing Validity Programmatically for ASP.NET Server Controls
Gives directions for testing in your code whether the user's entries passed validation checking.

XML Web Server Control

An **XML** Web server control reads XML and writes it into a Web Forms page at the location of the control. If an XSL Transformations (XSLT) is applied to the XML, the resulting transformed output will be rendered in the Web Forms page.

The following topics provide information that you will find useful in learning to work with the XML Web server control.

In This Section

Introduction to the XML Web Server Control
Provides background information on the XML Web server control and describes how the control is used.
Adding XML Web Server Controls to a Web Forms Page
Gives directions for including XML Web server controls on a form.
Loading XML Data in the XML Web Server Control
Gives directions for the three methods of setting up the XML Web server control.
Transforming XML Data in the XML Web Server Control
Gives directions for including an XSL document to transform your XML output.
Walkthrough: Displaying an XML Document in a Web Forms Page using Transformations
Gives step-by-step details on how to read an XML file and display it in different ways on a Web Forms page.

Related Sections

Walkthrough: Displaying an XML Document in a Web Forms Page using Transformations
Illustrates how to display information from an XML document on a Web Forms page.
Controls You Can Use on Web Forms Pages
Links to a description of each type of Web server control.
Adding Web Server Controls to a Web Forms Page
Gives directions for including Web server controls on a form.
XML Files
Introduces topics that explain how to use the XML Designer when working with XML documents.
XML in Visual Studio
Introduces topics about XML as it relates to various Visual Studio feature areas, including data, XML Web services, Web Forms, and reference materials.

Introduction to the XML Web Server Control

You can use the XML Web server control to write an XML document, or the results of an XSL Transformations (XSLT), into a Web page. The XML output appears in the Web page at the location of the control. For more information on XML and XSLT, see XML (Extensible Markup Language) in the Microsoft XML SDK documentation.

The XML and the XSLT information can be in external documents, or you can include the XML inline. There are two ways to reference external documents using property settings in the **XML** Web server control. You can provide a path to the XML document in the control tag, or you can load the XML and XSLT documents as objects and then pass them to the control. If you prefer to include the XML inline, write it between the opening and closing tags of the control.

See Also

Adding XML Web Server Controls to a Web Forms Page | Loading XML Data in the XML Web Server Control | Transforming XML Data in the XML Web Server Control | Walkthrough: Displaying an XML Document in a Web Forms Page using Transformations | XML in Visual Studio

Adding XML Web Server Controls to a Web Forms Page

Add **XML** Web server controls to the place in the page where you want the output to appear.

To add an XML Web server control to a Web Forms page

1. In Design view, drag an **XML** control from the **Web Forms** tab of the Toolbox onto the page.
2. Load the XML data that you want to display into the control by setting the **Document** or **DocumentSource** property, or by putting the XML between the control's opening and closing tags. For details, see Loading XML Data in the XML Web Server Control.

   Note You need to be sure that when your application runs, it has sufficient permissions to read the XML file. For details, see Access Permissions for Web Applications.

To add a control to a Web Forms page programmatically

1. Create an instance of the control and set its properties:

```
' Visual Basic
Dim myXML As System.Web.UI.WebControls.Xml = _
      New System.Web.UI.WebControls.Xml()
myXML.DocumentSource = "SourceDoc.xml"
myXML.TransformSource = "SourceTrans.xsl"

// C#
System.Web.UI.WebControls.Xml myXML =
      new System.Web.UI.WebControls.Xml();
myXML.DocumentSource = "SourceDoc.xml";
myXML.TransformSource = "SourceTrans.xsl";
```

2. Add the new control to the **Controls** collection of a container already on the page:

```
' Visual Basic
PlaceHolder1.Controls.Add(myXML)

// C#
PlaceHolder1.Controls.Add(myXML);
```

   Note Controls that you add dynamically to a Web Forms page do not automatically become part of the page's view state — neither the controls nor their values are saved when a page performs a round trip to the server. You are therefore responsible for saving the state of any dynamically-generated controls whose values you want to preserve. For details, see Introduction to Web Forms State Management.

See Also

Introduction to the XML Web Server Control | Transforming XML Data in the XML Web Server Control | Walkthrough: Displaying an XML Document in a Web Forms Page using Transformations

Loading XML Data in the XML Web Server Control

There are three ways to load XML data into the **XML** Web server control.

- Provide a path to an external XML document, using the **DocumentSource** property.
- Load an XML document as an object and pass it to the control, using the **Load** method in the **Page_Load** event and assigning the document to the **Document** property of the **XML** control.
- Include the XML content inline, between the opening and closing tags of the control.

To provide a path to an external XML document

1. Add an **XML** control to the Web Forms page. For details, see Adding XML Web Server Controls to a Web Forms Page.
2. Set the **DocumentSource** property of the control to the path to the XML source document.

   > **Note** You need to be sure that when your application runs, it has sufficient permissions to read the XML file. For details, see Access Permissions for Web Applications.

   The XML document will be written directly to the output stream unless you also specify the **TransformSource** property. **TransformSource** must be a valid XSL Transformations document, which will be used to transform the XML document before its contents are written to the output stream. The following example shows how to refer to source documents by using a relative path.

```
<body>
    <h3>XML Example</h3>
    <form runat=server>
        <asp:Xml id="Xml1" DocumentSource="MySource.xml"
            TransformSource="MyStyle.xsl" runat="server" />
    </form>
</body>
```

To load an XML document as an object and pass it to the control

1. Add an **XML** control to the Web Forms page. For details, see Adding XML Web Server Controls to a Web Forms Page.
2. On the **View** menu, click **Code**. In the Code Editor, find the **Page_Load** procedure.
3. Add code to load the XML source document, and assign the source to the **Document** property of the control. For example:

```
' Visual Basic
Private Sub Page_Load(ByVal sender As System.Object, _
      ByVal e As System.EventArgs) Handles MyBase.Load
   Dim doc As System.Xml.XmlDocument = New System.Xml.XmlDocument()
   doc.Load(Server.MapPath("MySource.xml"))
   Dim trans As System.Xml.Xsl.XslTransform = _
      New System.Xml.Xsl.XslTransform
   trans.Load(Server.MapPath("MyStyle.xsl"))
   Xml1.Document = doc
   Xml1.Transform = trans
End Sub

// C#
private void Page_Load(object sender, System.EventArgs e)
{
    System.Xml.XmlDocument doc = new System.Xml.XmlDocument();
    doc.Load(Server.MapPath("MySource.xml"));
    System.Xml.Xsl.XslTransform trans = new
       System.Xml.Xsl.XslTransform();
    trans.Load(Server.MapPath("MyStyle.xsl"));
    Xml1.Document = doc;
    Xml1.Transform = trans;
}
```

To include the XML content inline

1. Add an **XML** control to the Web Forms page. For details, see Adding XML Web Server Controls to a Web Forms Page.
2. In HTML view, find the `<asp:Xml>` and `</asp:Xml>` tags.
3. Add your XML code between these two tags. For example:

```
<asp:xml TransformSource="MyStyle.xsl" runat=server>
    <clients>
        <name>Frank Miller</name>
        <name>Judy Lew</name>
    </clients>
</asp:xml>
```

See Also

Introduction to the XML Web Server Control | Adding XML Web Server Controls to a Web Forms Page |
Transforming XML Data in the XML Web Server Control |
Walkthrough: Displaying an XML Document in a Web Forms Page using Transformations

Transforming XML Data in the XML Web Server Control

An XSL transformations (XSLT) style sheet (.xslt or .xsl file) is used to transform the content of a source XML document into a presentation that is tailored specifically to a particular user, media, or client. For an overview, see the XSLT Developer's Guide. There are two ways to transform XML data in the **XML** Web server control:

- Point to an external XSLT file, which will automatically apply the transformation to the XML document.
- Apply a transformation that is an object of type XslTransform to the XML document.

Both methods have the same results, and your choice is dependent primarily on what is most convenient in your application. If the transformation is in the form of an .xsl or .xslt file, it is easy to load the file. If the transformation is in the form of an object — perhaps it is being passed to your application by another process — then you can apply it as an object.

> **Note** The **XslTransform** class also allows you to load an .xsl or .xslt file into the instance of the transformation.

To apply a transformation from a file

1. Add an XML control to the Web Forms page. For details, see Adding XML Web Server Controls to a Web Forms Page.
2. Set the **XML** control's TransformSource property to the path to the XSLT document.

   > **Note** You need to be sure that when your application runs, it has sufficient permissions to read the XML file. For details, see Access Permissions for Web Applications.

   The following example shows how you can apply a transformation from a file to an **XML** control called `Xml1`.

```
'Visual Basic
Xml1.TransformSource = "mystyle.xsl"

//C#
Xml1.TransformSource = "mystyle.xsl";
```

To apply a transformation from an XslTransform object

1. Create an instance of the XslTransform class.
2. Set the **XML** control's Transform property to the instance of the transformation.

   The following example shows how you can create an instance of the transformation class and use it to apply the transformation to an object. In this example, both the XML document and the transformation are read from files, but in a real application, both objects might come from another component. The transformation is applied as soon as the page loads.

```
'Visual Basic
Private Sub Page_Load(ByVal sender As System.Object, _
      ByVal e As System.EventArgs) Handles MyBase.Load
   Dim doc As System.Xml.XmlDocument = New System.Xml.XmlDocument()
   doc.Load(Server.MapPath("MySource.xml"))
   Dim trans As System.Xml.Xsl.XslTransform = _
      New System.Xml.Xsl.XslTransform
   trans.Load(Server.MapPath("MyStyle.xsl"))
   Xml1.Document = doc
   Xml1.Transform = trans
End Sub

// C#
private void Page_Load(object sender, System.EventArgs e)
{
    System.Xml.XmlDocument doc = new System.Xml.XmlDocument();
    doc.Load(Server.MapPath("MySource.xml"));
    System.Xml.Xsl.XslTransform trans = new
       System.Xml.Xsl.XslTransform();
```

```
        trans.Load(Server.MapPath("MyStyle.xsl"));
        Xml1.Document = doc;
        Xml1.Transform = trans;
    }
```

See Also

Introduction to the XML Web Server Control | Adding XML Web Server Controls to a Web Forms Page |
Loading XML Data in the XML Web Server Control |
Walkthrough: Displaying an XML Document in a Web Forms Page using Transformations

Walkthrough: Displaying an XML Document in a Web Forms Page using Transformations

As more and more information becomes available in the format of XML documents, you will find it increasingly likely that you will need to be able to work with these documents. This walkthrough illustrates how you can easily display information on a Web Forms page from a "non-structured" XML document — that is, an XML document that does not represent relational data.

XML and XSL in Web Forms Pages

Information in an XML file is "raw," containing only the data and no indication about how to format it, display it, and so on. To display XML information in a Web Forms page, you need to provide formatting and display information — for example, you need to provide the <TABLE> tags, <P> tags, or whatever you want to use to display the information. In addition, you must provide instructions for how the data from the XML file fits into these tags. For example, should each element in the XML file be displayed as a table row?

One way to provide these instructions is to create a Web Forms page and then use code to parse the XML file and fill the data into the appropriate HTML tags. However, this is a somewhat tedious and inflexible way to do so. (On the other hand, it is also a very powerful way that gives you precise programmatic control over the XML file.)

XSL Transformations

A far better way is to use the XSL transformation language and create "transformations," or XSL files. An XSL transformation contains this information:

- A template — basically, an HTML page with the appropriate tags — that specifies how the XML information should be displayed.
- XSL processing instructions, which specify exactly how the information from the XML file fits into the template.

You can define multiple transformations and then apply them to the same XML file. By doing so, you can use the XML information different ways, displaying it differently, selecting different data from the XML file, and so on.

Once you have XSL transformations, you need to apply them to the XML file — that is, you process the XML file by transforming it according to one of the XSL files. The output is a new file with the XML information formatted according to the transformation file.

Again, this is a process you can perform programmatically. However, you can also take advantage of the **XML** Web server control. (For an overview see XML Web Server Control.) This control works as a placeholder on your Web Forms page. You can set its properties to a specific XML file and XSL transformation, and when the page is processed, the control reads the XML, applies the transformation, and displays the results.

For more information about XSL, see XSLT Transformations with the XslTransform Class.

Goals of the Walkthrough

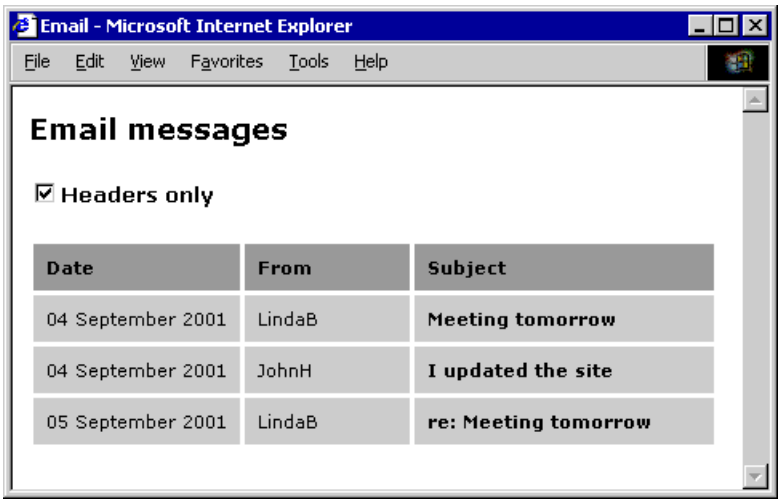
This walkthrough will illustrate how to use the **XML** Web server control to display XML information using XSL transformations. The scenario is simple. You will have the following:

- An XML file containing several fictional e-mail messages.
- Two XSL transformations. One displays only the date, sender, and subject of the e-mail messages. The other displays the entire e-mail message.

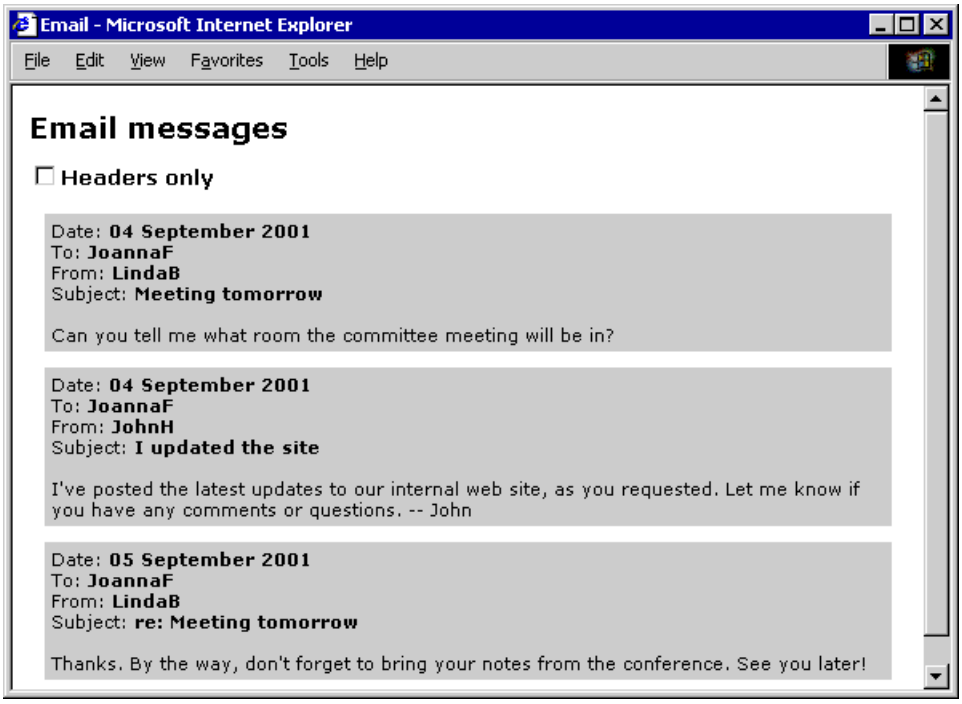
> **Note** All of these files are provided as part of the walkthrough.

You will create a Web Forms page that allows users to display the XML information two different ways. The page contains a "Headers only" check box that is checked by default; the default transformation, therefore, is the one that displays only the e-mail message headers. If users clear the check box, the XML information is redisplayed with the complete e-mail messages.

When the page displays only the headers, it might look something like the following:



When users clear the check box, the page changes to look something like this:



In order to complete this walkthrough, you will need sufficient permissions to be able to create ASP.NET Web application projects on the computer where your Web server is.

As part of the walkthrough, you will do the following:

- Create a new Web application project and Web Forms page.
- Add the XML file and the two XSL transformation files to the project.
- Add an **XML** Web server control to the Web Forms page and set it to display the XML file and a transformation.
- Add controls and code to allow the user to change the transformation applied to the XML file.

Creating the Project and Form

The first step is to create a Web application and a Web Forms page.

To create the project and form

1. On the **File** menu, point to **New**, then click **Project**.
2. In the **New Project** dialog box, do the following:
   a. In the **Project Types** pane, choose either **Visual Basic Projects** or **Visual C# Projects**.
   b. In the **Templates** pane, choose **ASP.NET Web Application**.
   c. In the **Location** box, enter the complete URL for your application, including http://, the name of the server, and the name of your project. The Web server must have IIS version 5 or later and the .NET Framework installed on it. If you have IIS on your computer, you can specify `http://localhost` for the server. (If you normally use a proxy server to access the Internet, you might need to configure Internet Explorer to bypass the proxy server in order to use localhost.)

When you click **OK**, a new Web Forms project is created at the root of the Web server you specified. In addition, a new Web Forms page called WebForm1.aspx is displayed in the Web Forms Designer in Design view.

> **Tip** If you have trouble creating a Web application project, see Web Access Failed Dialog Box.

Adding the XML File and XSL Transformations

The walkthrough scenario assumes that you are working with an existing XML file and two existing XSL transformation files. In a real application, you would create them yourself or get them from some source (perhaps another application). To save you the trouble of having to create or acquire these files, they are included as part of the walkthrough. The following steps (which would not be part of a real application) make these files available so you can continue with the walkthrough.

To add the XML file to your project

1. In Solution Explorer, right-click the name of your project, point to **Add**, then choose **Add New Item**.

   The **Add New Item** dialog box is displayed.

2. In the **Templates** pane, choose **XML File**.
3. In the Name box, change the name of the XML file to **Emails.xml**.

   > **Note** You can name the XML file anything you like. If you use a different name, make a note of it so you can use your name as appropriate later.

   The XML editor opens with a new XML file in it with an XML directive at the top.

4. Select the XML directive (and anything else in the file) and delete it.
5. Copy the following XML information and paste it into the file using the **Paste As HTML** command:

```
<?xml version="1.0" ?>
<MESSAGES>
   <MESSAGE id="101">
      <TO>JoannaF</TO>
      <FROM>LindaB</FROM>
      <DATE>04 September 2001</DATE>
      <SUBJECT>Meeting tomorrow</SUBJECT>
      <BODY>Can you tell me what room the committee meeting will be in?</BODY>
   </MESSAGE>
   <MESSAGE id="109">
      <TO>JoannaF</TO>
      <FROM>JohnH</FROM>
      <DATE>04 September 2001</DATE>
      <SUBJECT>I updated the site</SUBJECT>
      <BODY>I've posted the latest updates to our internal web site, as you requested. Le
t me know if you have any comments or questions. -- John
      </BODY>
   </MESSAGE>
   <MESSAGE id="123">
      <TO>JoannaF</TO>
      <FROM>LindaB</FROM>
      <DATE>05 September 2001</DATE>
      <SUBJECT>re: Meeting tomorrow</SUBJECT>
      <BODY>Thanks. By the way, do not forget to bring your notes from the conference. Se
e you later!</BODY>
   </MESSAGE>
</MESSAGES>
```

6. Save the file and close it.

Next you will add two XSL transformations to your project.

To add XSL transformations to your project

1. In Solution Explorer, right-click the name of your project, point to **Add**, then choose **Add New Item**.

   The **Add New Item** dialog box appears.

2. In the **Templates** pane, choose **XSLT File**.
3. In the **Name** box, change the name of the file to **Email_headers.xslt**.

   > **Note** You can name the XML file anything you like. If you use a different name, make a note of it so you can use your name as appropriate later.

   The XML editor opens with some XSLT directives and elements in it.

4. Select everything in the file and delete it.
5. Copy the following and paste it into the file using the **Paste As HTML** command:

```
<?xml version='1.0'?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
<xsl:template match="/">
<HTML>
<BODY>
<TABLE cellspacing="3" cellpadding="8">
   <TR bgcolor="#AAAAAA">
      <TD class="heading"><B>Date</B></TD>
      <TD class="heading"><B>From</B></TD>
      <TD class="heading"><B>Subject</B></TD>
   </TR>
   <xsl:for-each select="MESSAGES/MESSAGE">
   <TR bgcolor="#DDDDDD">
       <TD width="25%" valign="top">
         <xsl:value-of select="DATE"/>
       </TD>
      <TD width="20%" valign="top">
         <xsl:value-of select="FROM"/>
      </TD>
       <TD width="55%" valign="top">
         <B><xsl:value-of select="SUBJECT"/></B>
      </TD>
   </TR>
   </xsl:for-each>
</TABLE>
</BODY>
</HTML>
</xsl:template>
</xsl:stylesheet>
```

   > **Note** You can adjust this template as you like; for example, you might change colors, font sizes, styles, or other features.

6. Save the file and close it.
7. Repeat Steps 1 through 4, but this time name the file **Email_all.xslt**.
8. Paste the following into the **Email_all.xslt** file using the **Paste As HTML** command:

```
<?xml version='1.0'?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
<xsl:template match="/">
<HTML>
<BODY>
<FONT face="Verdana" size="2">
<TABLE cellspacing="10" cellpadding="4">
   <xsl:for-each select="MESSAGES/MESSAGE">
   <TR bgcolor="#CCCCCC">
```

```
    <TD class="info">
          Date: <B><xsl:value-of select="DATE"/></B><BR></BR>
          To: <B><xsl:value-of select="TO"/></B><BR></BR>
          From: <B><xsl:value-of select="FROM"/></B><BR></BR>
          Subject: <B><xsl:value-of select="SUBJECT"/></B><BR></BR>
          <BR></BR><xsl:value-of select="BODY"/>
        </TD>
    </TR>
    </xsl:for-each>
  </TABLE>
  </FONT>
  </BODY>
  </HTML>
  </xsl:template>
  </xsl:stylesheet>
```

Note You can adjust this template as you like; for example, you might change colors, font sizes, styles, or other features.

9. Save the file and close it.

Adding the XML Control to the Web Forms Page

Now that you have an XML file and two transformations, you can use them to display information in a Web Forms page. You do so using an **XML** Web server control.

When you are working with the **XML** control, it is easier to work in flow layout rather than grid layout, because you cannot use absolute positioning to specify the location of the XML control on the page. For more information, see pageLayout Property.

To set flow mode

1. Right-click the empty page and choose **Properties**.
2. Set the **Page Layout** value (**pageLayout** property) to **FlowLayout**.

Now you can add controls to the page.

To add an XML control

1. Open or move to the Web Forms page created by default when you created you Web application project.
2. From the **Web Forms** tab of the Toolbox, drag an **XML** control onto the page.

   A gray box representing the XML control is added to the top left corner of the page.

3. Select the control, and in the Properties window, set the following properties:

Property	Setting
DocumentSource	Emails.xml
TransformSource	Email_headers.xslt

   This will cause the **XML** control to display only the e-mail message headers by default.

Adding Controls to Change Transformations

In this walkthrough, you will use a check box to allow users to switch between transformations. By default, the **XML** control applies a transformation that displays only the e-mail message headers.

To add a check box to apply a different transformation

1. Select the **XML** control, press the BACKSPACE key, and press the ENTER key a few times to make space above the **XML** control.
2. From the **Web Forms** tab of the Toolbox, drag a **CheckBox** control onto the form above the **XML** control.
3. Set the following properties of the **CheckBox** control:

Property	Setting
Text	Headers only

Checked	**True**
AutoPostBack	**True**. This causes the form to be posted and processes as soon as the user clicks the box.

4. Double-click the check box.
5. The Code Editor opens with a skeleton handler for the **CheckBox** control's **CheckChanged** event.
6. Add code that switches transformations between **Email_headers.xslt** and **Email_all.xslt** depending on the state of the check box. The entire **CheckChanged** handler should look like the following:

```
' Visual Basic
Private Sub CheckBox1_CheckedChanged(ByVal sender As System.Object, ByVal e As System.Eve
ntArgs) Handles CheckBox1.CheckedChanged
   If CheckBox1.Checked Then
      Xml1.TransformSource = "email_headers.xslt"
   Else
       Xml1.TransformSource = "email_all.xslt"
   End If
End Sub

// C#
private void CheckBox1_CheckedChanged(object sender, System.EventArgs e)
{
   if (CheckBox1.Checked)
   {
      Xml1.TransformSource = "email_headers.xslt";
   }
   else
   {
      Xml1.TransformSource = "email_all.xslt";
   }
}
```

Testing

You can now see the transformations in action.

To test the transformations

1. Run the Web Forms page.

   By default, you will see the date, sender, and subject line of the e-mail messages.

2. Clear the **Headers only** check box.

   The e-mail messages are redisplayed, this time with a different layout and complete text.

Next Steps

This walkthrough has illustrated only the basics of how to work with an XML document and transformations. In a real application, you will often need to work with the XML document in more depth. The following are suggestions for further investigation:

- **Creating more sophisticated transformations** In this walkthrough, you have seen only one example of how you can use transformations. XSL is a powerful language, with sophisticated support not just for creating HTML pages, but also for virtually any kind of transformation from XML to another structure.For more details, see the Microsoft XSLT Developer's Guide.
- **Applying transformations programmatically** In this walkthrough, you took advantage of the **XML** Web server control to do the work of applying transformations to the raw XML data. You might find it useful in your application to do this work yourself (perhaps because it is impractical in your application to use an **XML** control). For details, see XSLT Transformations with the XslTransform Class.
- **Writing XML documents (instead of simply reading them)** The **XML** control makes it easy to display the contents of an XML file in a Web Forms page. However, you might want to create or amend XML files yourself. For details, see Employing XML in the .NET Framework.
- **Understanding security issues** Learn about the security issues associated with Web Forms. For details, see

Overview of Web Application Security Threats.

See Also

XML Web Server Control

Simple HTML Controls

This section covers controls that are available on the **HTML** tab of the Toolbox. These HTML controls can be added to HTML pages or to Web Forms pages.

By default, HTML elements on a Web Forms page are not available to the server; they are treated as opaque text that is passed through to the browser. However, by converting HTML elements to HTML server controls, you expose them as elements you can program in server-based code. For an overview of how HTML elements can be exposed to the server so that you can program them, see Introduction to ASP.NET Server Controls.

In This Section

HTML Div Controls
Contains links to information on the DIV element and gives related information on programming it.
HTML Horizontal Rule Control
Contains links to information on the HR element and gives related information on programming it.
HTML Image Control
Contains links to information on the IMG element and gives related information on programming it.
HTML Input Controls
Contains links to information on the various options for the INPUT element.
HTML Select Controls
Contains links to information on the SELECT element and gives related information on programming it.
HTML Table Control
Links to information on the TABLE element and gives related information on programming it.
HTML Text Area Control
Contains links to information on the TEXTAREA element and gives related information on programming it.

Related Sections

Introduction to ASP.NET Server Controls
Discusses how HTML elements can be exposed to the server so that you can program them.
Adding HTML Server Controls to a Web Forms Page
Gives basic steps for putting controls on a page.
ASP.NET Server Controls Recommendations
Compares HTML server controls with Web server controls.
HTML Designer
Provides reference information about the designer and links to related topics.
Navigating in the Document Outline Window
Shows how the Document Outline window relates to the HTML Designer.
HTML Elements
Gives a complete linked list of all the elements.

HTML Div Controls

Three controls based on the HTML DIV element are available on the HTML tab of the Toolbox:

- Flow Layout Panel control. Provides an area for flow (top to bottom) layout within a page configured for grid (x and y) layout.
- Grid Layout Panel control. Provides an area for grid layout (x and y) within a page configured for flow (top to bottom) layout.

  Note For details about flow and grid layout, see Positioning HTML Elements in Design View.

- Label control

  Note Dynamic HTML also includes an element named LABEL; it works only in Internet Explorer.

For a complete list of elements, see HTML Elements.

HTML controls added to an HTML page in Visual Studio are simply HTML elements with certain attributes already set.

The same is true for HTML controls added to a Web Forms (.aspx) page. By default, HTML elements on a Web Forms page are not available to the server; they are treated as opaque text that is passed through to the browser. However, by converting HTML elements to HTML server controls, you expose them as elements you can program in server-based code. For an overview of how HTML elements can be exposed to the server so that you can program them, see Introduction to ASP.NET Server Controls. For a list of these controls see HTML Server Controls.

See Also

Adding HTML Server Controls to a Web Forms Page | HTML Designer | Navigating in the Document Outline Window | HTML Elements

HTML Horizontal Rule Control

One control based on the HTML HR element is available on the HTML tab of the Toolbox as the Horizontal Rule control.

For a complete list of elements, see HTML Elements.

HTML controls added to an HTML page in Visual Studio are simply HTML elements with certain attributes already set.

The same is true for HTML controls added to a Web Forms (.aspx) page. By default, HTML elements on a Web Forms page are not available to the server; they are treated as opaque text that is passed through to the browser. However, by converting HTML elements to HTML server controls, you expose them as elements you can program in server-based code. For an overview of how HTML elements can be exposed to the server so that you can program them, see Introduction to ASP.NET Server Controls. For a list of these controls see HTML Server Controls.

See Also

Adding HTML Server Controls to a Web Forms Page | HTML Designer | Navigating in the Document Outline Window | HTML Elements

HTML Image Control

One control based on the HTML IMG element is available on the HTML tab of the Toolbox as the Image control.

For a complete list of elements, see HTML Elements.

HTML controls added to an HTML page in Visual Studio are simply HTML elements with certain attributes already set.

> **Security Note** The URL associated with an HTML Image control can be tampered with by a malicious user. For more information, see Scripting Exploits.

The same is true for HTML controls added to a Web Forms (.aspx) page. By default, HTML elements on a Web Forms page are not available to the server; they are treated as opaque text that is passed through to the browser. However, by converting HTML elements to HTML server controls, you expose them as elements you can program in server-based code. For an overview of how HTML elements can be exposed to the server so that you can program them, see Introduction to ASP.NET Server Controls. For a list of these controls see HTML Server Controls.

See Also

Adding HTML Server Controls to a Web Forms Page | HTML Designer | Navigating in the Document Outline Window | HTML Elements

HTML Input Controls

Several controls based on the HTML INPUT element are available on the HTML tab of the Toolbox:

- Button control: INPUT type=button Element
- CheckBox control: INPUT type=checkbox Element
- File Field control: INPUT type=file Element
- Hidden control: INPUT type=hidden Element
- Password control: INPUT type=password Element
- Radio Button control: INPUT type=radio Element
- Reset Button control: INPUT type=reset Element
- Submit Button control: INPUT type=submit Element
- Text Field control: INPUT type=text Element

For a complete list of elements, see HTML Elements.

> **Security Note** User input in a Web Forms page can include potentially malicious client script. By default, the Web Forms page validates that user input does not include script or HTML elements. For more information, see Scripting Exploits and Protecting Against Script Exploits in a Web Application.

HTML controls added to an HTML page in Visual Studio are simply HTML elements with certain attributes already set.

The same is true for HTML controls added to a Web Forms (.aspx) page. By default, HTML elements on a Web Forms page are not available to the server; they are treated as opaque text that is passed through to the browser. However, by converting HTML elements to HTML server controls, you expose them as elements you can program in server-based code. For an overview of how HTML elements can be exposed to the server so that you can program them, see Introduction to ASP.NET Server Controls. For a list of these controls see HTML Server Controls.

See Also

Adding HTML Server Controls to a Web Forms Page | HTML Designer | Navigating in the Document Outline Window | HTML Elements

HTML Select Controls

Two controls based on the HTML SELECT element are available on the HTML tab of the Toolbox:

- Dropdown control
- Listbox control

For a complete list of elements, see HTML Elements.

HTML controls added to an HTML page in Visual Studio are simply HTML elements with certain attributes already set.

> **Security Note** Controls in a Web Forms page can include potentially malicious client script. By default, the Web Forms page validates that user input does not include script or HTML elements. For more information, see Scripting Exploits and Protecting Against Script Exploits in a Web Application.

The same is true for HTML controls added to a Web Forms (.aspx) page. By default, HTML elements on a Web Forms page are not available to the server; they are treated as opaque text that is passed through to the browser. However, by converting HTML elements to HTML server controls, you expose them as elements you can program in server-based code. For an overview of how HTML elements can be exposed to the server so that you can program them, see Introduction to ASP.NET Server Controls. For a list of these controls see HTML Server Controls.

See Also

Adding HTML Server Controls to a Web Forms Page | HTML Designer | Navigating in the Document Outline Window | HTML Elements

HTML Table Control

One control based on the HTML TABLE element is available on the HTML tab of the Toolbox as the Table control.

When you first add a Table control to an HTML page or a Web Forms page, it measures three columns by three rows. You can manipulate it by using commands on the Table menu.

For a complete list of elements, see HTML Elements.

HTML controls added to an HTML page in Visual Studio are simply HTML elements with certain attributes already set.

The same is true for HTML controls added to a Web Forms (.aspx) page. By default, HTML elements on a Web Forms page are not available to the server; they are treated as opaque text that is passed through to the browser. However, by converting HTML elements to HTML server controls, you expose them as elements you can program in server-based code. For an overview of how HTML elements can be exposed to the server so that you can program them, see Introduction to ASP.NET Server Controls. For a list of these controls see HTML Server Controls.

See Also

Adding HTML Server Controls to a Web Forms Page | HTML Designer | Navigating in the Document Outline Window | HTML Elements

HTML Text Area Control

One control based on the HTML TEXTAREA element is available on the HTML tab of the Toolbox as the Text Area control.

For a complete list of elements, see HTML Elements.

HTML controls added to an HTML page in Visual Studio are simply HTML elements with certain attributes already set.

> **Security Note** User input in a Web Forms page can include potentially malicious client script. By default, the Web Forms page validates that user input does not include script or HTML elements. For more information, see Scripting Exploits and Protecting Against Script Exploits in a Web Application.

The same is true for HTML controls added to a Web Forms (.aspx) page. By default, HTML elements on a Web Forms page are not available to the server; they are treated as opaque text that is passed through to the browser. However, by converting HTML elements to HTML server controls, you expose them as elements you can program in server-based code. For an overview of how HTML elements can be exposed to the server so that you can program them, see Introduction to ASP.NET Server Controls. For a list of these controls see HTML Server Controls.

See Also

Adding HTML Server Controls to a Web Forms Page | HTML Designer | Navigating in the Document Outline Window | HTML Elements

HTML TITLE Element

Contains the title of the HTML document.

The TITLE element can be used only within the HEAD element. Any text between the opening and closing TITLE tags appears in the title bar of the browser.

For full details on this HTML element, see TITLE Element.

Directives for Web Forms Pages

Directives specify settings used by the page and user-control compilers when they process ASP.NET Web Forms page (.aspx) and user control (.ascx) files.

ASP.NET treats any directive block (<%@ %>) that does not contain an explicit directive name as an **@ Page** directive (for a page) or as an **@ Control** directive (for a user control).

For syntax information and descriptions of the attributes available for each directive, use the links that follow.

@ Page
: Defines page-specific attributes used by the ASP.NET page parser and compiler. Can only be included in .aspx files.

@ Control
: Defines control-specific attributes used by the ASP.NET page parser and compiler. Can only be included in .ascx files (user controls).

@ Assembly
: Declaratively links an assembly to the current page or user control.

@ Implements
: Declaratively indicates that a page or user control implements a specified .NET Framework interface.

@ Import
: Explicitly imports a namespace into a page or user control.

@ OutputCache
: Declaratively controls the output caching policies of a page or user control.

@ Reference
: Declaratively links a page or user control to the current page or user control.

@ Register
: Associates aliases with namespaces and class names, which allows user controls and custom server controls to be rendered when included in a requested page or user control.

See Also

New ASP.NET Page Directives | Application Directives

Web User Controls and Web Custom Controls

You can supplement the server controls that are provided with ASP.NET by defining your own controls, either as Web user controls or as Web custom controls. Each type of user-defined control has its advantages.

In This Section

Introduction to Web User Controls
Provides background information on Web user controls and how to work with them.
Creating Web User Controls
Outlines how to add a Web user control to a Web project.
Adding User Controls to a Web Forms Page
Provides details on the HTML tags needed to add Web user controls to Web Forms pages.
Introduction to Web Custom Controls
Provides background information on Web custom controls and when to use them.
Recommendations for Web User Controls vs. Web Custom Controls
Compares user controls to Web custom controls and presents guidelines for choosing between them.
Walkthrough: Creating a Web User Control
Presents step-by-step instructions on creating a user control for an ASP.NET Web application.
Walkthrough: Converting a Web Forms Page to a User Control
Presents step-by-step instructions for converting a regular Web Forms page into a Web user control.
Walkthrough: Creating a Web Custom Control
Presents step-by-step instructions for creating custom controls for use in an ASP.NET Web application.

Related Sections

Web Forms Pages
Presents a high-level overview of Web Forms features in Visual Studio and provides links for more information.
Developing ASP.NET Server Controls
Contains background information and examples to help you develop ASP.NET server controls.
Introduction to ASP.NET Web Applications in Visual Studio
Provides an overview of the components of an ASP.NET Web application, including Web Forms and XML Web services, and how Visual Studio helps you create a Web application.
Web Walkthroughs
Provides links to several walkthroughs that give step-by-step details of some common uses for Web Forms pages.

Introduction to Web User Controls

The ASP.NET server controls provide a great deal of functionality, but they cannot cover every situation. Web user controls enable you to easily define controls as you need them for your applications, using the same programming techniques that you use to write Web Forms pages. You can even convert a Web Forms page into a Web user control with a few modifications. (For details see Walkthrough: Converting a Web Forms Page to a User Control.) To make sure that a user control cannot be run as a standalone Web Forms page, user controls are identified by the file name extension .ascx.

> **Note** Web user controls are not to be confused with Web custom controls. For more information, see Recommendations for Web User Controls vs. Web Custom Controls.

User Control Structure

A Web user control is similar to a complete Web Forms page, with both a user interface page and a code-behind file. The user interface page differs from an .aspx file in these ways:

- The extension must be .ascx.
- The user control does not have <HTML>, <BODY>, and <FORM> elements in it (these elements must be in the hosting page).

In every other way, a user control is like a Web Forms page. You can use the same HTML elements and Web controls on a user control that you do on a standard Web Forms page. For example, if you are creating a user control to use as a toolbar, you can put a series of Button Web server controls onto the control and create event handlers for the buttons.

See Also

Creating Web User Controls | Adding User Controls to a Web Forms Page | Introduction to Web Custom Controls | Recommendations for Web User Controls vs. Web Custom Controls | Walkthrough: Creating a Web User Control

Creating Web User Controls

You design Web user controls in much the same way that you design Web Forms pages. You can use the same HTML elements and Web controls on a user control that you do on a standard Web Forms page. However, the user control does not have <HTML>, <BODY>, and <FORM> elements in it, and the file name extension must be .ascx.

To create a Web Forms user control

1. Create a Web project. For details, see Creating Web Projects.
2. On the **Project** menu, click **Add Web User Control**. Change the name as desired and click **Open** to open the control in the designer.
3. Add text and controls to the design surface. Any controls that you want to be able to access programmatically must be Web Forms server controls or HTML server controls.
4. Use the Web Forms Designer to set properties and create any code required for your control.

The following example shows a simple user control that you might use as a menu. The four menu choices are implemented as Hyperlink Web server controls.

```
<!-- Visual Basic -->
<%@ Control Language="vb" AutoEventWireup="false"
    Codebehind="menu.ascx.vb" Inherits="myProj.menu"
    TargetSchema="http://schemas.microsoft.com/intellisense/ie5"%>
<P>
<asp:HyperLink id="lnkLogin" runat="server"
    NavigateURL="Login.aspx">Login</asp:HyperLink>
  |
<asp:HyperLink id="lnkAddToCart" runat="server"
   NavigateURL="Cart.aspx">Add to Cart</asp:HyperLink>
  |
<asp:HyperLink id="lnkTechSupport" runat="server"
   NavigateURL="TechSupport.aspx">Technical Support</asp:HyperLink>
  |
<asp:HyperLink id="lnkAbout" runat="server"
   NavigateURL="AboutUs.aspx">About Us</asp:HyperLink>
</P>

<!-- C# -->
<%@ Control Language="c#" AutoEventWireup="false"
    Codebehind="menu.ascx.cs" Inherits="myProj.menu"
    TargetSchema="http://schemas.microsoft.com/intellisense/ie5"%>
<P>
<asp:HyperLink id="lnkLogin" runat="server"
   NavigateURL="Login.aspx">Login</asp:HyperLink>
  |
<asp:HyperLink id="lnkAddToCart" runat="server"
   NavigateURL="Cart.aspx">Add to Cart</asp:HyperLink>
  |
<asp:HyperLink id="lnkTechSupport" runat="server"
   NavigateURL="TechSupport.aspx">Technical Support</asp:HyperLink>
  |
<asp:HyperLink id="lnkAbout" runat="server"
   NavigateURL="AboutUs.aspx">About Us</asp:HyperLink></P>
```

See Also

Introduction to Web User Controls | Adding User Controls to a Web Forms Page | Introduction to Web Custom Controls | Recommendations for Web User Controls vs. Web Custom Controls | Walkthrough: Creating a Web User Control

Adding User Controls to a Web Forms Page

You can add a Web user control to a Web Forms page in Design view by simply dragging the control from Solution Explorer and dropping it where you want it to appear in the page. The Web Forms Designer automatically adds an @ Register directive and a tag for the control to the page. From that point, the control becomes part of the page and is rendered when the page is processed. Also, the control's public properties, events, and methods are exposed to the page and can be worked with programmatically. You can also add user controls to a page programmatically.

> **Note** The user control must be in the same project as the Web Forms page.

To add a user control to a Web Forms page

1. In the Web Forms Designer, open the Web Forms page you want to add the control to, and make sure that the page is in Design View.
2. Select the user control's file in Solution Explorer, and drag it onto the page.

To add a user control to a Web Forms page in HTML view

1. In the Web Forms Designer, open the Web Forms page you want to add the control to, and then switch to HTML view.
2. At the top of the page, before the <HTML> tag, add a directive that registers the control so it will be recognized when the page is processed. You use the directive to associate a name and a namespace with the Web user control by specifying **TagPrefix**, **TagName**, and **Src** location values. For example:

   ```
   <%@ Register TagPrefix="uc1" TagName="menu" Src="menu.ascx" %>
   ```

   Put the directive in its own line. If there are no other directives, make it the first line in the file.

   The values for each attribute are listed in the following table.

Attribute	Description
TagPrefix	The TagPrefix determines a unique namespace for the user control, so that if multiple user controls on the page happen to have the same name, they can be differentiated from each other. This will be the prefix to the control's name (such as <myNameSpace:xxx>) in the tag.
TagName	The TagName is the name for the user control. This name is used in conjunction with the tag prefix to uniquely identify the namespace for your control, as in the following prefix:tagname element: <myNameSpace:myUserControl ... />
Src	The Src attribute is the virtual path to the user control, for example "UserControl1.ascx" or "/MyApp/Include/UserControl1.ascx".

3. In the <BODY> portion of the file, create a tag for your control where you want the control to appear. Use the TagPrefix and TagName you registered in Step 2. Give your control an ID and set the attribute runat=server, as in the following example:

   ```
   <uc1:menu id="Menu1" runat="server"/>
   ```

4. If your control has properties that you can set at design time, optionally set them by declaring their values in the tag:

   ```
   <uc1:menu id="Menu1" runat="server" enabled=true />
   ```

5. Proceed with designing the rest of your Web Forms page. You can switch to Design view to work with your page. The user control will be displayed using a glyph to indicate its location in the page, but will not display a WYSIWYG rendering in the designer. To edit the control, switch back to HTML view.

Working with User Controls Programmatically

The public properties, methods, and events of a user control that is registered in a page are exposed to the parent Web Forms page. By simply adding a declaration for the user control to the code-behind class for the Web Forms page, you can work with the complete object model of the user control to your page's code.

To set a property of a user control in a Web Forms page

1. Select the user control file in Solution Explorer and drag it onto your page. Note the ID of the user control displayed in the

page.

2. Press F7 to switch from Design view to the code-behind file.
3. In the Declarations area, add a line to declare the user control. For example, for a user control of type WebUserControl1 whose ID is myControl1:

```
' Visual Basic
Public Class MyPage
   Inherits System.Web.UI.Page
   Protected Menu1 As menu

// C#
public class MyPage : System.Web.UI.Page
{
   protected menu Menu1;
```

Note The ID in the code-behind declaration must exactly match the ID for the user control in Design view.

Now that you have a code declaration for the user control, all of the public properties, methods, and events of your user control are available, and will appear in the IntelliSense statement completion dropdowns.

4. Write code to call methods or set properties of your user control. For example:

```
' Visual Basic
Menu1.Visible = True
Menu1.DataBind()

// C#
Menu1.Visible = true;
Menu1.DataBind();
```

For more information, see Developing ASP.NET Server Controls and ASP.NET Server Control Event Model.

Adding User Controls to a Web Forms Page Programmatically

To add Web user controls to a Web Forms page, you must perform some steps that you do not normally need when adding existing Web Forms server controls. For details, see Creating Instances of User Controls Programmatically.

See Also

Introduction to Web User Controls | Creating Web User Controls | Introduction to Web Custom Controls | Recommendations for Web User Controls vs. Web Custom Controls | Adding Controls to a Web Forms Page Programmatically | Walkthrough: Creating a Web User Control

Introduction to Web Custom Controls

Web custom controls are compiled components that run on the server and that encapsulate user-interface and other related functionality into reusable packages. They can include all the design-time features of standard ASP.NET server controls, including full support for Visual Studio design features such as the Properties window, the visual designer, and the Toolbox.

> **Note** Web custom controls are not to be confused with Web user controls. For more information, see Recommendations for Web User Controls vs. Web Custom Controls.

There are several ways that you can create Web custom controls:

- You can compile a control that combines the functionality of two or more existing controls. For example, if you need a control that encapsulates a button and a text box, you can create it by compiling the existing controls together. For more information, see Developing a Composite Control.
- If an existing server control almost meets your requirements but lacks some required features, you can customize the control by deriving from it and overriding its properties, methods, and events. For more information, see Developing ASP.NET Server Controls and Web Server Controls Hierarchy.
- If none of the existing Web server controls (or their combinations) meet your requirements, you can create a custom control by deriving from one of the base control classes. These classes provide all the basic functionality of Web server controls, so you can focus on programming the features you need. For more information, see Developing Custom Controls: Key Concepts and Developing a Simple ASP.NET Server Control.

> **Security Note** At design time in Visual Studio, your code always runs fully trusted, even if the code will eventually be in a project where it would receive less-than-full trust at run time. This means that your custom control might work properly when you are testing it on your own computer, but might fail due to lack of adequate permissions in a deployed application. Be sure to test your controls in the security context in which they will run in real-world applications. For more information, see Code Access Security.

See Also

Introduction to Web User Controls | Recommendations for Web User Controls vs. Web Custom Controls

Recommendations for Web User Controls vs. Web Custom Controls

If none of the existing ASP.NET server controls meet the specific requirements of your applications, you can create either a Web user control or a Web custom control that encapsulates the functionality you need. The main difference between the two controls lies in ease of creation vs. ease of use at design time.

Web user controls are easy to make, but they can be less convenient to use in advanced scenarios. You develop Web user controls almost exactly the same way that you develop Web Forms pages. Like Web Forms, user controls can be created in the visual designer, they can be written with code separated from the HTML, and they can handle execution events. However, because Web user controls are compiled dynamically at run time they cannot be added to the Toolbox, and they are represented by a simple placeholder glyph when added to a page. This makes Web user controls harder to use if you are accustomed to full Visual Studio .NET design-time support, including the Properties window and Design view previews. Also, the only way to share the user control between applications is to put a separate copy in each application, which takes more maintenance if you make changes to the control.

Web custom controls are compiled code, which makes them easier to use but more difficult to create; Web custom controls must be authored in code. Once you have created the control, however, you can add it to the Toolbox and display it in a visual designer with full Properties window support and all the other design-time features of ASP.NET server controls. In addition, you can install a single copy of the Web custom control in the global assembly cache and share it between applications, which makes maintenance easier. For more information see global assembly cache.

If your control has a lot of static layout, a user control might make sense. If your control is mostly dynamically generated — for instance rows of a data-bound table, nodes of a tree view, or tabs of a tab control — a custom control would be a better choice.

The main differences between the two types are outlined in this table:

Web user controls	**Web custom controls**
Easier to create	Harder to create
Limited support for consumers who use a visual design tool	Full visual design tool support for consumers
A separate copy of the control is required in each application	Only a single copy of the control is required, in the global assembly cache
Cannot be added to the Toolbox in Visual Studio	Can be added to the Toolbox in Visual Studio
Good for static layout	Good for dynamic layout

See Also

Introduction to Web User Controls | Walkthrough: Creating a Web Custom Control | Walkthrough: Creating a Web User Control

Walkthrough: Creating a Web User Control

Web user controls are controls that you can define as you need them for your applications, using the same programming techniques that you use to write Web Forms pages. You can create Web user controls when you need to create reusable pieces of UI that will be used throughout your application. A user control can contain HTML, server controls, and event-handling logic. Like a compiled control, it can expose properties that the host Web Forms page can set.

> **Note** Web user controls are not to be confused with Web custom controls. For more information, see Recommendations for Web User Controls vs. Web Custom Controls.

The steps to create a user control are very similar to those for creating a Web Forms page. You design the UI visually by arranging ASP.NET server controls, HTML, and static text on the design surface, binding to data, and writing code to handle events raised by the controls. Like a Web Forms page, a user control can handle page processing events, such as **Page_Load**.

This walkthrough creates a menu selection control that passes the selected item back to the page in the URL query string. This style of navigation is useful when your application uses a single page to display a particular kind of data, but you also want to provide a filtered view of the data. For example, you might have a Sales page showing data that can be filtered by region. A user can navigate directly to the data he wishes to see by clicking a button that puts the filter in the URL as a query-string element, as in this example that limits the information displayed to sales in the northeast region:

```
http://sales/salesreport.aspx?region=northeast
```

In this walkthrough, you will create a control that displays information categories as a menu on a Web Forms page. It gets the categories and populates the menu dynamically from a data source on the page, in this case a string array. When a menu item is clicked, that menu item's index number in the array is added as a parameter to a URL. That URL is passed to the host page, which takes the parameter and displays the information that is associated with that index number in a Label control; in this case, the information is simply the category name from the same string array.

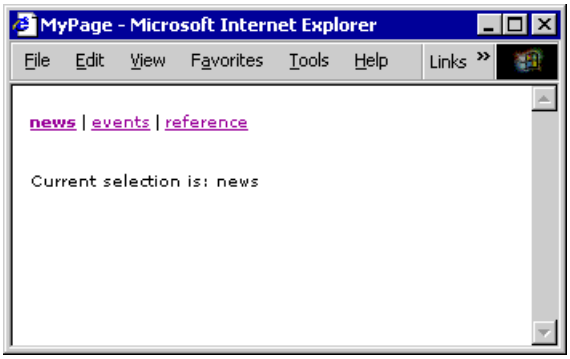
You will do the following steps to create the user control:

- Open a new user control in the designer
- Set a databinding expression for the user control
- Expose the data source and selection as properties to a Web Forms page
- Initialize the user control

You will also do the following to use the control on a Web Forms page:

- Add the user control to a page
- Create a data source for the user control
- Set the user control's properties
- Provide a filter presentation in the Web Forms page, based on the user control's selection
- Run the Web Forms page to test the control

When you finish, your page should look something like this:



Creating the User Control

The first step is to start a new project and add a Web user control to it.

To create the Web user control

1. On the **File** menu, point to **New**, and then click **Project**.

   The **New Project** dialog box appears.

2. In the **Project Types** pane, choose **Visual Basic Projects** or **Visual C# Projects**. In the **Templates** pane, select **ASP.NET Web Application** and click **OK**.

   A new project will be created with the default name at the default location.

3. Close the newly-created .aspx file. You won't need it in this walkthrough.
4. On the **Project** menu, click **Add Web User Control**.

   The **Add New Item** dialog box appears, with **Web User Control** selected.

5. Type **menu.ascx** in the **Name** box, and click **Open**. The user control opens in the designer.

To display the data in your user control, add a **DataList** control to the design surface. The **DataList** control takes each item from a data source, formats it and the entire data list according to the properties you specify, and displays the list of items. Because the list is dynamically generated based on the data source, this control is easily reusable.

To add the DataList control and set its properties

1. From the **Web Forms** tab of the Toolbox, add a **DataList** control to the menu.ascx page.

   This control will populate the menu by displaying a list of available options, which it will get from an array that is created in the host Web Forms page. To make the items appear on the page in the menu format "Item1 | Item2 | Item3", the **DataList** control display properties must be set correctly.

2. Make sure the **DataList** control is selected, and click **Property Builder** at the bottom of the Properties window to open the **DataList1 Properties** dialog box.
3. On the **General** page, in the **Repeat layout** group, set **Direction** to **Horizontal**.
4. On the **Format** page, in the **Objects** list, expand the **Items** node and select **Selected Items**. In the **Appearance** group, select **Bold**, and then click **OK**.

   This setting will make the menu item that is selected appear in bold letters.

5. Right-click the **DataList** control to show the shortcut menu, and then point to **Edit Template** and click **Separator Template**.
6. Position the insertion point in the blank row of the **Separator** template and type the "|" character.

   This character will be used to separate the menu items when they are listed.

Each data item that appears in the **DataList** menu will have an array index value, with a lower bound of 0. You will use this number to request the information associated with that menu item, by adding it as a parameter to the URL that is passed to the host page. To generate the URL, format the data items as hyperlinks using the **HyperLink** control, and bind that control to the data source. You can then design a binding expression that dynamically generates the appropriate request string for each hyperlink.

To add a HyperLink control and set the binding expression

1. Right-click the **DataList** control, and then point to **Edit Template** and click **Item Templates**. Several more template fields appear in the **DataList** control.

   The user needs to be able to click a menu item to retrieve the data related to that menu item. Therefore, each item needs to be a hyperlink.

2. From the **Web Forms** tab of the Toolbox, drag a **HyperLink** control to the blank row in the **DataList** control beneath the heading **ItemTemplate**.
3. Make sure that the **HyperLink** control is selected, and then select **(DataBindings)** in the Properties window and click the ellipsis button in the value field.

   The **DataBindings** dialog box appears. The **Text** property is selected by default in the **Bindable Properties** list.

4. In the **Binding for Text** group, under **Simple binding**, expand the **Container** node and select **DataItem**.

   This will bind the **Text** property of each hyperlink to the data item of the container, which in this case is the **DataList** control. The **DataList** control will receive its data items from the array that is created in the host Web Forms page.

5. In the **Bindable Properties** list, select the **NavigateUrl** property.

Note When you click the **NavigateUrl** property, the icon next to the **Text** property changes to show that it is data-bound.

6. In the **Binding for NavigateUrl** group, under **Simple binding**, select **ItemIndex**.

   This is the index number of the data item in the array. To append this number correctly to the URL so that the URL retrieves the data item at that index number, you must format it as a parameter. Name the parameter "category".

   Security Note The URL associated with a hyperlink can be tampered with by a malicious user. For more information, see Scripting Exploits.

7. In the **Format** box, enter this string (note the curly braces around the 0):

   ```
   ?category={0}
   ```

   Now you must complete the URL so that it points to the host Web Forms page.

8. Click **Custom binding expression** and add this code to the beginning of your almost-complete databinding expression:

   ```
   Request.Path +
   ```

   Your complete custom databinding expression should be:

   ```
   Request.Path + DataBinder.Eval(Container, "ItemIndex", "?category={0}")
   ```

9. Click **OK** to close the DataBindings dialog box.

The databinding expression that you entered contains the syntax for building a URL at run time, based on information from the link associated with the array number.

When the user control runs, this expression is evaluated as follows:

- First, **Request.Path** is replaced by the name of the current page. This approach lets you use your menu control on any page, regardless of what it is named. For more information, see HttpRequest.Path Property.
- Next, the **DataBinder.Eval** method takes the item's index number, and substitutes it for the "{0}" token in the format string. For more information, see DataBinder.Eval Method.

For example, if the hosting page is called MyPage.aspx, and the index for this item is 1, the generated URL would be:

```
MyPage.aspx?category=1
```

Exposing Properties to the Web Forms Page

You can bind a **DataList** control to a number of different types of data sources. For the sake of simplicity, this walkthrough binds the **DataList** control to a simple string array, which is passed from the Web Forms page to the user control via a property exposed by the user control.

To declare the array and selection properties

1. Press **F7** to switch from Design view to the code-behind file.
2. Add a public, class-level declaration for a string array, and name it `values`.

   This array will hold the list of data items.

3. Similarly, declare `selection` as **Integer** and initialize it to -1.

   This variable will hold the index value for the menu item that the user selects. Since -1 is not an index value in the array, the variable will not be assigned an index value on initialization.

   The top section of your code should look something like this:

   ```
   ' Visual Basic
   Public MustInherit Class menu
   ```

```
   Inherits System.Web.UI.UserControl
   Protected WithEvents DataList1 As System.Web.UI.WebControls.DataList
   Public values() As String
   Public selection As Integer = -1

// C#
public abstract class menu : System.Web.UI.UserControl
{
   protected System.Web.UI.WebControls.DataList DataList1;
   public string[] values;
   public int selection = -1;
```

Initializing the Control

Now that the pathways and formats are established, you need to add the code to initialize the connection between the control and the data. Use the page load event to do this initialization.

1. In the **Page_Load** procedure, set `values` to be the **DataSource** property for the **DataList** control.
2. Set **selection** to be the **SelectedIndex** property for the **DataList** control.
3. Call the **DataBind()** method of the **DataList** control to create the DataList items.

   Your **Page_Load** procedure will look like this:

```
' Visual Basic
Private Sub Page_Load(ByVal sender As System.Object, _
      ByVal e As System.EventArgs) Handles MyBase.Load
   DataList1.DataSource = values
   DataList1.SelectedIndex = selection
   DataList1.DataBind()
End Sub

// C#
private void Page_Load(object sender, System.EventArgs e)
{
   DataList1.DataSource = values;
   DataList1.SelectedIndex = selection;
   DataList1.DataBind();
}
```

4. Save and close the control.

Consuming the User Control

Now that you have created a user control, you will actually use it in a Web Forms page. It is possible to use your control on multiple pages and bind it to different data on each page. In this walkthrough, however, you consume the user control on a single page called MyPage.aspx.

To add the user control to a Web Forms page

1. On the **Project** menu, click **Add Web Form**.

   The **Add New Item** dialog box appears with **Web Form** selected.

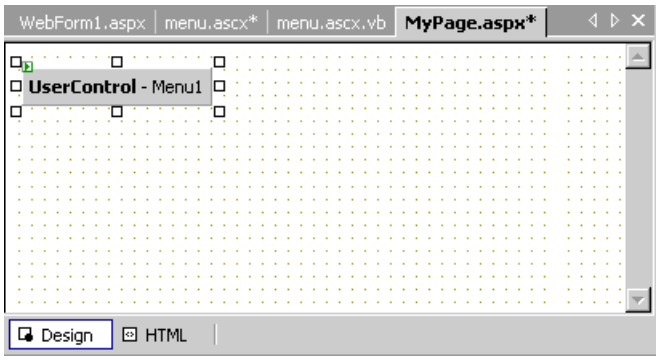
2. Change the **Name** to **MyPage.aspx**, and click **Open**.

   The new Web Forms page opens in Design view.

3. Select the menu.ascx file in Solution Explorer and drag it onto your page. It will appear as a user control, with the ID `Menu1`.

   The user control will be displayed on the page as a tag showing its type and name; there is no graphical display for user controls within a page since this type of control is not fully compiled until you run the page.

   User Control Glyph on a Web Forms Page



4. From the **Web Forms** tab of the Toolbox, add a **Label** control to the page.

   This control is used to display the information associated with the selected menu item.

5. Save your files.

Since Visual Studio does not automatically add a declaration of user controls to the code, the next step is to add the declaration manually.

To add a code-behind declaration for the user control

1. Press **F7** to switch from Design view to the code-behind file.
2. In the Declarations area, add a line to declare the user control **Menu1** as menu.

```
' Visual Basic
Public Class MyPage
   Inherits System.Web.UI.Page
   Protected WithEvents Label1 As System.Web.UI.WebControls.Label
   Protected Menu1 As menu

// C#
public class MyPage : System.Web.UI.Page
{
   protected System.Web.UI.WebControls.Label Label1;
   protected menu Menu1;
```

Now that you have a code declaration for the user control, you must set its properties programmatically.

To create a data source for the user control

1. In the **Page_Load** method of the Web Forms page, declare a string array and name it **values**. This is the array that will be called by the user control that you created. Populate the array with three items: "News", "Events", and "Reference".
2. Assign the array to the `values` property of the `Menu1` user control.

   Your code should look like this:

```
' Visual Basic
' set datasource for menu
Dim values() As String = New String() {"News", "Events", "Reference"}
Menu1.values = values

// C#
//set datasource for menu
String[] values = new String[]{"News", "Events", "Reference"};
Menu1.values = values;
```

To find out which item in the menu is selected, check the "category" parameter that you set when you created the custom binding expression for the **Hyperlink** control.

To capture the user control selection

1. In the **Page_Load** method, add a line of code to copy the current selection (if any) from the URL query string:

```
' Visual Basic
Dim selectionId As String = Request.Params("category")

// C#
String selectionId = Request.Params["category"];
```

Security Note Information passed in a query string can be tampered with by a malicious user. Do not rely on query strings to convey important or sensitive data. For more information, see Scripting Exploits.

2. If the selection was set, pass it on to the user control via the control's `selection` property and set the text of the **Label** control to indicate the current selection. The Web Forms page can use the selection information to filter the view it displays.

```
' Visual Basic
If (selectionId <> "") Then
   Dim SelectIndex As Integer = CInt(selectionId)
   Menu1.selection = SelectIndex
   Label1.Text = "Current selection is: " & values(SelectIndex)
End If

// C#
if(selectionId != "")
   {
      int SelectIndex = Convert.ToInt16(selectionId);
      Menu1.selection = SelectIndex;
      Label1.Text = "Current selection is: " + values[SelectIndex];
   }
```

That completes the Web Forms page. Your entire **Page_Load** procedure should look like this:

```
' Visual Basic
Dim values() As String = New String() {"news", "events", "reference"}
Menu1.values = values
Dim selectionId As String = Request.Params("category")
If (selectionId <> "") Then
   Dim SelectIndex As Integer = CInt(selectionId)
   Menu1.selection = SelectIndex
   Label1.Text = "Current selection is: " & values(SelectIndex)
End If

// C#
String[] values = new String[]{"News", "Events", "Reference"};
Menu1.values = values;
String selectionId = Request.Params["category"];
if(selectionId != "")
   {
      int SelectIndex = Convert.ToInt16(selectionId);
      Menu1.selection = SelectIndex;
      Label1.Text = "Current selection is: " + values[SelectIndex];
   }
```

3. Save and close the page.

Testing the Web Forms Page

You can now test your page to see the functionality of the menu.

To test the Web Forms page

1. Right-click MyPage.aspx in Solution Explorer and choose **Set As Start Page** on the shortcut menu.

2. Press F5 to run MyPage.aspx and test the user control. Click each of the menu selections to test their functionality.

The user control displays each string that is passed from the array in the Web Forms page as a separate link. You set the **DataSource** property of the **DataList** control to the array and use the **DataList** to render a hyperlink for each of the items in the **DataSource** property. The **DataList** control also takes care of applying a different visual style to the selected link.

See Also

Developing ASP.NET Server Controls | Web Control Library Template |
Recommendations for Web User Controls vs. Web Custom Controls | Introduction to Web User Controls |
Walkthrough: Converting a Web Forms Page to a User Control | Overview of Web Application Security Threats

Walkthrough: Converting a Web Forms Page to a User Control

If you have developed a Web Forms page and decide you would like to access its functionality throughout your application, you can make some minor alterations to the file to change it to a user control. Web user controls are very similar to Web Forms pages, and they are created using the same techniques. When you convert a Web Forms page to a Web user control, you create a reusable UI component that you can use on other Web Forms pages.

For this walkthrough, you will create a basic Web Forms page that presents a personalized welcome message to the user. To convert the page to a Web user control, you will make a few changes to the code:

- Change the code-behind base class from **Page** to **UserControl**
- Delete the `<html>`, `<head>`, `<body>`, and `<form>` tags from the .aspx file
- Change the ASP.NET directive type from **@ Page** to **@ Control**
- Change the **Codebehind** attribute to refer to the control's code-behind class file (ascx.vb or ascx cs)
- Change the .aspx file extension to .ascx

To test the control, you will create a new Web Forms page, add the control to it, and open the page in a browser.

Creating the Web Forms Page

The first step is to create a Web application and a Web Forms page.

To create the project and form

1. On the **File** menu, point to **New**, then click **Project**.
2. In the **New Project** dialog box, do the following:
   a. In the **Project Types** pane, choose either **Visual Basic Projects** or **Visual C# Projects**.
   b. In the **Templates** pane, choose **ASP.NET Web Application**.
   c. In the **Location** box, enter the complete URL for your application, including http://, and the name of the server. The last part of the URL is the name of your project; for this walkthrough, name your project **Conversion**. The Web server must have IIS version 5 or later and the .NET Framework installed on it. If you have IIS on your computer, you can specify `http://localhost` for the server. (If you normally use a proxy server to access the Internet, you might need to configure Internet Explorer to bypass the proxy server in order to use localhost.)

      Tip If you already have a solution open, choose **Close Solution** so that your new Web Forms project will be part of its own solution.

   When you click **OK**, a new Web Forms project is created at the root of the Web server you specified. In addition, a new Web Forms page called WebForm1.aspx is displayed in the Web Forms Designer in Design view.

You can now add a few controls and write some code to display a personalized welcome message.

To add controls and program them

1. From the **Web Forms** tab of the Toolbox, drag a **TextBox** control onto the designer.
2. Drag a **Label** control to the left of the text box. In the **Properties** window, change the **Label** control's **Text** property to **Enter Name:**.
3. Drag a **Button** control underneath the text box, and change the **Text** property to **Enter**.
4. Drag a second **Label** control underneath the button, and delete the default text so that the **Text** property is empty.
5. Double-click the **Button** control to open the **Code** view with a **Button_Click** event handler added.
6. Add the following code to the **Button1_Click** procedure:

```
' Visual Basic
Label2.Text = "Hi " & TextBox1.Text & ", welcome to ASP.NET!"

// C#
Label2.Text = "Hi " + TextBox1.Text + ", welcome to ASP.NET!";
```

   Do not close this file, because you will return to it later in the walkthrough.

   Security Note User input in a Web Forms page can include potentially malicious client script. By default, the

Web Forms page validates that user input does not include script or HTML elements. For more information, see Scripting Exploits and Protecting Against Script Exploits in a Web Application.

7. Press CTRL+F5 to run the Web Forms page and test it. When you are finished, close the browser that opened when you tested the page.

Now you have a basic Web Forms page that you can convert to a Web user control.

Converting the Page to a User Control

Since the `<html>`, `<body>`, and `<form>` elements are not included in user controls, you must remove these elements and include them in the parent Web Forms page instead. You must also change the ASP.NET directive type in the Web Forms page from @ Page to @ Control. These ASP.NET directives specify settings used by the page and user control compilers. For more information, see Directive Syntax.

To convert the Web Forms page to a user control

1. Return to WebForm1.aspx in the designer, and switch to HTML view by clicking the HTML tab at the bottom of the designer window.
2. Delete the `<html>` tags, `<!doctype>` tag, `<head>` tags and inner content, `<body>`, and `<form>` tags from the .aspx file. If you are using Visual Basic, the HTML looks like this:

```
<%@ Page Language="vb" AutoEventWireup="false"
      Codebehind="WebForm1.aspx.vb" Inherits="Conversion.WebForm1">
<asp:TextBox id="TextBox1" runat="server"></asp:TextBox>
<asp:Label id="Label1" runat="server">Enter Name:</asp:Label>
<asp:Button id="Button1" runat="server" Text="Enter"></asp:Button>
<asp:Label id="Label2" runat="server"></asp:Label>
```

If you are using C#, the HTML will look identical except for the first line, which will look like this:

```
<%@ Page Language="cs" AutoEventWireup="false"
      Codebehind="WebForm1.aspx.cs" Inherits="Conversion.WebForm1">
```

Note The Code Editor adds underlines that normally indicate problems with the code. These marks will disappear after the conversion is completed.

3. Change the ASP.NET directive type from @ **Page** to @ **Control**. In a Visual Basic project, it will look like this:

```
<%@ Control Language="vb" AutoEventWireup="false"
      Codebehind="WebForm1.aspx.vb" Inherits="Conversion.WebForm1">
```

In a C# project, it will look like this:

```
<%@ Control Language="cs" AutoEventWireup="false"
      Codebehind="WebForm1.aspx.cs" Inherits="Conversion.WebForm1">
```

4. Change the **Codebehind** attribute reference of the directive to reflect the fact that the .aspx extension will be changed to .ascx, which is the extension for user controls. You will also change the name of the control to welcome.ascx later in this walkthrough, so set the **Codebehind** attribute to **welcome.ascx.vb** or **welcome.ascx.cs** as appropriate.
5. Similarly, change the **Inherits** attribute of the directive to refer to **Conversion.welcome**.
6. Return to the **WebForm1.aspx.vb** or **WebForm1.aspx.cs** file.
7. Change the base class from **System.Web.UI.Page** to **System.Web.UI.UserControl**, and change the **Public Class** declaration from **WebForm1** to **welcome**.

```
' Visual Basic
Public Class welcome
   Inherits System.Web.UI.UserControl

// C#
```

```
public class welcome : System.Web.UI.UserControl
```

8. Save and close the code-behind file and the .aspx file.
9. Right-click **WebForm1.aspx** in Solution Explorer and click **Rename** on the shortcut menu.
10. Change the name of the file to **welcome.ascx** to reflect its new function. It is important to note that you must change the file name extension from .aspx to .ascx. When the confirmation dialog appears, click **OK**.

The file is now a Web user control, and it is ready to be consumed by Web Forms pages throughout your project.

Testing the User Control

You can test your user control by creating a new Web Forms page and adding the control to it.

To test the user control

1. On the **Project** menu, click **Add Web Form**. The **Add New Item** dialog appears with **Web Form** selected.
2. Change the **Name** to WelcomeTest.aspx and click **Open**. The new Web Forms page opens in the designer.
3. Drag welcome.ascx from Solution Explorer onto the design surface.
4. Right-click WelcomeTest.aspx in Solution Explorer and click **Set As Start Page** on the shortcut menu.
5. Press F5 to run the Web Forms page and test it.

See Also

Introduction to Web User Controls | Recommendations for Web User Controls vs. Web Custom Controls | Developing User Controls in a Code-Behind File | Overview of Web Application Security Threats

Walkthrough: Creating a Web Custom Control

If none of the existing ASP.NET server controls meet the requirements of your application, you can create a custom control by deriving from one of the base control classes. These classes provide all the basic functionality of server controls, so you can focus on programming the features you need.

In this walkthrough, you will use the code for a custom Label control that is included by default in the **Web Control Library** template. This control is derived from the WebControl class and behaves the same as a standard Label control with the added property of rendering the value of the Text property as a hyperlink. To complete the walkthrough, you will perform these procedures:

- Create two projects in the same solution — one for the custom Label control, and one for a Web Forms page to consume and test the control.
- Add the control to the Toolbox.
- Consume the custom control by adding it to the Web Forms page.
- Create a custom designer for the control.
- Test the control in the browser.

> **Note** The **Web Control Library** template is not available in the Standard Edition of Visual Basic and Visual C# .NET. For more information, see Visual Basic Standard Edition Features or Visual C# Standard Edition Features.

The emphasis in this walkthrough is not on writing code for the control, but rather on what to do to make a control once the code is written.

Creating the Projects

The first step is to start a new project with the custom control template and build the control.

To create the custom control

1. On the **File** menu, point to **New**, and then click **Project**.

   The **New Project** dialog box appears.

2. In the **Project Types** pane, choose either **Visual Basic Projects** or **Visual C# Projects**. Select **Web Control Library** in the **Templates** pane.
3. Change the **Name** to **CustomLabel** and click **OK**.

   The new project is created, and **WebCustomControl1** opens in the Code Editor.

   Code for a custom **Label** control is included in the **Web Control Library** template by default. This is the control you will use in this walkthrough.

4. On the **Build** menu, click **Build CustomLabel** to compile the control.

   The control is compiled as CustomLabel.dll. By default, it is created in the **Bin** folder inside the **CustomLabel** project folder.

5. Save the file.

Now that the control has been created, you will add a Web Forms page to the same solution so that the page is convenient for testing the control.

To create the Web Forms page

1. On the **File** menu, point to **Add Project**, and then click **New Project**.

   The **Add New Project** dialog box appears.

2. In the **Project Types** pane, choose either **Visual Basic Projects** or **Visual C# Projects**. Select **ASP.NET Web Application** in the **Templates** pane.
3. In the **Location** box, enter the complete URL for your application, including http://, and the name of the server. The last part of the URL is the name of your project; for this walkthrough, name your project **LabelWebForm**. The Web server must have IIS version 5 or later and the .NET Framework installed on it. If you have IIS on your computer, you can specify `http://localhost` for the server. (If you normally use a proxy server to access the Internet, you might need to configure Internet Explorer to bypass the proxy server in order to use localhost.)

4. Click **OK**.

   The new project is created, and **WebForm1** opens in the designer.

5. Save the project.

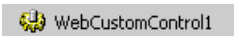
Adding the Control to the Toolbox

Now that your control has been compiled and you have a Web Forms page ready to test it, you can add the control to the Toolbox so it will be readily available for use.

To add your control to the Toolbox

1. On the **Tools** menu, click **Add/Remove Toolbox Items**.
2. On the **.NET Framework Components** tab of the **Customize ToolBox** dialog box, click the **Browse** button. Find **CustomLabel.dll**, select it, and click **Open** to add **WebCustomControl1** to the list of components in the **Customize Toolbox** dialog box.
3. Select **WebCustomControl1** in the list of .NET Framework components and click **OK**. **WebCustomControl1** is added to the Toolbox.

When you are finished, you should see an icon on the toolbar that looks like this:



The next step is to test the control in the page.

To add the control to the Web Forms page and test it

1. Open **WebForm1** in Design view and drag **WebCustomControl1** from the Toolbox to the page.

   The control's default rendering, which is simply the name of the control followed by its ID, appears in the Design view.

2. Switch to HTML view, and verify that an **@ Register** directive for the control's assembly is added to the page's HTML, with TagPrefix "cc1".
3. Set the **Text** property of the control to **Hello**.

   The control's appearance in Design view is updated to show the new text.

Next, you are going to customize the control, so you need to remove it from the Toolbox.

To remove the control from the Toolbox

1. Right-click **WebCustomControl1** in the Toolbox and click **Delete** on the shortcut menu. This action will not delete the control from the project, only from the Toolbox.
2. Close the Web Forms page without saving.

The custom Label control is now functional, but it's not very attractive and the tag is not distinctive in the code. The next step is to individualize the control.

Individualizing the Control

The control is using the default icon in the Toolbox and the default tag prefix "cc1". To make the control more easily identifiable, change the tag prefix and create a new icon.

To change the tag prefix

1. Under the **CustomLabel** project in Solution Explorer, double-click **AssemblyInfo** to open the AssemblyInfo file in the designer. Add an **Imports** statement (in Visual Basic) or **using** statement (in Visual C#) at the top of the AssemblyInfo page:

```
' Visual Basic
Imports System.Web.UI

// C#
using System.Web.UI;
```

2. Add an **Assembly: TagPrefix** attribute to the list of assembly attributes. This will specify the new **TagPrefix** for the **CustomLabel** control, in this case **xxx**.

```
' Visual Basic
<Assembly: TagPrefix("CustomLabel", "xxx")>

//C#
[assembly: TagPrefix("CustomLabel", "xxx")]
```

To create a new icon

1. Select the **CustomLabel** project in Solution Explorer. On the **Project** menu, click **Add New Item**.

   The **Add New Item** dialog box appears.

2. Select **Bitmap File** in the Templates pane, change the **Name** to **WebCustomControl1.bmp**, and click **Open**. The new bitmap file appears in Solution Explorer and opens in the designer.

   > **Note** For this walkthrough, the name of the icon file name must match that of the control class exactly (WebCustomControl1), including capitalization.

3. In the Properties window, change both the **Height** and **Width** properties of the bitmap to 16. Icons must be 16 by 16 pixels in size.
4. Draw a distinctive design on the bitmap, using several colors, so you can recognize the icon.
5. Save and close the bitmap file.
6. Select **WebCustomControl1.bmp** in Solution Explorer. In the Properties window, change the **Build Action** property to **Embedded Resource**.
7. Select the **CustomLabel** project in Solution Explorer. On the **Build** menu, click **Rebuild CustomLabel**.
8. Save all your work.

When you add the control to the Toolbox now, the icon you created will show up next to the control name. When you consume the control on your Web Forms page, the control's HTML tag will contain the identifying prefix that you specified.

To consume the custom control

1. Add **WebCustomControl1** to the Toolbox again.

   The control appears with the icon you created. Depending on your settings, it may appear under the **General** tab of the Toolbox.

2. Reopen the Web Forms page named **WebForm1** and drag the control onto it.

   > **Note** The **Text** property has reverted to the default, because you closed the Web Forms page without saving earlier.

3. Switch to HTML view and verify that the **TagPrefix** attribute of the **@ Register** directive is **xxx**.
4. Because you are going to make another modification to the control, delete **WebCustomControl1** from the Web Forms page, but leave it in the Toolbox.

Your custom control is now fully functional and individualized, but its appearance on the design surface is not very helpful. To make the control look better, you will add a custom designer to it.

Creating a Custom Designer

Designers are classes that allow you to modify the design-time appearance and behavior of components and controls. Although the usual goal of any WYSIWYG form designer is to minimize differences between design-time and run-time appearance, at times special design-time cues are necessary. For more information, see Custom Designers.

In this walkthrough, you will implement a custom designer that causes the custom control to render its text as a hyperlink.

To create a custom designer for the control

1. Select the **CustomLabel** project in Solution Explorer.
2. On the **Project** menu, click **Add Reference** and add a reference to **System.Design.dll**.
3. On the **Project** menu, click **Add New Item**.

   The **Add New Item** dialog box appears.

4. Select **Class** in the Templates pane, change the **Name** to **SampleDesigner**, and click **Open**. The new class file appears in Solution Explorer and opens in the designer.
5. Add code that overrides the base class's GetDesignTimeHtml method. This method is called at design time to get the representation of your control. In the overridden method, do the following:
   a. Get the **Text** property of the custom control.
   b. If the **Text** property has been set (that is, it's not an empty string), create a new **Hyperlink** control, set its properties to match the original control, and call the **Hyperlink** control's **Render** method.

The end result is that the **Hyperlink** control is displayed in the designer using the value of the **Text** property.

The code to perform these steps looks like the following, which is a complete listing of the **SampleDesigner** file:

```
' Visual Basic
Imports System
Imports System.IO
Imports System.Web
Imports System.Web.UI
Imports System.Web.UI.WebControls
Imports System.Web.UI.Design

Public Class SampleDesigner
    Inherits System.Web.UI.Design.ControlDesigner

    Public Overrides Function GetDesignTimeHtml() As String
        ' Component is the control instance, defined in the base
        ' designer
        Dim ctl As WebCustomControl1 = CType(Me.Component, _
            WebCustomControl1)
        If (ctl.Text <> "") Then
            Dim sw As New StringWriter()
            Dim tw As New HtmlTextWriter(sw)
            Dim placeholderlink As New HyperLink()

            ' put control text into the link's Text
            placeholderlink.Text = ctl.Text
            placeholderlink.NavigateUrl = ctl.Text
            placeholderlink.RenderControl(tw)

            Return sw.ToString()
        Else
            Return Me.GetEmptyDesignTimeHtml()
        End If
    End Function
End Class

// C#

using System;
using System.IO;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.Design;

namespace CustomLabel
{
   public class SampleDesigner : System.Web.UI.Design.ControlDesigner
   {
```

```
        public override string GetDesignTimeHtml()
        {
        // Component is the control instance, defined in the base
        // designer
        WebCustomControl1 ctl = (WebCustomControl1 ) Component;

        if (ctl.Text != "" && ctl.Text != null)
        {
           StringWriter sw = new StringWriter();
           HtmlTextWriter tw = new HtmlTextWriter(sw);

           HyperLink placeholderLink = new HyperLink();

           // put control text into the link's Text
           placeholderLink.Text = ctl.Text;
           placeholderLink.NavigateUrl = ctl.Text;
           placeholderLink.RenderControl(tw);

           return sw.ToString();
        }
        else
           return GetEmptyDesignTimeHtml();
        }
     }
  }
```

6. Open **WebCustomControl1.vb** or **WebCustomControl1.cs** (depending on the language you are using) in the designer, and add this attribute to the attribute list just before the WebCustomControl1 class declaration:

```
Designer("CustomLabel.SampleDesigner, CustomLabel")
```

The WebCustomControl1 class declaration should look like this:

```
' Visual Basic
<Designer("CustomLabel.SampleDesigner, CustomLabel"), _
DefaultProperty("Text"), ToolboxData("<{0}:WebCustomControl1 _
runat=server></{0}:WebCustomControl1>")> Public Class WebCustomControl1

// C#
[Designer("CustomLabel.SampleDesigner, CustomLabel"), DefaultProperty("Text"), ToolboxDat
a("<{0}:WebCustomControl1 runat=server></{0}:WebCustomControl1>")]
public class WebCustomControl1 : System.Web.UI.WebControls.WebControl
```

7. On the **Build** menu, click **Build Solution** so that both projects are built.
8. Save all, and close all the open files.

Testing the Control

To test that the Text property works correctly

1. Open **WebForm1** in the designer.
2. Drag **WebCustomControl1** from the Toolbox onto the page.
3. Set the **Text** property of your Web custom control to **Hello**. The text should appear as a hyperlink on the Web Forms page.
4. Save WebForm1.

To test the control in a browser

1. Select **WebForm1** in Solution Explorer. On the **File** menu, click **View in Browser**.
2. Verify that **Hello** shows up in the browser. The text looks like a label, not like a hyperlink.

Note At design time in Visual Studio, your code always runs fully trusted, even if the code will eventually be in a project where it would receive less than full trust at run time. This means that your custom control might work properly when you are testing it on your own computer, but might fail due to lack of adequate permissions in a deployed application. Be sure to test your controls in the security context in which they will run in real-world applications.

See Also

Developing ASP.NET Server Controls | Recommendations for Web User Controls vs. Web Custom Controls | Overview of Web Application Security Threats

XML Web Services in Managed Code

XML Web services provides the ability to exchange messages in a loosely coupled environment using standard protocols such as HTTP, XML, XSD, SOAP, and WSDL. The messages can be structured and typed or loosely defined. Because standard protocols serve as the basis for XML Web services, your XML Web service applications can communicate with a broad variety of implementations, platforms, and devices. The topics in this section focus on how to create and deploy XML Web services and how to access XML Web services in managed code. See the list below for links to other topics that discuss XML Web services.

In This Section

Introduction to Programming XML Web Services in Managed Code
Describes the XML Web services programming model in managed code.
XML Web Services Walkthroughs
Demonstrates creation and access of XML Web services using Visual C#, Visual Basic, ATL Server, and Managed Extensions for C++.
Creating XML Web Services in Managed Code
Describes the process of creating and deploying XML Web services.
Accessing XML Web Services in Managed Code
Describes the process of accessing XML Web services from a variety of managed code application types.

Related Sections

Programming the Web with XML Web Services
Provides a portal to all XML Web services documentation.
XML Web Services Created Using ASP.NET and XML Web Service Clients
Provides documentation from the .NET Framework SDK on XML Web services.
Walkthrough: Creating a Distributed Application
Demonstrates the creation and access of an XML Web service for returning and updating data using managed code.
Customizing SOAP Messages
Describes how to customize SOAP messages, including how to work with various XML encoding methods.
Using SOAP Headers
Describes how to utilize SOAP headers in XML Web services created using ASP.NET and how to include additional information with SOAP messages.
Debugging Preparation: ASP.NET Web Service Projects
Instructions on how to enable debugging in XML Web services and how to change the build configuration.
XML Web Services
Describes the use of XML Web services in Duwamish 7.0.
Decision Chart
Provides a graphical, linked guide to decisions about designing your application.

Introduction to Programming XML Web Services in Managed Code

XML Web services make possible the exchange of data in the form of XML messages between heterogeneous systems. Although remote access of data and application logic is not a new concept, doing so in a loosely coupled fashion is. Previous attempts, such as DCOM, IIOP, and Java/RMI, required tight integration between the client and the server and used platform and implementation specific binary data formats. While such protocols require a particular component technology or object-calling convention, XML Web services do not. The only assumption made between the client and the server is that recipients will understand the messages they receive. In other words, the client and server agree to a contract, in this case described using WSDL and XSD, and then communicate by generating messages that honor the contract over a specified transport like HTTP. As a result, programs written in any language, using any component model, and running on any operating system can access XML Web services. Furthermore, the flexibility of using a text format like XML makes possible the message exchange to evolve over time in a loosely coupled way. This loose coupling is mandatory in environments where the simultaneous updating of all parties in the message exchange is not possible.

You create XML Web services using the ASP.NET page framework, enabling these XML Web services to access the many features of the .NET Framework, such as authentication, caching, and state management. Because ASP.NET and the .NET Framework are the basis for XML Web services in managed code, developers can focus on creating or accessing XML Web services without needing to write infrastructure code.

In the ASP.NET application model, Web pages intended for the browser use the .aspx extension. To differentiate XML Web services from regular ASP.NET pages, XML Web services use the .asmx extension.

There are two fundamental roles when working with XML Web services:

- **Creating an XML Web service** — When you create an XML Web service, you are creating an application that exposes functionality to XML Web service clients.
- **Accessing an XML Web service** — When you access an XML Web service, your client application locates, references, and uses the functionality contained within a separate XML Web service. The client of an XML Web service is typically an application that is able to send, receive, and process messages to and from the XML Web service. However, the minimum requirement is that the client must be able to send messages to the XML Web service. This includes all applications using the .NET Framework.

XML Web services can be either stand-alone applications or sub-components of a larger Web application. For example, suppose you are creating a Web application that sells books online. Your Web application might interact with XML Web services as follows:

- **Creating an XML Web service** — Your application exposes its order processing logic as an XML Web service, which your affiliate Web sites can in turn use in their Web applications to sell books through your online store without requiring their customers to visit your site.
- **Accessing an XML Web service** — Your application accesses an XML Web service provided by another online company that specializes in writing and supplying book reviews for online booksellers. When a visitor to your online store views the details of a particular book, they also see the reviews of that book on the same page.

Remember, XML Web services are accessible from just about any other kind of application, including other XML Web services, Web applications, Windows applications, and console applications. The minimum requirement is that the client must be able to send messages to the XML Web service.

See Also

XML Web Services in Managed Code | Programming the Web with XML Web Services |
Creating XML Web Services in Managed Code | Accessing XML Web Services in Managed Code

XML Web Services Walkthroughs

Walkthroughs give step-by-step instructions for common scenarios, which makes them a good place to start learning about the product or a particular feature area.

The following walkthroughs will help orient you to the use of XML Web services in your Visual Basic and Visual C# applications:

Creating an XML Web Service Using Visual Basic or Visual C#
: Explains how to create an XML Web service that converts temperatures measured in Fahrenheit to Celsius.

Accessing an XML Web Service Using Visual Basic or Visual C#
: Describes how to make use of an existing XML Web service.

Calling XML Web Services from Windows Forms
: Describes how to make calls to Web Service methods from a Windows application.

Creating a Distributed Application
: Describes the process of making a basic multi-tiered, distributed application.

See Also

Walkthroughs in Visual Basic and Visual C# | Visual Studio Walkthroughs | Samples and Walkthroughs | Visual Basic Code Example Topics

Walkthrough: Calling XML Web Services from Windows Forms

A new aspect of Visual Studio is XML Web services, which provides the ability to exchange messages in a loosely coupled environment using standard protocols such as HTTP, XML, XSD, SOAP, and WSDL. The messages can be structured and typed or loosely defined. Because Web services are based on standard protocols, your Web service applications can communicate with a broad variety of implementations, platforms, and devices. For more information, see XML Web Services in Managed Code.

Web services can be used to enhance the functionality of Windows Forms. Connecting Windows Forms to Web services is as simple as making calls to Web service methods, which are processed on a server that then returns the results of the method call.

There are two types of Web service methods, synchronous and asynchronous. When synchronous Web service methods are called, the caller waits for the Web service to respond before continuing operations. When asynchronous Web service methods are called, you can continue to use the calling thread while waiting for the Web service to respond. This allows you to use the existing set of threads efficiently in the client application. For more information about working with synchronous and asynchronous Web service methods, see Accessing XML Web Services in Managed Code.

Synchronous Web Service Methods

A call to a synchronous Web service method involves calling the method and waiting for the computation to occur on the server and return a value before continuing with the rest of the code in the Windows Form.

To create an XML Web service

1. Create a Web service application. For more information, see Creating XML Web Services in Managed Code.
2. In Solution Explorer, right-click the .asmx file and choose **View Code**.
3. Create a Web service method that does addition. This following Web service method will take two integers and add them, returning the sum:

```
' Visual Basic
<WebMethod()> Public Function WebAdd(ByVal x As Integer, ByVal y As Integer) As Integer
   Return x + y
End Function

// C#
[WebMethod]
public int WebAdd(int x, int y)
{
   return x + y;
}
```

4. Create another Web service method that does multiplication. The following Web service method will take two integers and multiply them, returning the product:

```
' Visual Basic
<WebMethod()> Public Function WebMultiply(ByVal x As Integer, ByVal y As Integer) As Inte
ger
   Return x * y
End Function

// C#
[WebMethod]
public int WebMultiply(int x, int y)
{
   return x * y;
}
```

5. From the **Build** menu, choose **Build Solution**. You can also browse to the .asmx file you created in this project to learn more about Web services. Your Web service is now available for calling from a Windows Form.

To call an XML Web service synchronously

1. Create a new Windows application. For more information, see Creating a Windows Application Project.

   Security Note Calls to Web Methods require a privilege level granted by the System.Net.WebPermisson class. If you are running in a partial-trust context, the process might throw an exception. For more information, see Code Access Security Basics.

2. Add a reference to the Web service created above. For details, see Adding and Removing Web References.
3. From the Toolbox, add three **TextBox** controls and two **Button** controls. The text boxes will be for the numbers, and the buttons will be used for the calculations and to call the Web service methods.
4. Set the properties of the controls as follows:

Control	Property	Text
TextBox1	Text	0
TextBox2	Text	0
TextBox3	Text	0
Button1	Text	Add
Button2	Text	Multiply

5. Right-click the form and choose **View Code**.
6. Create an instance of the Web service as a member of the class. You need to know the name of the server where you created the Web service above.

```
' Visual Basic
' Replace localhost below with the name of the server where
' you created the Web service.
Dim MathServiceClass As New localhost.Service1()

// C#
localhost.Service1 MathServiceClass = new localhost.Service1();
```

7. Create an event handler for Button1's **Click** event. For details, see Creating Event Handlers on the Windows Forms Designer.

```
' Visual Basic
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Han
dles Button1.Click
' Create instances of the operands and result.
   Dim x, y, z As Integer
' Parse the contents of the text boxes into integers.
   x = Integer.Parse(TextBox1.Text)
   y = Integer.Parse(TextBox2.Text)
' Call the WebAdd Web service method from the instance of the Web service.
   z = MathServiceClass.WebAdd(x, y)
   TextBox3.Text = z.ToString
End Sub

// C#
private void button1_Click(object sender, System.EventArgs e)
{
// Create instances of the operands and result.
   int x, y, z;
// Parse the contents of the text boxes into integers.
   x = int.Parse(textBox1.Text);
   y = int.Parse(textBox2.Text);
// Call the WebAdd Web service method from the instance of the Web service.
   z = MathServiceClass.WebAdd(x, y);
   textBox3.Text = z.ToString();
}
```

Visual C# Note Be sure that the necessary code to enable the event handler is present. In this case, it would be similar to the following:

this.button1.Click += new System.EventHandler(this.button1_Click);

8. Create an event handler for Button2's **Click** event in the same fashion, and add the following code.

```
' Visual Basic
Private Sub Button2_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Han
dles Button2.Click
' Create instances of the operands and result.
   Dim x, y, z As Integer
' Parse the contents of the text boxes into integers.
   x = Integer.Parse(TextBox1.Text)
   y = Integer.Parse(TextBox2.Text)
' Call the WebMultiply Web service method from the instance of the Web service.
   z = MathServiceClass.WebMultiply(x, y)
   TextBox3.Text = z.ToString
End Sub

// C#
private void button2_Click(object sender, System.EventArgs e)
{
// Create instances of the operands and result.
   int x, y, z;
// Parse the contents of the text boxes into integers.
   x = int.Parse(textBox1.Text);
   y = int.Parse(textBox2.Text);
// Call the WebAdd Web service method from the instance of the Web service.
   z = MathServiceClass.WebMultiply(x, y);
   textBox3.Text = z.ToString();
}
```

Visual C# Note Be sure that the necessary code to enable the event handler is present. In this case, it would be similar to the following:

this.button2.Click += new System.EventHandler(this.button2_Click);

9. Press F5 to run your application. Enter values into the first two text boxes. When you press the **Add** button, the third text box should show their sum. When you press the **Multiply** button, the third text box should show their product.

Note The first call to a Web service takes a while for the server to process, because the Web service is instantiated on the server. Keep this in mind when pressing the buttons in your application. This lag is dealt with in the section below.

Asynchronous Web Services

When you call asynchronous Web service methods, the application continues to run while waiting for the Web service to respond. This allows you to use the resources efficiently in the client application. This is a far more resource-savvy way to implement Web services within your Windows application.

For details, see Accessing an XML Web Service Asynchronously in Managed Code.

See Also

Accessing XML Web Services in Managed Code | XML Web Services in Managed Code | Windows Forms and Web Forms Recommendations | Web References | Generating an XML Web Service Proxy | Windows Applications as Clients

Creating XML Web Services in Managed Code

You create XML Web services in managed code using the ASP.NET page framework. You start by creating an .asmx file. This file contains the **WebService** directive, which identifies the public class that implements the XML Web service logic. The XML Web service class contains one or more public methods for exposure in an XML Web service. These XML Web service methods are prefaced with the **WebMethod** attribute. ASP.NET handles the infrastructure to expose those methods as an XML Web service. For more information, see Code Model for XML Web Services in Managed Code.

To create XML Web services in managed code using Visual Studio, you need access to a Web server configured for developing ASP.NET applications. You develop XML Web services in managed code on a *development server*. By default, the development server is the machine on which you installed Visual Studio. Typically, you develop and build the project on a development server, and then deploy it to another server (the *deployment server*) that will host the XML Web service. However, if development occurs directly on the server that will host the XML Web service, the development server and deployment server are the same. For more information, see Creating Web Projects and Visual Studio .NET Software Requirements.

In This Section

ASP.NET Web Service Projects in Visual Studio
Describes the structure of an ASP.NET Web Service project.
Code Model for XML Web Services in Managed Code
Describes the code model used by Web Services in managed code.
Exploring XML Web Service Content
Describes the content of a Web Service that you can view from a Web browser.
Creating ASP.NET Web Service Projects
Provides instructions on how to create an XML Web service using the ASP.NET Web Service project template.
Adding an XML Web Service to an Existing Web Project in Managed Code
Provides instructions on how to add a Web Service to an existing Web project.
Using the WebService Attribute
Provides instructions on how to apply the System.Web.Services.WebServiceAttribute attribute.
Inheriting from the WebService Class
Provides instructions on how to inherit from the **System.Web.Services.WebService** class.
Creating an XML Web Service Method
Provides instructions on how to create a method exposed by a Web Service.
Using the WebMethod Attribute
Provides instructions on how to apply the **System.Web.Services.WebMethodAttribute** attribute.
Debugging XML Web Services in Managed Code
Provides instructions on how to debug a Web Service using multiple methods.
Deploying XML Web Services in Managed Code
Provides instructions on how to deploy a Web Service using multiple methods.
Fine-Tuning Discovery Mechanisms
Describes the reasons for modifying Web Service discovery documents created in Visual Studio.

Related Sections

Programming the Web with XML Web Services
Provides links to various XML Web services topics.
Introduction to Programming XML Web Services in Managed Code
Describes the Web Services programming paradigm.
Accessing XML Web Services in Managed Code
Describes the process and provides instructions on how to access Web Services in managed code.
XML Web Services
Describes the use of XML Web services in Duwamish 7.0.
Customizing SOAP Messages
Describes how to customize SOAP messages, including how to work with various XML encoding methods.
Using SOAP Headers
Describes how to utilize SOAP headers in XML Web services created using ASP.NET and how to include additional information with SOAP messages.
Building XML Web Services Using ASP.NET
Provides information from the .NET Framework SDK on building XML Web Services using ASP.NET.
Handling and Throwing Exceptions in XML Web Services
Describes how to handle and throw exceptions in XML Web services created using ASP.NET.

XML Web Services Created with ATL Server

Describes how to create, distribute, and consume XML Web services using ATL Server and C++.

Code Model for XML Web Services in Managed Code

XML Web services consist of two parts: the XML Web service entry point and the code that implements the XML Web service functionality. In ASP.NET, the .asmx file is a text file that serves as the addressable entry point for the XML Web service. It references code in pre-compiled assemblies, a code-behind file, or code contained in the .asmx file itself.

> **Note** If the code is contained in the .asmx file directly, then ASP.NET will compile it on demand on the server.

The **WebService** processing directive at the top of the .asmx file determines where to find the implementation of the XML Web service. By default, Visual Studio uses code-behind files, such as Service1.asmx.vb or Service1.asmx.cs, when you create an XML Web service with the ASP.NET Web Service project template.

> **Note** Visual Studio does not support cross-language compiling. For example, it is not possible to include a Visual C# XML Web service in a Visual Basic ASP.NET Web Application project, and vice versa.

When you build an XML Web service in managed code, ASP.NET automatically provides the infrastructure and handles the processing of XML Web service requests and responses, including the parsing and creation of SOAP messages.

WebService Processing Directive

At the top of the .asmx page is a **WebService** processing directive, which includes information in the form of attributes regarding the implementation of the XML Web service. This processing directive provides necessary information to the ASP.NET environment, such as which class implements the XML Web service functionality. An example of the **WebService** processing directive follows:

```
' Visual Basic
<%@ WebService Language="vb" Codebehind="Service1.asmx.vb"
    Class="WebService1.Service1" %>

// C#
<%@ WebService Language="c#" Codebehind="Service1.asmx.cs"
    Class="WebService1.Service1" %>
```

The **Language** attribute indicates the programming language used to develop the XML Web service. You can create XML Web services in any .NET-compatible language, such as Visual Basic .NET or Visual C#.

When creating XML Web services in managed code using Visual Studio .NET, the implementation of the XML Web service resides in a code-behind file. The code-behind file is associated with the .asmx page using the **Codebehind** attribute.

> **Note** This attribute helps Visual Studio to manage your XML Web service project and is not required at runtime.

When using the ASP.NET Web Service project template, the **Class** attribute indicates which class in the code-behind file implements the functionality of the XML Web service.

Visual Studio .NET inserts this processing directive automatically into the .asmx file when you use the ASP.NET Web Service project template.

> **Tip** If you choose to rename the class, be sure to change the class name in the **Class** attribute of the **WebService** directive.

> **Note** To view the contents of the .asmx file, in **Solution Explorer**, right-click the .asmx file and click **Open With** on the shortcut menu. In the **Open With** dialog box, select **Source Code (Text) Editor** and then click **Open**.

For more information, see Declaring an XML Web Service.

System.Web.Services.WebService Class

The System.Web.Services.WebService class, which defines the optional base class for XML Web services, provides direct access to common ASP.NET objects, such as those for application and session state. By default, XML Web services created in managed code using Visual Studio inherit from this class. The XML Web service can inherit from this class to gain access to ASP.NET's intrinsic objects, such as Request and Session. For more information, see WebService Class.

If the XML Web service does not inherit from this class, it can access the ASP.NET intrinsic objects from **System.Web.HttpContext.Current**. The class implementing the XML Web service must be public and must have a public default

constructor (a constructor without parameters). This makes it possible for ASP.NET to create an instance of the XML Web service class to process incoming XML Web service requests. For more information, see HttpContext.Current Property.

```
' Visual Basic
Imports System.Web.Services
Public Class Service1
   Inherits System.Web.Services.WebService
   ' Implementation code.
End Class

// C#
using System.Web.Services;
public class Service1 : System.Web.Services.WebService
{
   // Implementation code.
}
```

For more information, see Inheriting from the WebService Class.

WebService Attribute

Each XML Web service requires a unique namespace, which makes it possible for client applications to differentiate among XML Web services that might use the same method name. The default namespace for XML Web services created in Visual Studio .NET is "`http://tempuri.org/WebService1/Service1`" where `WebService1` is the project name and `Service1` is the class name. Although the namespace resembles a typical URL, you should not assume that it is viewable in a Web browser, it is merely a unique identifier.

> **Note** You may want to provide a Web page at that location that contains information about XML Web services you provide.

Using the **WebService** attribute, you can specify the namespace and provide a brief description for the XML Web service. The brief description will appear on the Service help page when you call the XML Web service from a browser without specifying a query string:

```
' Visual Basic
<System.Web.Services.WebService( _
   Namespace:="http://tempuri.org/WebService1/Service1", _
   Description:="A short description of the XML Web service.")> _
Public Class Service1
   Inherits System.Web.Services.WebService
   ' Implementation code.
End Class

// C#
[System.Web.Services.WebService(
   Namespace="http://tempuri.org/WebService1/Service1",
   Description="A short description of the XML Web service.")]
public class Service1 : System.Web.Services.WebService
{
   // Implementation code.
}
```

For more information, see WebServiceAttribute Class and Using the WebService Attribute.

Applying the WebMethod Attribute

To expose a method as part of an XML Web service, you must place a **WebMethod** attribute before the declaration of each public method you want to expose. For more information, see Creating an XML Web Service Method.

```
' Visual Basic
<System.Web.Services.WebMethod()> _
Public Function MyString(ByVal x as string) As String
   ' Implementation code.
End Function
```

```
// C#
[System.Web.Services.WebMethod()]
public string MyString(string x)
{
   //implementation code
}
```

The WebMethod attribute contains several properties for configuring the behavior of the XML Web service. For more information, see WebMethodAttribute Class and Using the WebMethod Attribute. For example, you can use this attribute to provide a brief description that will appear on the associated Service help page:

```
' Visual Basic
<System.Web.Services.WebMethod( _
   Description:="A short description of this method.")> _
Public Function MyString(ByVal x as string) As String
   ' Implementation code.
End Function

// C#
[System.Web.Services.WebMethod(
   Description="A short description of this method.")]
public string MyString(string x)
{
   // Implementation code.
}
```

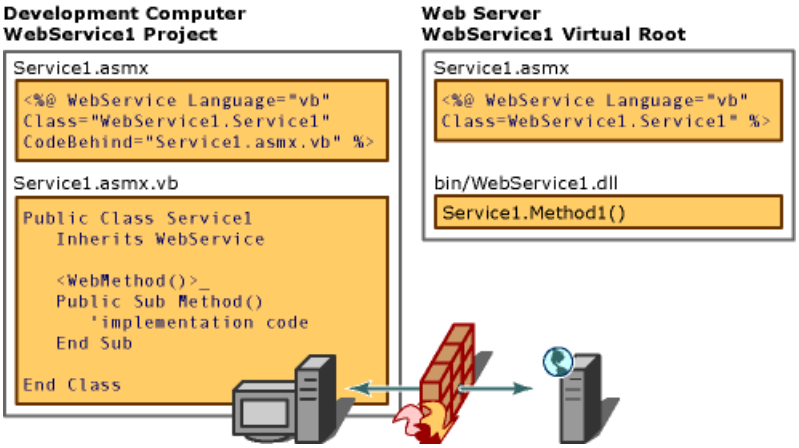
Separate multiple properties with a comma. For example, to provide a description and to cache the results of an XML Web service for 60 seconds:

```
' Visual Basic
<System.Web.Services.WebMethod( _
   Description:="A short description of this method.", _
           CacheDuration:=60)> _
Public Function MyString(ByVal x as string) As String
   ' Implementation code.
End Function

// C#
[System.Web.Services.WebMethod(
   Description="A short description of this method.",
   CacheDuration=60)]
public string MyString(string x)
{
   // Implementation code.
}
```

Summary

Each XML Web service consists of an .asmx file and an XML Web service class. The .asmx file contains a **WebService** processing directive that references the class. The XML Web service class is public, has a public default constructor, and contains one or more public methods marked with the **WebMethod** attribute. The following diagram shows the relationship between the project, the class, its methods, and the resulting XML Web service.

WebService Classes and Resulting Items



See Also

Creating XML Web Services in Managed Code | Code Model for Accessing XML Web Services in Managed Code

Exploring XML Web Service Content

The .asmx file represents the addressable entry point for XML Web services created with managed code. How you access this file through HTTP determines the type of response received.

Service Help Page

When called from a Web browser without supplying a recognized query string, the .asmx file returns an automatically generated Service help page for the XML Web service. This is the equivalent of doing an HTTP GET on the .asmx file.

This Service method help page provides a list of the methods the XML Web service provides and that you can access programmatically. This page contains links for each method that will take you to additional information about that method. In addition, this page contains a link to the XML Web service description document.

To access the Service help page

- In your browser's Address field, type the base URL for the XML Web service by using this format:

```
http://servername/projectname/xmlwebservicename.asmx
```

Parameter	Value
servername	The server on which the XML Web service resides.
projectname	The name of the project for your XML Web service and any additional directory information needed to access the .asmx file for your XML Web service.
webservicename.asmx	The name of the .asmx file for the XML Web service.

For example, to access an XML Web service named Service1.asmx that is part of a project named WebService1 on your local machine, you would enter the following:

```
http://localhost/XmlWebService1/Service1.asmx
```

Service Method Help Page

The Service method Help page provides additional information that relates to that particular XML Web service method.

The page provides the ability to invoke the method using the HTTP-POST protocol. However, the XML Web service method must support the HTTP-POST protocol to enable this functionality.

At the bottom of the page, the Service method help page provides sample request and response messages for the protocols the XML Web service method supports.

Service Description

The Service Help page also provides a link to the XML Web service's service description, which is a formal definition of the XML Web service's capabilities. The service description is a document that uses the Web Services Description Language (WSDL) grammar. The service description defines the contract for the message formats clients need to follow when exchanging messages with the XML Web service. For more information, see XML Web Service Description.

To access the service description

- In your browser's Address field, type the base URL for the XML Web service and provide the WSDL query string by using this format:

```
http://servername/projectname/xmlwebservicename.asmx?WSDL
```

Parameter	Value
servername	The server on which the XML Web service resides.
projectname	The name of the project for your XML Web service and any additional directory information needed to access the .asmx file for your XML Web service.

webservicenam e.asmx	The name of the .asmx file for the XML Web service.

For example, to access the service description of an XML Web service named Service1.asmx that is part of a project named WebService1 on your local machine, you would enter the following:

```
http://localhost/XmlWebService1/Service1.asmx?WSDL
```

Accessing XML Web Services from a Web Browser

You can access an XML Web service that supports the HTTP-GET protocol from a Web browser. By default, XML Web services that you create in Visual Studio by using the ASP.NET Web Service project template support HTTP-GET, HTTP-POST, and HTTP-SOAP. Not all protocols can support all methods. In general, the HTTP-GET and HTTP-POST protocols are more restricted than HTTP-SOAP in the data types they can support.

To access an XML Web service from a Web Browser

- In your browser's Address field, type the base URL for the XML Web service with the method name and parameters in the query string by using the following format:

```
http://servername/projectname/xmlwebservicename.asmx/methodname?parametername0=value0&par
ametername1=value1&...&parameternamen=valuen
```

Parameter	Value
servername	The server on which the XML Web service resides.
projectname	The name of the project for your XML Web service and any additional directory information needed to acce ss the .asmx file for your XML Web service.
webservicenam e.asmx	The name of the .asmx file for the XML Web service.
methodname	The case-sensitive name of the Web method.
parameternam e	The names of the Web method parameters.
value	The URL encoded values of each of the Web method parameters.

For example, to access the **ReturnThisString** method of an XML Web service named **Service1.asmx** with a parameter named **MyString** set to a value of "This is my test string" and that is part of a project named **WebService1** on your local machine, you would enter the following:

```
http://localhost/XmlWebService1/Service1.asmx/ReturnThisString?MyString=This+is+my+test+s
tring
```

See Also

Creating XML Web Services in Managed Code | XML Web Service Description

Creating ASP.NET Web Service Projects

You can start a project that includes the necessary files for creating an XML Web service in either Visual Basic or Visual C# by using the project templates provided by Visual Studio.

To create an XML Web service project

1. On the **File** menu, point to **New**, and then click **Project**.
2. In the **New Project** dialog box, select either the **Visual Basic Projects** or the **Visual C# Projects** folder.
3. Click the **ASP.NET Web Service** icon.
4. Enter the address of the Web server on which you will develop the XML Web service. For example, use "http://MyServer/Application" or simply "http://MyServer" to develop in the root directory.

   > **Note** For some project types, the Name textbox is unavailable because specifying the location sets the project name instead. For example, Web applications and XML Web services are located on a Web server and derive their name from the virtual directory specified on that server.

5. Click **OK** to create the project.

Visual Studio automatically creates the necessary files and references to support an XML Web service. When completed, the IDE displays the .asmx file in Design view. For more information, see ASP.NET Web Service Projects in Visual Studio.

See Also

Creating XML Web Services in Managed Code | Working with Web Projects

Adding an XML Web Service to an Existing Web Project in Managed Code

You can add an XML Web service to an existing Web project in Visual Studio.

To add a new XML Web service to an existing Web project

1. In **Solution Explorer**, select the project to which you want to add an XML Web service.
2. On the **Project** menu, click **Add Web Service**. Alternatively, right-click the project name, point to **Add** and then click **Add Web Service**.
3. The **Add New Item** dialog appears. By default, the dialog box selects the **ASP.NET Web Service** template and supplies a name for the new XML Web service in the **Name** box.
4. Click **Open** to add the XML Web service to your project.

Visual Studio automatically creates the necessary files and references to support an XML Web service. When completed, the IDE displays the .asmx file in Design view. For more information, see ASP.NET Web Service Projects in Visual Studio.

To add an existing XML Web service to an existing Web project

You can also add an existing XML Web service to an existing Web project in Visual Studio. This process is limited to XML Web services written in the same programming language as your existing Web project. For more information, see Adding Existing Items to a Project.

1. In **Solution Explorer**, select the project to which you want to add an existing XML Web service.
2. On the **Project** menu, select **Add Existing Item**. Alternatively, right-click the project name, point to **Add** and then click **Add Existing Item**. The **Add Existing Item** dialog box appears.
3. In the **Files of type** list, click **Web Files**.
4. Select the .asmx file for the XML Web service you want to add.
5. Click **Open** to add the XML Web service to your project.

Visual Studio adds the .asmx file and its associated code-behind file to your project. For more information, see ASP.NET Web Service Projects in Visual Studio.

See Also

Creating XML Web Services in Managed Code | Working with Web Projects | Adding New Project Items

Using the WebService Attribute

You can use the **WebService** attribute to specify the namespace, which defaults to "`http://tempuri.org`", and description text for the XML Web service. By default the ASP.NET Web Service project template does not generate a class that includes this attribute. Separate multiple properties with a comma.

For more information, see Declaring an XML Web Service.

To apply the WebService attribute

- Insert the **WebService** attribute before the class declaration and set the **Namespace** and **Description** properties as shown below:

```
' Visual Basic
<System.Web.Services.WebService( _
   Namespace:="http://servername/xmlwebservices/", _
   Description:="Some descriptive text could go here.")> _
Public Class Service1
    Inherits System.Web.Services.WebService
    ' Implementation code.
End Class

// C#
[System.Web.Services.WebService(
   Namespace="http://servername/xmlwebservices/",
   Description="Some descriptive text could go here.")]
public class Service1 : System.Web.Services.WebService
{
    // Implementation code.
}
```

See Also

Creating XML Web Services in Managed Code | WebServiceAttribute Class

Inheriting from the WebService Class

By default, XML Web services created by using the ASP.NET Web Service project template inherit from the **System.Web.Services.WebSerivce** class. Inheriting this class makes it possible for the XML Web service to access the ASP.NET intrinsic objects, such as Application and Session. For more information, see Inheritance (Visual Basic .NET) or 10.2.1 Inheritance (C#)

> **Note** If your class does not inherit from the **WebService** class, the Component Designer for the XML Web service will not be available.

If the XML Web service does not inherit from this class, it can access the ASP.NET intrinsic objects from **System.Web.HttpContext.Current**. This can be the case when your XML Web service needs to inherit from a custom base class. The class implementing the XML Web service must be public and must have a public default constructor (a constructor without parameters). This makes it possible for ASP.NET to create an instance of the XML Web service class to process incoming XML Web service requests. For more information, see HttpContext.Current Property.

To inherit from the System.Web.Services.WebService class

- You can inherit from the **System.Web.Services.WebService** class as shown in the example below:

```
' Visual Basic
Public Class Service1
    Inherits System.Web.Services.WebService
    <System.Web.Services.WebMethod( _
       Description:="Get SessionID", _
       EnableSession:=True)> _
    Public Function GetSessionID() As String
        GetSessionID = Me.Session.SessionID
    End Function
End Class

// C#
public class Service1 : System.Web.Services.WebService
{
    [System.Web.Services.WebMethod(
       Description="Get SessionID",
       EnableSession=true)]
    public string GetSessionID()
    {
        return this.Session.SessionID;
    }
}
```

See Also

Creating XML Web Services in Managed Code | WebService Class | HttpContext.Current Property

Creating an XML Web Service Method

When you create an XML Web service in managed code, you indicate the methods that are available through that XML Web service by placing the **WebMethod** attribute before the method declaration of a **Public** method. **Private** methods cannot serve as the entry point for an XML Web service although they can be in the same class and the XML Web service code can call them. The **WebMethod** attribute must be applied to each public method that is available as part of the XML Web service. For more information, see Defining XML Web Service Methods.

The procedures below assume that you are working with public methods of a class that is the implementation of an XML Web service. For more information, see Code Model for XML Web Services in Managed Code.

To create an XML Web service method

1. Select your .asmx file (for example, **Service1.asmx)** in **Solution Explorer**. On the **View** menu, click **Code** to view the code-behind file. Alternatively, right-click the .asmx file in **Solution Explorer** and then click **View Code** on the shortcut menu.

   > **Note** **Solution Explorer** hides code-behind files by default. Click **Show All Files** in the **Solution Explorer** toolbar to see hidden files. Alternatively, on the **Project** menu, click **Show All Files**.

   The .asmx file specifies the class that is the implementation of the XML Web service in the **Class** property of the .asmx file's **WebService** processing directive. Following is an example:

   ```
   ' Visual Basic
   <%@ WebService Language="vb" Codebehind="Service1.asmx.vb"
       Class="WebService1.Service1" %>

   // C#
   <%@ WebService Language="c#" Codebehind="Service1.asmx.cs"
       Class="WebService1.Service1" %>
   ```

   If you rename the class, be sure to change the class name in the **Class** property of the **WebService** directive as well.

2. Add a public method to the class specified in the **Class** property of the .asmx file's **WebService** processing directive as shown below:

   ```
   ' Visual Basic
   Public Class Service1
       Inherits System.Web.Services.WebService
       <System.Web.Services.WebMethod()> _
       Public Function ConvertTemperature(ByVal dFahrenheit As Double) _
                                          As Double
           ConvertTemperature = ((dFahrenheit - 32) * 5) / 9
       End Function
   End Class

   // C#
   public class Service1 : System.Web.Services.WebService
   {
       [System.Web.Services.WebMethod()]
       public double ConvertTemperature(double dFahrenheit)
       {
          return ((dFahrenheit - 32) * 5) / 9;
       }
   }
   ```

   Attaching the **WebMethod** attribute to a **Public** method indicates that you want the method exposed as part of the XML Web service. For more information, see Code Model for XML Web Services in Managed Code.

See Also

Using the WebMethod Attribute

Attaching the **WebMethod** attribute to a **Public** method indicates that you want the method exposed as part of the XML Web service. You can also use the properties of this attribute to further configure the behavior of the XML Web service method. For more information, see Code Model for XML Web Services in Managed Code.

The WebMethod attribute provides the following properties:

- BufferResponse
- CacheDuration
- Description
- EnableSession
- MessageName
- TransactionOption

BufferResponse

The **BufferResponse** property of the **WebMethod** attribute enables buffering of responses for an XML Web service method. When set to **true**, the default setting, ASP.NET buffers the entire response before sending it down to the client. The buffering is very efficient and helps improve performance by minimizing communication between the worker process and the IIS process. When set to **false**, ASP.NET buffers the response in chunks of 16KB. Typically, you would set this property to **false** only if you did not want the entire contents of the response in memory at once. For example, you are writing back a collection that is streaming its items out of a database. Unless otherwise specified, the default value is **true**. For more information, see WebMethodAttribute.BufferResponse Property.

To buffer the response of an XML Web service method

- Use the **BufferResponse** property of the **WebMethod** attribute, as shown below:

```
' Visual Basic
Public Class Service1
    Inherits System.Web.Services.WebService
    <System.Web.Services.WebMethod(BufferResponse:=False)> _
    Public Function GetBigData() As DataSet
        'implementation code
    End Function
End Class

// C#
public class Service1 : System.Web.Services.WebService
{
    [System.Web.Services.WebMethod(BufferResponse=false)]
    public DataSet GetBigData()
    {
       //implementation code
    }
}
```

CacheDuration

The **CacheDuration** property of the **WebMethod** attribute enables caching of the results for an XML Web service method. ASP.NET will cache the results for each unique parameter set. The value of this property specifies how many seconds ASP.NET should cache the results. A value of zero disables the caching of results. Unless otherwise specified, the default value is zero. For more information, see WebMethodAttribute.CacheDuration Property.

To cache the results of an XML Web service method

- Use the **CacheDuration** property of the **WebMethod** attribute, as shown below:

```
' Visual Basic
Public Class Service1
```

```
    Inherits System.Web.Services.WebService
    <System.Web.Services.WebMethod(CacheDuration:=60)> _
    Public Function ConvertTemperature(ByVal dFahrenheit As Double) _
                                       As Double
        ConvertTemperature = ((dFahrenheit - 32) * 5) / 9
    End Function
End Class

// C#
public class Service1 : System.Web.Services.WebService
{
    [System.Web.Services.WebMethod(CacheDuration=60)]
    public double ConvertTemperature(double dFahrenheit)
    {
       return ((dFahrenheit - 32) * 5) / 9;
    }
}
```

Description

The **Description** property of the **WebMethod** attribute supplies a description for an XML Web service method that will appear on the Service help page. Unless otherwise specified, the default value is an empty string. For more information, see WebMethodAttribute.Description Property.

To supply a description for an XML Web service method

- Use the **Description** property of the **WebMethod** attribute, as shown below:

```
' Visual Basic
Public Class Service1
    Inherits System.Web.Services.WebService
    <System.Web.Services.WebMethod( _
       Description:="This method converts a temperature " & _
       "in degrees Fahrenheit to a temperature in degrees Celsius.")> _
    Public Function ConvertTemperature(ByVal dFahrenheit As Double) _
                                       As Double
        ConvertTemperature = ((dFahrenheit - 32) * 5) / 9
    End Function
End Class

// C#
public class Service1 : System.Web.Services.WebService
{
    [System.Web.Services.WebMethod(
       Description="Converts F to C a temperature in " +
       "degrees Fahrenheit to a temperature in degrees Celsius.")]
    public double ConvertTemperature(double dFahrenheit)
    {
       return ((dFahrenheit - 32) * 5) / 9;
    }
}
```

EnableSession

The **EnableSession** property of the **WebMethod** attribute enables session state for an XML Web service method. Once enabled, the XML Web service can access the session state collection directly from **HttpContext.Current.Session** or with the **WebService.Session** property if it inherits from the **WebService** base class. Unless otherwise specified, the default value is **false**. For more information, see WebMethodAttribute.EnableSession Property.

To enable session state in an XML Web service method

- Use the **EnableSession** property of the **WebMethod** attribute, as shown below:

```
' Visual Basic
Public Class Service1
    Inherits System.Web.Services.WebService
    <System.Web.Services.WebMethod(EnableSession:=True)> _
    Public Function ConvertTemperature(ByVal dFahrenheit As Double) _
                                       As Double
        Session("Conversions") = Session("Conversions") + 1
        ConvertTemperature = ((dFahrenheit - 32) * 5) / 9
    End Function
    <System.Web.Services.WebMethod(EnableSession:=True)> _
    Public Function GetNumberOfConversions() As Integer
        GetNumberOfConversions = Session("Conversions")
    End Function
End Class

// C#
public class Service1 : System.Web.Services.WebService
{
    [System.Web.Services.WebMethod(EnableSession=true)]
    public double ConvertTemperature(double dFahrenheit)
    {
       Session["Conversions"] = (int) Session["Conversions"] + 1;
       return ((dFahrenheit - 32) * 5) / 9;
    }
    [System.Web.Services.WebMethod(EnableSession=true)]
    public int GetNumberOfConversions()
    {
       return (int) Session["Conversions"];
    }
}
```

MessageName

The **MessageName** property of the **WebMethod** attribute enables the XML Web service to uniquely identify overloaded methods using an alias. Unless otherwise specified, the default value is the method name. When specifying the **MessageName**, the resulting SOAP messages will reflect this name instead of the actual method name. For more information, see WebMethodAttribute.MessageName Property.

To supply a message name for an XML Web service method

- Use the **MessageName** property of the **WebMethod** attribute, as shown below:

```
' Visual Basic
Public Class Service1
    Inherits System.Web.Services.WebService
    <System.Web.Services.WebMethod(MessageName:="AddDoubles")> _
    Public Function Add(ByVal dValueOne As Double, _
                        ByVal dValueTwo As Double) As Double
        Add = dValueOne + dValueTwo
    End Function
    <System.Web.Services.WebMethod(MessageName:="AddIntegers")> _
    Public Function Add(ByVal iValueOne As Integer, _
                        ByVal iValueTwo As Integer) As Integer
        Add = iValueOne + iValueTwo
    End Function
End Class

// C#
```

```
public class Service1 : System.Web.Services.WebService
{
    [System.Web.Services.WebMethod(MessageName="AddDoubles")]
    public double Add(double dValueOne, double dValueTwo)
    {
       return dValueOne + dValueTwo;
    }
    [System.Web.Services.WebMethod(MessageName="AddIntegers")]
    public int Add(int iValueOne, int iValueTwo)
    {
       return iValueOne + iValueTwo;
    }
}
```

The SOAP request message for the method that adds doubles, **AddDoubles**, will resemble the following:

```
POST /myWebService/Service1.asmx HTTP/1.1
Host: localhost
Content-Type: text/xml; charset=utf-8
Content-Length: length
SOAPAction: "http://tempuri.org/AddDoubles"

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <AddDoubles xmlns="http://tempuri.org/">
      <dValueOne>double</dValueOne>
      <dValueTwo>double</dValueTwo>
    </AddDoubles>
  </soap:Body>
</soap:Envelope>
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: length
```

The SOAP response message for the method that adds doubles, **AddDoubles**, will resemble the following:

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <AddDoublesResponse xmlns="http://tempuri.org/">
      <AddDoublesResult>double</AddDoublesResult>
    </AddDoublesResponse>
  </soap:Body>
</soap:Envelope>
```

TransactionOption

The **TransactionOption** property of the **WebMethod** attribute enables the XML Web service method to participate as the root object of a transaction. Even though you can set the **TransactionOption** property to any of the values of the **TransactionOption** enumeration, an XML Web service method only has two possible behaviors; it does not participate in a transaction (**Disabled**, **NotSupported**, **Supported**), or it creates a new transaction (**Required**, **RequiresNew**). Unless otherwise specified, the default value is **TransactionOption.Disabled**. For more information, see WebMethodAttribute.TransactionOption Property.

In addition to the prerequisites for any XML Web service method, you need to add a reference to **System.EnterpriseServices.dll**. This namespace contains methods and properties that expose the distributed transaction model found in COM+ Services. The **System.EnterpriseServices.ContextUtil** class lets you vote on the transaction using the **SetAbort** or **SetComplete** methods. For more information, see Participating in Transactions in XML Web Services Created Using ASP.NET and Automatic Transactions and XML Web Services.

To create a new transaction with an XML Web service method

1. Add a reference to **System.EnterpriseServices.dll**. For more information, see Adding and Removing References.
2. Add the **System.EnterpriseServices** namespace to the XML Web service, as shown below:

```
' Visual Basic
Imports System.EnterpriseServices

// C#
using System.EnterpriseServices;
```

3. Use the **TransactionOption** property of the **WebMethod** attribute, as shown below:

```
' Visual Basic
Public Class Service1
    Inherits System.Web.Services.WebService
    <System.Web.Services.WebMethod( _
       TransactionOption:=TransactionOption.RequiresNew)> _
    Public Function DoSomethingTransactional() As String
       'The transaction was successful...
       ContextUtil.SetComplete
       DoSomethingTransactional = ContextUtil.TransactionId.ToString()
    End Function
End Class

// C#
public class Service1 : System.Web.Services.WebService
{
    [System.Web.Services.WebMethod(
       TransactionOption=TransactionOption.RequiresNew)]
    public string DoSomethingTransactional()
    {
       // The transaction was successful...
       ContextUtil.SetComplete();
       return ContextUtil.TransactionId.ToString();
    }
}
```

See Also

Creating XML Web Services in Managed Code | WebMethodAttribute Class

Debugging XML Web Services in Managed Code

When you are ready to build and run your XML Web service, you must first compile the ASP.NET Web Service project. Once compiled, you can run the XML Web service. For more information, see Debugging Preparation: ASP.NET Web Service Projects.

You can choose from three methods to build and run your XML Web service:

Method	Description
With the debugger	This method starts the default browser and loads the specified start page. Running a page in the debugger makes it possible for you to step through code line-by-line and use additional analysis tools and runtime information. If Visual Studio detects that key files have changed, it will also build the project before launching the browser with the specified start page.
Without the debugger	This method makes it possible for you to run your code as it normally would run outside of the context of development tools, and therefore no runtime information is available through these tools. If Visual Studio detects that key files have changed, it will build the project before launching the browser with the specified start page. However, you can attach the debugger to the process as needed.
View in browser	This method compiles the project and opens a Web page chosen from Solution Explorer. Visual Studio compiles and runs the project in the default browser within Visual Studio.

To build and run an XML Web service with the debugger

1. In **Solution Explorer**, right-click the .asmx file for the XML Web service you wish to run and click **Set As Start Page** on the shortcut menu.
2. On the **Debug** menu, click **Start**.

   This command instructs Visual Studio to run the XML Web service in the debugger.

3. To stop running the form and return to the Code Editor, close the browser, or click **Stop Debugging** on the **Debug** menu.

To build and run an XML Web service without the debugger

1. In Solution Explorer, right-click the .asmx file for the XML Web service you wish to run and click **Set As Start Page** on the shortcut menu.
2. On the **Debug** menu, click **Start Without Debugging**.

   Visual Studio saves all files in the project and then builds it. Once built, Visual Studio launches the default browser and navigates to the project's start page.

3. To stop running the XML Web service and return to the Code Editor, close the browser.

To build and run an XML Web service using the View in Browser option

1. In **Solution Explorer**, right-click the .asmx file for the XML Web service you want to run and click **View in Browser** on the shortcut menu.

   Visual Studio builds the XML Web service and launches the specified start page within the IDE.

2. To stop running the XML Web service and return to the Code Editor, close the browser.

See Also

Creating XML Web Services in Managed Code | Debugging Preparation: ASP.NET Web Service Projects

Deploying XML Web Services in Managed Code

When using Visual Studio .NET to create XML Web services in managed code, you use a standard deployment model: you compile your project and then you deploy the resulting files to a production server. The project .dll file contains the XML Web services code-behind class file (.asmx.vb or .asmx.cs) along with all other class files included in your project, but not the .asmx file itself. You then deploy this single project .dll file to the production server without any source code. When the XML Web service receives a request, the project .dll file is loaded and executed.

Compiling and Deploying an XML Web Service in Managed Code

Suppose you have a simple ASP.NET Web Service project named WebService1 that contains the following:

- An XML Web service entry page named Service1.asmx.
- Some additional modules, named Service1.vb (or .cs) and BusinessLogic.vb (or .cs), which contain business logic.

When you compile the project, the following occurs:

- Visual Studio saves all of the files in the project you modified since the last build.
- Visual Studio copies the Service1.asmx file and the default project files to the development Web server.
- The Service1.asmx.vb (or .asmx.cs) class file, the Global.asax class file, and the BusinessLogic.vb (.cs) class file are compiled into the project .dll file, which is then copied to the server in the default \Bin directory of the target virtual directory. If the project is set to compile a debug version, Visual Studio creates a project .pdb file in the \Bin directory.

> **Note** If compilation errors occur, the deployment will not succeed. For more information, see Debugging XML Web Services in Managed Code.

To deploy the XML Web service to a server other than the development server, you can add a Web Setup project or you can copy the required files to the destination server. To make your XML Web service available to others, you will need to deploy it to a Web server that is accessible to the clients you wish to support. For more information, see Deploying Applications and Components and Copying a Project.

To deploy the XML Web service using a Web Setup project

1. On the **File** menu, point to **Add Project** and then click **New Project**.
2. Select the **Setup and Deployment Projects** folder and then click **Web Setup Project**.
3. In the **Name** box, type a name for your deployment project and then click **OK**.
4. In Solution Explorer, right-click the deployment project. On the shortcut menu, point to **Add**, and then click **Project Output**.
5. In the **Add Project Output Group** dialog box, select **Primary output**, **Debug Symbols**, and **Content Files**. For more information, see Adding and Removing Project Outputs in the File System Editor.
   - The Primary output group consists of the project .dll file and its dependencies.
   - The Debug Symbols group consists of the project PDB file.
   - The Content Files group consists of the remaining files for the XML Web service, such as the .asmx, .asax, and .config files.
6. Click **OK**.

To deploy the XML Web service by copying the project

1. In Solution Explorer, select the project you want to copy.
2. On the **Project** menu, click **Copy Project**.
3. In the **Destination project folder** box, enter the location to which you want to copy the project.
4. Select either **FrontPage** or **File share** to select a **Web access method**. For more information, see Changing Web Project Access Methods.
5. Select **Only files needed to run this application**.
6. Click **OK**.

After deploying your XML Web service, you need to consider how to make it possible for developers to locate it if you intend others to use it. For more information, see Fine-Tuning Discovery Mechanisms.

See Also

Accessing XML Web Services in Managed Code

An XML Web service client is any component or application that references and uses an XML Web service. This does not necessarily need to be a client-based application; in fact, in many cases your XML Web service clients might be other Web applications, such as Web Forms or even other XML Web services.

In This Section

Code Model for Accessing XML Web Services in Managed Code
Describes the process of accessing Web Services in managed code.
Locating XML Web Services
Provides instructions on how to locate Web Services to access.
Generating an XML Web Service Proxy
Provides instructions on how to generate a proxy class for a Web Service.
Accessing an XML Web Service in Managed Code
Provides instructions on how to access a Web Service from a client application written in managed code.
Accessing an XML Web Service Asynchronously in Managed Code
Provides instructions on how to asynchronously access a Web Service from a client application written in managed code.

Related Sections

Programming the Web with XML Web Services
Provides a portal for all Web Services documentation.
Introduction to Programming XML Web Services in Managed Code
Describes the Web Services programming paradigm.
Creating XML Web Services in Managed Code
Describes the process and provides instructions on how to create Web Services in managed code.
Building XML Web Service Clients
Describes the process and provides instructions on building XML Web Service clients.
Customizing SOAP Messages
Describes how to customize SOAP messages, including how to work with various XML encoding methods.
Using SOAP Headers
Describes how to utilize SOAP headers in Web Services created using ASP.NET and how to include additional information with SOAP messages.
Accessing XML Web Services from a Browser
Provides instructions on accessing ASP.NET Web Services from a Web browser using HTTP-GET, HTTP-POST and HTTP-SOAP.
XML Web Services Created with ATL Server
Describes the process and provides instructions on how to create and access Web Services using ATL Server and C++.

Code Model for Accessing XML Web Services in Managed Code

An XML Web service client is any component, service, or application that references and utilizes an XML Web service. It is worthwhile to note that XML Web service clients need not be a client-based application, such as a desktop application; in most cases, your XML Web service clients may be other Web applications or even other XML Web services.

When accessing XML Web services in managed code, a proxy class and the .NET Framework handle all of the infrastructure coding. Accessing an XML Web service follows these basic steps:

- Locate the XML Web service you want to access.
- Create a proxy class for the XML Web service by adding a Web reference to your project.
- Reference the proxy class in the client code by including its namespace.
- Create an instance of the XML Web service proxy class in the client code.
- Access the XML Web service using the methods of the proxy.

For more information, see Accessing an XML Web Service in Managed Code or Accessing an XML Web Service Asynchronously in Managed Code.

Locating an XML Web Service and Adding a Web Reference

To simplify the coding model, applications written in managed code use a Web reference to locally represent each XML Web service. You add a Web reference to your project using the Add Web Reference dialog box. This dialog box supports browsing Web addresses for an XML Web service. For more information, see Locating XML Web Services and Web References.

Reference the Proxy Class

The generated proxy class has its own namespace associated with it, and you must add the namespace to your client application before you can create an instance of that class. For more information, see using Directive (C#) or References and the Imports Statement (Visual Basic .NET).

Create an Instance of the Proxy Class

Before you can call any of the methods of the proxy class, you must first create an instance of that class. This process does not differ at all from creating an instance of any other class. For more information, see new (C#) or Object Variable Declaration (Visual Basic .NET).

Access the XML Web Service Using the Proxy

To access an XML Web service method, your client application invokes either the corresponding synchronous method or asynchronous methods of the proxy object. These methods in turn do the necessary work to remote the call over the wire to call the desired XML Web service method. By default, the proxy class uses SOAP to access the XML Web service method, as SOAP supports the richest set of data types of the three supported protocols (HTTP-GET, HTTP-POST, and HTTP-SOAP). For more information, see XML Web Service Wire Formats.

Summary

Using a proxy class that Visual Studio generated directly from the service description of an XML Web service, accessing an XML Web service in managed code is a relatively simple process. The proxy class does the work to transform the method call into a request message and the response message back into a method return value.

The following picture shows the relationship of an application that calls an XML Web service, which in turn calls another XML Web service to obtain pricing information. In this example, the Get Current Prices XML Web service is both an XML Web service provider and an XML Web service consumer.

XML Web Service Clients and XML Web Services

See Also

Locating XML Web Services

Sometimes you may be both the provider and consumer of an XML Web service. In this case, you probably know the location and function of the XML Web service. At other times, you may be accessing an XML Web service provided by someone else. When this occurs, you may not even know if an XML Web service that suits your purposes in fact exists.

To simplify the coding model, applications written in managed code use a Web reference to locally represent each XML Web service. You add a Web reference to your project by using the Add Web Reference dialog box. This dialog box makes it possible for you to browse your local server, the Microsoft UDDI Directory, and the Internet at large. XML Web service providers publicize their XML Web services in XML Web services directories, which is where XML Web service consumers search of XML Web services to consume. For more information, see XML Web Services Directories.

The Add Web Reference dialog box uses the process of XML Web service discovery to locate eligible XML Web services on Web sites to which you navigate in the dialog box. For a given address, it will interrogate the Web site using an algorithm designed to locate XML Web service discovery (DISCO) documents and ultimately, XML Web service description documents that adhere to the grammar of the Web Services Description Language (WSDL). For more information, see XML Web Service Discovery and XML Web Service Description.

After you have located an XML Web service for your application to access by using the Add Web Reference dialog box, clicking the Add Reference button will instruct Visual Studio to download the service description to the local machine and then generate a proxy class for the chosen XML Web service. The proxy class will contain methods for calling each exposed XML Web service method both synchronously and asynchronously. This class is contained in the local .wsdl file's code-behind file. For more information, see Web References and Add Web Reference Dialog Box.

You can use the Web Reference URL property to specify the URL to the XML Web service. The Add Web Reference sets this property by default to the URL of the XML Web service you select, which is a static URL. A Web reference can use either a static URL or a dynamic URL.

If you leave the URL Behavior set to the default value of static, the proxy class sets the URL property using a hard-coded URL when you create an instance of the class.

If you set the URL Behavior property of the Web reference to dynamic, the application obtains the URL at run time from the <appSettings> Element of your application's configuration file. When you specify a dynamic URL after adding a Web reference, Visual Studio updates the proxy class to obtain the URL from the configuration file.

```
<appSettings>
   <add key="myApplication.myServer.Service1"
        value="http://myServer/myXmlWebService/Service1.asmx"/>
</appSettings>
```

When you create an instance of a proxy object, you can also set the URL property of the object in your application.

Regardless of which URL the proxy uses, it must be for an XML Web service that adheres to a WSDL that matches the one used when adding the Web reference. For more information, see Managing Project Web References.

Alternatively, you can generate a proxy class using a tool (XML Web services Description Language Tool) that is similar to the one used by Visual Studio to create a proxy class when adding a Web reference. This is necessary when you are unable to access the XML Web service from the machine on which Visual Studio is installed, such as when the XML Web service is located on a network that will not be accessible to the client until run time. You then manually add the file that the tool generated to your application project. For more information, see Generating an XML Web Service Proxy.

See Also

Accessing XML Web Services in Managed Code | Managing Project Web References | Building XML Web Service Clients | Web Services Description Language (WSDL) Explained

Generating an XML Web Service Proxy

To access an XML Web service from a client application, you first add a Web reference, which is a reference to an XML Web service. When you create a Web reference, Visual Studio creates an XML Web service proxy class automatically and adds it to your project. This proxy class exposes the methods of the XML Web service and handles the marshalling of appropriate arguments back and forth between the XML Web service and your application. Visual Studio uses the Web Services Description Language (WSDL) to create the proxy. For more information, see XML Web Services Infrastructure.

You can use the Web Reference URL property to specify the URL to the XML Web service. The Add Web Reference sets this property by default to the URL of the XML Web service you select, which is a static URL. A Web reference can use either a static URL or a dynamic URL. If you set the URL Behavior property of the Web reference to dynamic, the application obtains the URL at run time from the <appSettings> Element section of your application's configuration file. For more information, see Managing Project Web References.

After adding a Web reference to a client application, you can invoke the exposed methods of that XML Web service and access the results just as you would any other method of a component. For more information, see Web References.

If you are unable to add a Web reference, such as when the XML Web service is not accessible from the machine on which you are using Visual Studio, you can use the XML Web Services Description Language Tool (Wsdl.exe) to generate an XML Web service client proxy class. For more information, see Creating an XML Web Service Proxy.

See Also

Accessing XML Web Services in Managed Code | Web References | Managing Project Web References | Creating an XML Web Service Proxy | Web Services Description Language Tool (Wsdl.exe)

Accessing an XML Web Service in Managed Code

Accessing an XML Web service from managed code is a straightforward process. First, you add a Web reference to your project for the XML Web service you wish to access. The Web reference creates a proxy class with methods that serve as proxies for each exposed method of the XML Web service. Next, you add the namespace for the Web reference. Finally, you create an instance of the proxy class and then access the methods of that class as you would the methods of any other class. For more information, see Code Model for Accessing XML Web Services in Managed Code.

By default, adding a Web reference also adds methods to the proxy class for accessing the XML Web service asynchronously. For more information, see Accessing an XML Web Service Asynchronously in Managed Code.

To access an XML Web service in managed code

1. Create the application from which you want to access an XML Web service. This application could even be another XML Web service.
2. Add a Web reference for the XML Web service with which your application will interact. For instructions, see Adding and Removing Web References.
3. Create an instance of the proxy object in your client code where you want to access the XML Web service.
4. Access the methods of the XML Web service as you would any other component.

   In the example code below, the client application (**Application1**) is accessing an XML Web service for which it has a Web reference (**Converter**) that contains a proxy class (**Service1**), which has a method (**ConvertTemperature**) for calling the XML Web service. The two lines of code in bold represent the code that is necessary to access the XML Web service.

```
' Visual Basic
Imports System
Module Module1
    Sub Main()
        Dim cService As New Converter.Service1()
        Dim dFahrenheit As Double
        Dim dCelsius As Double
        Console.Write("Temperature in degrees Fahrenheit: ")
        dFahrenheit = Convert.ToDouble(Console.ReadLine())
        dCelsius = cService.ConvertTemperature(dFahrenheit)
        Console.Write("Temperature in degrees Celsius: ")
        Console.WriteLine(dCelsius.ToString())
    End Sub
End Module

// C#
using System;
namespace Application1
{
   class Class1
   {
      static void Main()
      {
         Converter.Service1 cService = new Converter.Service1();
         Console.WriteLine("Temperature in degrees Fahrenheit: ");
         double dFahrenheit = Convert.ToDouble(Console.ReadLine());
         double dCelsius = cService.ConvertTemperature(dFahrenheit);
         Console.Write("Temperature in degrees Celsius: ");
         Console.WriteLine(dCelsius.ToString());
      }
   }
}
```

See Also

Accessing an XML Web Service Asynchronously in Managed Code

When adding a Web reference, Visual Studio generates a proxy class that is capable of calling an XML Web service both synchronously and asynchronously. By using an asynchronous method, you can continue to use the calling thread while waiting for the XML Web service to respond. This makes it possible for you to efficiently use the existing set of threads in the client application, which can be critical if the client is a Web application. The XML Web service does not require special configuration to support asynchronous calls nor does it know in which mode the client is calling it because the request and response message behavior is the same for synchronous and asynchronous calls. Client-side asynchronous calls simply uses a system I/O completion port to efficiently wait for the server's response.

For each synchronous method, the generated proxy class also contains a corresponding *Begin* and *End* method. Therefore, if the name of the XML Web service method is **ConvertTemperature**, the name of the synchronous method in the proxy class is also **ConvertTemperature** and the additional proxy methods to support asynchronous calls are **BeginConvertTemperature** and **EndConvertTemperature**.

Calling an XML Web service asynchronously is a two-step operation. The first step, calling the *Begin* method, initiates the XML Web service call. The second step, calling the *End* method, completes the XML Web service call and returns the XML Web service response.

The *Begin* method returns a **System.Web.Services.Protocols.WebClientAsyncResult** object. This object, which implements **System.IAsyncResult**, serves two purposes. First, it provides status about the pending asynchronous call. For example, the **IsCompleted** property indicates if the operation has completed. Second, by passing this object to the *End* method, the proxy can identify which request you would like to complete. This is important because you can make multiple asynchronous calls simultaneously.

There are several ways to determine when the asynchronous XML Web service call has completed:

- The first is to supply a callback delegate to the *Begin* method. After the XML Web service returns its response, a thread will call the callback from the thread pool; this is likely to be a different thread than the one that you used to call the *Begin* method. Typically, you will then call the *End* method inside that callback. Using the callback is the most efficient way to manage usage of your threads since callback functions do not block threads while waiting for the response.
- The second is to wait using one of the **WaitHandle** methods of the **IAsyncResult.AsyncWaitHandle** object. When using the methods of the **WaitHandle** class, the client can also specify a timeout after which it will abandon waiting for results from the XML Web services it called.
- The third is to poll the value of **IAsyncResult.IsCompleted**. When this property returns true, the XML Web service response is available.
- The fourth is to simply call the *End* method directly. This method will not return until the asynchronous operation is complete since it uses the **IAsyncResult.AsyncWaitHandle**.

The **WaitHandle** class makes it possible for clients to make asynchronous call and wait for:

- a single XML Web service (**WaitHandle.WaitOne**),
- the first of many XML Web services (**WaitHandle.WaitAny**), or
- all of many XML Web services (**WaitHandle.WaitAll**) to return results.

If you are making multiple asynchronous XML Web service calls simultaneously, then you might want to use the **WaitHandle.WaitAll** method to block the thread until all calls have completed. If you want to process the results as they arrive, you can use the **WaitHandle.WaitAny** method. This method will indicate that one of the operations has completed and will identify the completed operation.

When accessing XML Web services asynchronously from a Web application, using the methods of the **WaitHandle** class is the preferred method. Otherwise, if execution in the Web application completes before the asynchronous process calls the callback function, as described in the previous section, it may never have the opportunity to do so if the original function's completion results in the destruction of the instance of the Web Form.

If you want to abort an asynchronous XML Web service call, cast the **IAsyncResult** returned from the *Begin* method to **WebClientAsyncResult** and call its **Abort** method.

For more information, see Communicating with XML Web Services Asynchronously.

To call an XML Web service asynchronously using a callback function in managed code

1. Create the application from which you want to access an XML Web service. This application could even be another XML Web service.
2. Add a Web reference for the XML Web service with which your application will interact. For instructions, see Adding and Removing Web References.
3. Implement a callback function. The callback function contains the call to the *End* method of the XML Web service method.
4. Create an instance of the proxy object in your client code where you want to access the XML Web service.
5. Call the *Begin* method of the XML Web service method.
6. Continue doing any other processing in the client application.
7. After the XML Web service finishes processing and returns the results to the proxy, the proxy will call the callback function.

The code shown below is from a Windows application that is accessing an XML Web service for which it has a Web reference (**Translator**) that contains a proxy class (**Service1**), which has a *Begin* method (**BeginLongProcess**) for calling the XML Web service asynchronously and an *End* method (**EndLongProcess**) for completing the call and obtaining the results. In this example, the XML Web service accepts a string, performs some time-consuming translation, and then returns the result. After initiating the asynchronous call to the XML Web service method, the event handler **button1_Click** returns control to the client. When the XML Web service returns its results, it calls the callback function.

```
' Visual Basic
Private Sub Button1_Click(ByVal sender As System.Object, _
                          ByVal e As System.EventArgs) _
                          Handles Button1.Click

   Dim cService As New Translator.Service1()

   ' cb is essentially a function pointer to ServiceCallback.
   Dim cb As New AsyncCallback(AddressOf ServiceCallback)

   ' Call the Begin method of the proxy class to initiate the
   ' asynchronous call to the XML Web service method.
   cService.BeginLongProcess(TextBox1.Text, cb, cService)

End Sub

Public Sub ServiceCallback(ByVal ar As IAsyncResult)

   Dim cService As Translator.Service1

   ' Retrieve the original state for the proxy.
   cService = ar.AsyncState

   ' Retrieve results by calling the End method of the proxy class.
   TextBox2.Text = cService.EndLongProcess(ar)

End Sub

// C#
private void button1_Click(object sender, System.EventArgs e)
{
   Translator.Service1 cService = new Translator.Service1();

   // cb is essentially a function pointer to ServiceCallback.
   AsyncCallback cb = new AsyncCallback(ServiceCallback);

   // Call the Begin method of the proxy class to initiate the
   // asynchronous call to the XML Web service method.
   cService.BeginLongProcess(textBox1.Text, cb, cService);
}
```

```
public void ServiceCallback(IAsyncResult ar)
{
   // Retrieve the original state for the proxy.
   Translator.Service1 cService = (Translator.Service1)ar.AsyncState;

   // Retrieve results by calling the End method of the proxy class
   textBox2.Text = cService.EndLongProcess(ar);
}
```

To call an XML Web service asynchronously using a WaitHandle in managed code

1. Create the application from which you want to access an XML Web service. This application could even be another XML Web service.
2. Add a Web reference for the XML Web service with which your application will interact. When you add a Web reference, Visual Studio creates a proxy class that contains methods for accessing each exposed method of that XML Web service. For instructions, see Adding and Removing Web References.

   To access the XML Web service, you must create an instance of this proxy class and then interact with the methods just as you would with the methods of any other instance of a class.

3. Include the namespace **System.Threading**. You will need this namespace to access the **WaitHandle** class.
4. Create an instance of the proxy object in your client code where you want to access the XML Web service.
5. Create an interface to an **AsyncResult** by calling the *Begin* method of the XML Web service method, which returns a type that implements **IAsyncResult** interface.
6. Continue doing any other processing in the client application.
7. When you reach a point that you require the results from the XML Web service, use a **WaitHandle** to halt processing and wait for the results.
8. Once the XML Web service finishes processing and returns the results to the proxy, the proxy will call the callback function.

   The code shown below is from a Web application that is accessing an XML Web service for which it has a Web reference (**Translator**) that contains a proxy class (**Service1**), which has a *Begin* method (**BeginLongProcess**) for calling the XML Web service asynchronously and an *End* method (**EndLongProcess**) for completing the call and obtaining the results. In this example, the XML Web service accepts a string, performs some time-consuming translation, and then returns the result. After initiating the asynchronous call to the XML Web service method, the client application continues with some additional processing and then waits for the XML Web service to return. When the XML Web service returns, processing continues.

```
' Visual Basic
Private Sub Button1_Click(ByVal sender As System.Object, _
                          ByVal e As System.EventArgs) _
                          Handles Button1.Click

   Dim cService As New Translator.Service1()
   Dim ar1 As IAsyncResult
   Dim ar2 As IAsyncResult

   ' Call the Begin method of the proxy class to initiate the
   ' asynchronous call to the XML Web service method.
   ar1 = cService.BeginLongProcess(TextBox1.Text, Nothing, Nothing)
   ar2 = cService.BeginLongProcess(TextBox3.Text, Nothing, Nothing)

   '...
   ' Keep processing.
   '...

   Dim wh() As WaitHandle = {ar1.AsyncWaitHandle, ar2.AsyncWaitHandle}

   ' Waits for both async XML Web service calls to finish.
   WaitHandle.WaitAll(wh)
```

```
    ' Retrieves the results by calling the End method of the proxy
    ' class.
    TextBox2.Text = cService.EndLongProcess(ar1)
    TextBox4.Text = cService.EndLongProcess(ar2)
End Sub

// C#
private void Button1_Click(object sender, System.EventArgs e)
{
    Translator.Service1 cService = new Translator.Service1();
    IAsyncResult ar1 = cService.BeginLongProcess(TextBox1.Text,
                                                 null, null);
    IAsyncResult ar2 = cService.BeginLongProcess(TextBox3.Text,
                                                 null, null);

    //...
    // Keep processing.
    //...

    WaitHandle[] wh = {ar1.AsyncWaitHandle, ar2.AsyncWaitHandle};

    // Waits for both async XML Web service calls to finish.
    WaitHandle.WaitAll(wh);

    // Retrieves the results by calling the End method of the proxy
    // class.
    TextBox2.Text = cService.EndLongProcess(ar1);
    TextBox4.Text = cService.EndLongProcess(ar2);
}
```

See Also

Accessing XML Web Services in Managed Code | Communicating with XML Web Services Asynchronously | WaitHandle Class | IAsyncResult Interface | Including Asynchronous Calls

Creating Windows Applications Portal

You start to create traditional Windows applications and client/server applications by designing your forms with the Windows Forms Designer. You assign characteristics to the forms and put controls on them, then write code to augment the functionality of the controls and forms. You can also inherit from forms.

The Help topics about this subject are located in another part of the Visual Studio table of contents, under the section named "Developing with Visual Studio .NET." The links below jump to Help topics in that section on creating Windows applications.

In the Visual Studio Documentation on Windows Applications

Code for Windows Forms Applications
Gives links to topics that demonstrate how to perform common programming tasks in Windows Forms applications.
Introduction to Windows Applications
Discusses the particulars of Windows application creation with Visual Studio .NET
Windows Forms Walkthroughs
Provides step-by-step instructions for creating a number of Windows applications.
Windows Forms
Provides links to topics about the technologies and tools for creating Windows applications.
Windows Forms Controls
Provides links to topics about the controls designed specifically to work with Windows Forms.
Windows Service Applications
Describes how to create long-running executables.

In the Visual Basic and Visual C# Documentation

Visual Basic Code Example Topics
Lists quickstart-like code examples that help you perform common tasks in Visual Basic .NET.
Decision Chart
Provides an interactive guide to help you choose which approach or technology to use, as well as links to more information.
Windows Forms and Web Forms Recommendations
Discusses considerations for choosing between the two forms packages.
Windows Applications Namespaces in Visual Studio
Provides links to namespaces relevant to Windows Forms.

.NET Framework Class Library in Visual Studio

The .NET Framework class library is comprised of namespaces. Each namespace contains types that you can use in your program: classes, structures, enumerations, delegates, and interfaces.

When you create a Visual Basic or Visual C# project in Visual Studio, the most common base class DLLs (assemblies) are already referenced. However, if you need to use a type that is in a DLL not already referenced, you will need to add a reference to the DLL. For more information, see Adding and Removing References.

The topics below provide the following information:

- Lists of the most important namespaces for each feature area.
- Links to reference topics in the .NET Framework about each major namespace.
- Links to procedural and conceptual topics about how to use those namespaces in your Visual Basic .NET and Visual C# .NET applications.

In This Section

Introduction to the .NET Framework Class Library in Visual Studio
Explains namespaces and assemblies.
Programming Basics Namespaces in Visual Studio
Covers the .NET namespaces for collections, character encodings, file I/O, and threading.
Data Namespaces in Visual Studio
Covers the namespaces in the .NET Framework for manipulating and accessing data from traditional data sources and from standards-based XML.
Web Namespaces in Visual Studio
Covers .NET namespaces for ASP.NET Web applications and XML Web services.
Windows Applications Namespaces in Visual Studio
Covers the namespaces in the .NET Framework for creating rich Windows applications and graphics with Windows Forms.
Component Model Namespaces in Visual Studio
Covers the namespaces for implementing the run-time and design-time behavior of components and controls.
Framework Services Namespaces in Visual Studio
Covers the namespaces for interacting with a variety of back-end server resources.
Security Namespaces in Visual Studio
Covers the namespaces that provide the underlying structure of the .NET Framework security system and that implement ASP.NET security in Web applications.
Network Namespaces in Visual Studio
Covers the namespace that provides a simple programming interface to many of the protocols found on the network.
Configuration Namespaces in Visual Studio
Covers the namespaces that work with .NET Framework configuration settings and handle errors in configuration files, and that allow you to write custom installers for components.
Globalization and Localization Namespaces in Visual Studio
Provides links to the namespaces in the .NET Framework that are designed for developing world-ready applications.
Reflection Namespaces in Visual Studio
Covers the namespace that provides a managed view of loaded types, methods, and fields, with the ability to dynamically create and invoke types.
Visual Basic Runtime Namespace in Visual Studio
Describes the modules, enumerations, and attributes that provide the properties, functions, and constants you can use in Visual Basic .NET applications.

Related Sections

Inside the .NET Framework
Provides links to essential information about the common language runtime, assemblies, common type system, metadata, interoperability, security, and other features of the .NET Framework.
Class Library (.NET Framework)
Describes the functionality provided by the types (classes, structures, enumerations, delegates, and interfaces) in the .NET Framework and lists the base data types provided by the **System** namespace.
Microsoft .NET Web site (http://www.microsoft.com/net/)
Provides the latest information and resources about .NET.
Adding and Removing References

Explains how to reference external projects and .NET and COM components so that you can make use of them in your application.

Assemblies

Provides links to topics that explain what assemblies do and how to use them.

Introduction to the .NET Framework Class Library in Visual Studio

A key feature of the .NET Framework is the base class library. For more information about the common language runtime, the class library, the common language specification (CLS), and other features of the .NET Framework, see Inside the .NET Framework.

Namespaces

The .NET Framework class library is comprised of namespaces. Each namespace contains types that you can use in your program: classes, structures, enumerations, delegates, and interfaces. For an overview of the namespaces, see Class Library.

Namespaces provide scope: Two classes with the same name can be used in your program as long as they are in different namespaces and as long as you qualify the names to the namespaces. The namespace name is part of the fully qualified name of the type (*namespace.typename*).

All namespaces shipped by Microsoft begin with one of two names: System or Microsoft.

Managed DLLs (Assemblies)

The functionality of the .NET Framework class library is not contained in a single DLL. By putting the functionality of the base classes in multiple DLLs, a managed program does not need to load, at start up, a large DLL, but, rather, one or more smaller DLLs. This reduces the startup time for a program.

You can use namespaces that are defined in your project. However, you will commonly use types in namespaces that reside in a managed DLL. A managed DLL is also called an *assembly*.

When you create a Visual Basic or Visual C# project in Visual Studio, the most common base class DLLs (assemblies) are already referenced. However, if you need to use a type that is in a DLL not already referenced, you will need to add a reference to the DLL. The Add Reference dialog box allows you to add an assembly.

Namespaces and Assemblies: Usage Summary

The following is a summary of how you will commonly use managed DLLs (assemblies), namespaces, and the types in the namespaces:

1. Locate a class that provides the functionality you need. For more information, see .NET Framework Class Library in Visual Studio and Common Tasks in Visual Basic and Visual C#.
2. In the documentation overview of the type, note the name of the type's assembly and namespace.
3. See if the assembly is already referenced in your project. Open **Solution Explorer** and look under the **References** node.
4. If you do not see an assembly reference, right-click the **References** node and select **Add Reference**. For more information, see Adding and Removing References.
5. After you have an assembly reference, you can access the types in the assembly.

> **Tip** If you do not want to use fully qualified names (namespace.typename), you can use the Imports statement in Visual Basic or the using keyword in Visual C#.

The following example shows how Visual Basic and Visual C# implicitly use the namespace (and in Visual Basic, the class) qualifier:

```
' Visual Basic
' TestImports.vb
Imports System.Console
Module Module1
   Sub Main()
      WriteLine("Hello")
      ' Without the Imports statement, you would have to use
      ' System.Console.WriteLine("Hello").
      End Sub
End Module

// C#
// TestUsing.cs
using System;
```

```
class MyClass
{
   public static void Main()
   {
      Console.WriteLine("Hello");
      //  Without the using statement, you would have to use
      // System.Console.WriteLine("Hello");
   }
}
```

You can put this code into a file, called test.vb or test.cs, and compile it at the command line by typing **vbc TestImports.vb** or **csc TestUsing.cs**. You do not have to explicitly reference the mscorlib.dll, which is the assembly that contains the **System** namespace; the Visual Basic or Visual C# compiler always references mscorlib.dll.

See Also

.NET Framework Class Library in Visual Studio | Class Library (.NET Framework) | Inside the .NET Framework | Microsoft .NET Web site (http://www.microsoft.com/net/)

Programming Basics Namespaces in Visual Studio

The fundamental programming namespaces in the .NET Framework include:

- System.Collections - contains interfaces and classes that define various collections of objects, such as lists, queues, arrays, hashtables, and dictionaries.
- System.IO - contains types that allow synchronous and asynchronous reading and writing on data streams and files.
- System.Text - contains classes representing ASCII, Unicode, UTF-7, and UTF-8 character encodings; abstract base classes for converting blocks of characters to and from blocks of bytes; and a helper class that manipulates and formats String objects without creating intermediate instances of **String**.
- System.Threading - provides classes and interfaces that enable multithreaded programming. This namespace includes a **ThreadPool** class that manages groups of threads, a **Timer** class that enables a delegate to be called after a specified amount of time, and a **Mutex** class for synchronizing mutually exclusive threads. **System.Threading** also provides classes for thread scheduling and wait notification.

In the Visual Studio Documentation

Walkthrough: Authoring a Simple Multithreaded Component with Visual Basic
Provides step-by-step instructions that include how to use the **System.Threading.Thread** object to add multiple threads of execution to your component.

Creating File System and Timer Components
Introduces components that respond to changes to files and directories, raise events on a schedule, or monitor and interact with Windows processes.

Walkthrough: Creating Your Own Collection Class (Visual Basic)
Provides step-by-step instructions for how to inherit from a class in **System.Collections** to create simple, strongly typed collection with Visual Basic.

C# Tutorials
Lists topics that demonstrate threading, attributes, delegates, events, and collection classes that make use of **System.Collections**; an unsafe code tutorial that covers the **System.Thread** class; and an indexers tutorial that covers **System.IO**.

Processing Drives, Folders, and Files (Visual Basic)
Lists topics that discuss using the **System.IO** namespace in Visual Basic.

Introduction to the .NET Framework Class Library in Visual Studio
Provides an overview of the namespaces in the .NET Framework.

Additional Information

Working with I/O
Provides information from the *.NET Framework Developer's Guide* about I/O.

Threading
Provides information from the *.NET Framework Developer's Guide* about threading.

Data Namespaces in Visual Studio

The data and XML namespaces in the .NET Framework include:

- System.Data - consists of the classes that constitute the ADO.NET architecture, which is the primary data access method for managed applications. The ADO.NET architecture enables you to build components that efficiently manage data from multiple data sources. ADO.NET also provides the tools to request, update, and reconcile data in distributed applications.
- System.Data.Common - contains classes shared by the .NET Framework data providers. Data providers describe a collection of classes used to access a data source, such as a database, in the managed space.
- System.Xml - classes that provide standards-based support for processing XML.
- System.Data.OleDb – classes that make up the .NET Framework Data Provider for OLE DB-compatible data sources. These classes allow you to connect to an OLE DB data source, execute commands against the source, and read the results.
- System.Data.SqlClient – classes that make up the .NET Framework Data Provider for SQL Server, which allows you to connect to SQL Server 7.0, execute commands, and read results. The **System.Data.SqlClient** namespace is similar to the **System.Data.OleDb** namespace, but is optimized for access to SQL Server 7.0 and later.
- System.Data.SqlTypes - Provides classes for native data types within SQL Server. These classes provide a safer, faster alternative to other data types.
- System.Data.Odbc - classes that make up the .NET Framework Data Provider for ODBC. These classes allow you to access ODBC data source in the managed space.
- System.Data.OracleClient - classes that make up the .NET Framework Data Provider for Oracle. These classes allow you to access an Oracle data source in the managed space.

In the Visual Studio Documentation

Accessing Data
Provides conceptual and procedural information about how to work with data and XML with Visual Basic and Visual C#.
Introduction to Distributed Applications and Data Integration
Provides an overview of data access in Visual Studio applications.
Data Walkthroughs
Lists step-by-step topics covering common data scenarios.
Connecting to Data Sources with ADO.NET
Provides topics about creating and working with connection objects in Visual Studio.
ADO.NET Datasets
Provides topics about what datasets are and how to use them.
ADO.NET Data Adapters
Provides topics about defining the SQL statement or stored procedure calls that you use to fill datasets and write data back to the database.
Filtering and Sorting in Datasets
Provides topics about viewing the data in a dataset in different ways by selecting subsets of records or sorting.
Relationships in ADO.NET Datasets
Provides topics about working with dataset tables that have a parent-child relationship.
Performing Database Operations Directly
Provides topics about how to execute SQL statements and stored procedures directly and work with the results they return.
Concurrency Control in ADO.NET
Provides topics about making changes in datasets and then writing those changes back to a data source.
Introduction to the .NET Framework Class Library in Visual Studio
Provides an overview of the namespaces in the .NET Framework.
Data Access Technologies
Helps you choose a data-access technology.
XML in Visual Studio
Provides links to sections of the Visual Studio documentation that discuss XML.
Overview of Visual Database Tools
Introduces the Query Designer and other data tools.
Data Design
Provides links to Visual Studio topics about database design and data validation.

Additional Information

XML Schema Reference (XSD)

Provides reference topics for elements and data types in XML Schema, the standard developed by the World Wide Web Consortium (W3C).

Processing Transactions

Provides information from the *.NET Framework Developer's Guide* about transactions.

XML Developer Center on MSDN Online (http://msdn.microsoft.com/xml)

Provides the latest articles and other resources about XML.

World Wide Web Consortium (W3C) Web Site (http://www.w3.org/)

Provides resources about Internet standards, including XML Schema, HTML, and HTTP.

Web Namespaces in Visual Studio

The namespaces in the .NET Framework about ASP.NET Web applications and XML Web services include:

- System.Web - contains classes and interfaces that enable browser/server communication. This namespace classes for managing HTTP output to the client (**HttpResponse**) and reading HTTP requests (**HttpRequest**). Additional classes provide facilities for server-side utilities and processes, cookie manipulation, file transfer, exception information, and output cache control.
- System.Web.UI – contains classes for creating Web Form pages, including the **Page** class and other standard classes used to create Web user interfaces.
- System.Web.UI.HtmlControls – contains classes for HTML-specific controls that can be added to Web Forms to create Web user interfaces.
- System.Web.UI.WebControls – contains classes for creating ASP.NET Web sever controls. When added to a Web Form, these controls render browser-specific HTML and script to create a device-independent Web user interface.
- System.Web.Services - contains classes that enable you to build and use XML Web services, which are programmable entities residing on a Web server and exposed via standard Internet protocols.

In the Visual Studio Documentation

Introduction to ASP.NET Web Applications in Visual Studio
Provides an overview of the components of an ASP.NET Web application, including Web Forms and XML Web services, and how Visual Studio helps you create a Web application.
Creating Web Applications and Services
Provides links to all the Visual Studio topics about ASP.NET Web applications and XML Web services.
Web Forms Pages
Provides links to topics that describe what Web Forms pages are, the architecture of Web Forms pages, and how to create and work with Web Forms pages.
Programming the Web with XML Web Services
Provides information about creating and accessing XML Web services.
Working with Web Projects
Explains basic concepts about creating and managing ASP.NET Web applications, XML Web services created using ASP.NET, and Empty Web projects.
Web Walkthroughs
Lists all the step-by-step topics that incorporate Web Forms pages.
XML Web Services Walkthroughs
Lists all the step-by-step topics about creating or accessing XML Web services.
Windows Forms and Web Forms Recommendations
Discusses considerations when choosing between the two forms packages.
ASP.NET Technology Backgrounder
Provides a summary of ASP.NET technology and where Web Forms pages fit into it.
Introduction to the .NET Framework Class Library in Visual Studio
Provides an overview of the namespaces in the .NET Framework.

Additional Information

Developing ASP.NET Server Controls
Provides information from the *.NET Framework Developer's Guide* about creating custom controls for Web Forms pages.
IBuySpy Web Site (http://www.ibuyspy.com/)
Provides sample code for ASP.NET development.
MSDN Web Site (http://msdn.microsoft.com)
Provides the latest information about XML Web services, ASP.NET, and other Microsoft technologies.

Windows Applications Namespaces in Visual Studio

The namespaces in the .NET Framework for creating rich Windows applications and graphics include:

- System.Windows.Forms - contains classes for creating Windows-based applications that take full advantage of the rich user interface features available in the Microsoft Windows operating system. In this namespace you will find the **Form** class and many other controls that can be added to forms to create user interfaces.
- System.Drawing - provides access to GDI+ basic graphics functionality. More advanced functionality is provided in the **System.Drawing.Drawing2D**, **System.Drawing.Imaging**, and **System.Drawing.Text** namespaces.
- System.ServiceProcess - provides classes that allow you to install and run long-running executables that do not have a user interface.

In the Visual Studio Documentation

Windows Forms and Web Forms Recommendations
Discusses considerations when choosing between the two forms packages.
Windows Forms
Provides links to topics about the technologies and tools for creating Windows applications.
Windows Forms Controls
Provides links to topics about the controls designed specifically to work with Windows Forms.
Windows Service Applications
Describes how to create long-running executable files.
Windows Forms Walkthroughs
Lists all the step-by-step topics that use Windows Forms in common scenarios.
Providers of Data to Windows Forms
Describes how to bind data to controls on Windows Forms and Web Forms.
GDI+ Graphics
Provides an introduction to GDI+ for Visual Basic and Visual C# users.
Color Changes in Visual Basic .NET
Provides.NET Framework **System.Drawing.Color** and **System.Drawing.SystemColors** types that are equivalent to color constants in Visual Basic 6.0.
Introduction to the .NET Framework Class Library in Visual Studio
Provides an overview of the namespaces in the .NET Framework.

Additional Information

GDI+ SDK
Provides a comprehensive reference with C++ code examples for all aspects of GDI+ programming.
Drawing and Editing Images
Provides additional information about creating GDI+ graphics with the .NET Framework.
System.Windows.Forms Hierarchy
Provides links to all the members of the **System.Windows.Forms** namespace
System.Drawing Hierarchy
Provides links to all the members of the **System.Windows.Forms** namespace
System.ServiceProcess Hierarchy
Provides links to all the members of the **System.ServiceProcess** namespace

Component Model Namespaces in Visual Studio

The namespaces in the .NET Framework for creating your own components and controls include:

- System.ComponentModel - provides classes that are used to implement the run-time and design-time behavior of components and controls. This namespace includes the base classes and interfaces for implementing attributes, working with type converters, binding to data sources, and licensing components. The classes in this namespace divide into the following categories:
  - **Core component classes**. For more information, see Component and Container classes and IContainer and IComponent interfaces.
  - **Component licensing**. For more information, see License, LicenseManager, LicenseProvider, and LicenseProviderAttribute classes.
  - **Attributes**. For more information, see Attribute class.
  - **Descriptors and persistence**. For more information, see TypeDescriptor, EventDescriptor, and PropertyDescriptor classes.
  - **Type converters**. For more information, see TypeConverter class.
- System.CodeDOM - contains classes that can be used to represent the elements and structure of a source code document.

In the Visual Studio Documentation

Component Authoring
Provides links to all the topics about creating your own component or Windows Forms control with Visual Basic or Visual C#.
Component Programming Walkthroughs
Lists all the step-by-step topics about creating your own component or control.
Framework Services Namespaces
Provides links to reference and conceptual topics about components that interact with a variety of back-end server resources.
Introduction to the .NET Framework Class Library in Visual Studio
Provides an overview of the namespaces in the .NET Framework.

Additional Information

Developing ASP.NET Server Controls
Provides information from the *.NET Framework Developer's Guide* about creating custom controls for Web Forms pages.
Interoperating with Unmanaged Code
Provides information from the *.NET Framework Developer's Guide* about exposing COM components to the .NET Framework and .NET Framework components to COM.
Inside the .NET Framework
Provides links to essential information about the common language runtime, assemblies, common type system, metadata, interoperability, security, and other features of the .NET Framework.

Framework Services Namespaces in Visual Studio

The namespaces in the .NET Framework for interacting with a variety of back-end server resources include:

- System.Diagnostics - consists of classes that allow you to debug your application and to trace the execution of your code. For more information, see the Trace and Debug classes. **System.Diagnostics** also provides classes that allow you to start system processes, read and write to event logs, and monitor system performance using performance counters. For more information, see the Process, EventLog, and PerformanceCounter classes.
- System.DirectoryServices - consists of classes that provide easy access to the Active Directory from managed code. The classes in this namespace can be used with any of the Active Directory service providers. The current providers are: Internet Information Services (IIS), Lightweight Directory Access Protocol (LDAP), Novell NetWare Directory Service (NDS), and Windows NT. For more information about the Active Directory, see Directory Services.
- System.IO - provides classes, including the FileSystemWatcher class, which listens to the file system change notifications and raises events when a directory, or file in a directory, changes.
- System.Management - provides classes for managing information and events about the system, devices, and applications instrumented to the Windows Management Instrumentation (WMI) infrastructure.
- System.Messaging - provides classes that allow you to connect to message queues on the network, send messages to queues, and receive or peek (read without removing) messages from queues.
- System.ServiceProcess - provides classes that allow you to install and run services. Services are long-running executables that run without a user interface. They can be installed to run under a system account that enables them to be started at computer reboot. Services whose implementation is derived from the processing in the **ServiceBase** class can define specific behavior for start, stop, pause, and continue commands, as well as behavior to take when the system shuts down.
- System.Timers - provides the server-based Timer component, which allows you to raise an event on a specified interval.

> **Note** There is also a Timer control for Windows Forms in the **System.Windows.Form** namespace and a Timer class in the **System.Threading** namespace.

In the Visual Studio Documentation

Monitoring Performance Thresholds
Provides basic information on monitoring Microsoft Windows performance thresholds.

Logging Application, Server, and Security Events
Provides additional information on monitoring Windows 2000 event logs by reading information from and writing information to them.

Creating Messaging Components
Discusses how to use the **MessageQueue** component to incorporate message-based communication into your application.

Introduction to the .NET Framework Class Library in Visual Studio
Provides an overview of the namespaces in the .NET Framework.

Monitoring and Managing Windows Processes
Provides an overview of the **Process** component and how you can use it to interact with system resources.

Tracing and Instrumenting Applications in Visual Basic and Visual C#
Provides an overview of the trace feature and how you can use it to generate output from your applications.

Creating Active Directory Components
Provides an overview on how you can use the **DirectorySearcher** and **DirectoryEntry** components to retrieve information from Active Directory file systems.

Reacting to File System Events
Provides an overview of how to use the **FileSystemWatcher** component to monitor directories for changes and react to these changes when they occur.

Monitoring Windows Services
Describes how to use the **ServiceController** component to interact with existing services.

Creating Timer-Based Server Tasks
Describes how to use the server-based **Timer** component to react to periodic events on a server.

Programming with Components
Presents a high-level overview of component programming and authoring.

Introduction to ASP.NET Web Applications in Visual Studio
Provides an overview of the components of an ASP.NET Web application, including Web Forms and XML Web services, and how Visual Studio helps you create a Web application.

Additional Information

Message Queuing (MSMQ)

Provides an entry point into the Message Queuing documentation in the Platform SDK.

Directory Services

Provides an entry point into the Active Directory documentation in the Platform SDK.

Event Logging

Provides an entry point into the Event Logging documentation in the Platform SDK.

Security Namespaces in Visual Studio

The security namespaces in the .NET Framework include:

- System.Security - provides the underlying structure of the common language runtime security system, including base classes for permissions.
- System.Web.Security - contains classes that are used to implement ASP.NET security in Web applications.

In the Visual Studio Documentation

Securing Applications Portal
Provides links to all the Visual Basic and C# topics about security.
Introduction to Web Application Security
Presents an overview of the security features provided by the ASP.NET page framework that leverage the security infrastructure supplied in the Windows operating system and Internet Information Services (IIS).
Code Security and Signing in Components
Describes how to make your application secure at installation and run time.
Message Queue Security
Gives an overview of Microsoft Windows Message Queuing.
Security Portal
Provides links to more resources.
Securability
Discusses how to provide security to an application and its data.
Introduction to the .NET Framework Class Library in Visual Studio
Provides an overview of the namespaces in the .NET Framework.

Additional Information

.NET Framework Security
Provides information from the *.NET Framework Developer's Guide* about Web application security, code-access security, and role-based security.
Securing Applications
Provides information from the *.NET Framework Developer's Guide* about classes and services that enable developers to write secure code and enable system administrators to customize the access that code has to protected resources.
Security Best Practices
Presents Platform SDK recommendations about privileges, buffer overruns, discretionary access control lists (DACLs), and passwords.

Network Namespaces in Visual Studio

The network namespaces in the .NET Framework include:

- System.Net - contains classes that provides a simple programming interface to many of the protocols found on the network today. The **WebRequest** and **WebResponse** classes form the basis of "pluggable protocols," an implementation of network services that enables you to develop applications that use Internet resources without worrying about the specific details of the protocol used.

In the Visual Studio Documentation

Walkthrough: Creating a Distributed Application
Demonstrates the use of **System.Net** to configure Integrated Windows authentication.
Controls and Programmable Objects Compared in Different Languages and Libraries
Mentions classes in the **System.Net** namespace that replace Internet controls in earlier versions of Visual Basic.
Introduction to the .NET Framework Class Library in Visual Studio
Provides an overview of the namespaces in the .NET Framework.

Additional Information

Best Practices for System.Net Classes
Provides recommendations for using the classes in **System.Net**.
Securing XML Web Services Created Using ASP.NET
Demonstrates the use of **System.Net** in security for XML Web services.

Configuration Namespaces in Visual Studio

The configuration namespaces in the .NET Framework include:

- System.Configuration - contains classes that are used to programmatically access .NET Framework configuration settings and handle errors in configuration files.
- System.Configuration.Assemblies - contains classes that are used to configure an assembly.
- System.Configuration.Install - provides classes that allow you to write custom installers for your own components.

In the Visual Studio Documentation

Configuring Applications Using Dynamic Properties
: Explains how you can store some of a component's property values to an external file and load them when the application runs. These dynamic properties allow you to make changes to property values, without recompiling the application, after your project has been deployed.

Introduction to the .NET Framework Class Library in Visual Studio
: Provides an overview of the namespaces in the .NET Framework.

Additional Information

Assemblies
: Provides information from the *.NET Framework Developer's Guide* about these building blocks of .NET Framework applications.

Exposing Management Data
: Demonstrates how to use **System.Configuration.Install** to create a management instrumentation instance class and publish an instance of the class to Windows Management Instrumentation (WMI).

Declaring and Accessing Custom Configuration Sections
: Demonstrates how to use **System.Configuration** to declare a custom section for your application to read at run time and retrieve settings from a configuration file.

Globalization and Localization Namespaces in Visual Studio

The namespaces in the .NET Framework for globalizing and localizing applications and components include:

- System.Globalization - contains classes that define culture-related information, including the language, the country/region, the calendars in use, the format patterns for dates, currency and numbers, and the sort order for strings.
- System.Resources - provides classes and interfaces that allow developers to create, store, and manage various culture-specific resources used in an application.
- System.Text - contains classes representing ASCII, Unicode, UTF-7, and UTF-8 character encoding.

In the Visual Studio Documentation

Introduction to International Applications in Visual Basic and Visual C#
Describes how to create an international-ready application.
Globalizing and Localizing Applications
Explains how to design and develop applications to distribute to a global audience.
Planning World-Ready Applications
Describes how to plan applications that will be distributed to an international market.
International Changes in Visual Basic .NET
Summarizes for users of Visual Basic 6.0 how locale, resources, and Unicode are handled.
Introduction to the .NET Framework Class Library in Visual Studio
Provides an overview of the namespaces in the .NET Framework.

Additional Information

Common Resource Concepts
Provides additional information about the .NET Framework globalization and localization namespaces.

Reflection Namespaces in Visual Studio

The reflection namespace in the .NET Framework is:

- System.Reflection - contains classes and interfaces that provide a managed view of loaded types, methods, and fields, with the ability to dynamically create and invoke types.

In the Visual Studio Documentation

Providing Metadata Descriptions About Your Component
Discusses the relationship between attributes for components you create and the **System.Reflection** namespace.
Attributes Tutorial (Visual C#)
Includes a section about accessing attributes through reflection.
Introduction to the .NET Framework Class Library in Visual Studio
Provides an overview of the namespaces in the .NET Framework.

Additional Information

Reflection Overview
Provides information from the *.NET Framework Developer's Guide* about reflection and how it is commonly used.

Visual Basic Runtime Namespace in Visual Studio

The **Microsoft.VisualBasic** namespace contains modules, enumerations, and attributes that provide the properties, functions, and constants you can use in Visual Basic .NET applications. Examples of functions in this namespace include:

- **Microsoft.Visual Basic.Interaction.MsgBox**
- **Microsoft.Visual Basic.Interaction.Shell**
- **Microsoft.Visual Basic.Strings.Mid**
- **Microsoft.Visual Basic.Collection.Count**

Examples of properties and constants include **Microsoft.Visual Basic.DateAndTime.TimeOfDay** and **vbCrLf**.

Examples of attributes include **VBFixedStringAttribute**, **VBFixedArrayAttribute**, and **ComClassAttribute**.

In the Visual Basic Documentation

Visual Basic Run-Time Library Members
  Provides a complete reference for the Visual Basic run-time elements.
Namespaces
  Describes how namespaces organize objects within assemblies.
Programming Element Support Changes Summary
  Provides information on programming elements that are replaced by methods in the .NET Framework class library.
References and the Imports Statement
  Discusses how to reference and use objects in assemblies.
Attributes
  Discusses how .NET attributes provide additional information about entities defined inside a Visual Basic .NET application

Additional Information

.NET Framework Class Library in Visual Studio
  Provides an overview of the namespaces in the .NET Framework.
Introduction to the .NET Framework Class Library in Visual Studio
  Explains namespaces and assemblies.

Globalizing and Localizing Applications

If you plan on distributing your application to an international audience, there are a number of things you'll need to keep in mind during the design and development phases. Even if you don't have such plans, a small effort up front can make things considerably easier should your plans change in future versions of your application. Services built into the .NET Framework make it easy to develop a single application that can adapt to different locales using managed development with Visual Basic or Visual C#.

Visual Studio .NET was designed from the start to make developing for an international audience easy by taking advantage of services built into the .NET Framework. The following topics will help introduce you to the internationalization features built into Visual Studio .NET.

In This Section

Introduction to International Applications in Visual Basic and Visual C#
: Introduces the concepts related to developing software for an international market using Visual Basic .NET or Visual C# .NET.

Localizing Applications
: Provides links to topics about customizing applications for a given culture.

Globalizing Applications
: Provides links to topics about creating applications that support multiple cultures.

Bi-directional Language Features in Visual Studio
: Discusses the bi-directional features in Visual Studio, which support Arabic, Hebrew, Farsi, Urdu, and other languages that are written right-to-left.

Related Sections

Deployment and Localization
: Describes deployment tools in Visual Studio that allow you to distribute multiple versions of your application for different locales.

Planning World-Ready Applications
: Provides background information on programming for an international audience.

Introduction to the .NET Framework Class Library
: Introduces the classes, interfaces, and value types that expedite and optimize the development process and provide access to system functionality.

Globalization and Localization Namespaces in Visual Studio
: Lists the major namespaces for this area and provides links to .NET Framework reference topics about the types (classes, structures, enumerations, delegates, and interfaces) in the namespaces, as well as links to topics about using them in Visual Basic and Visual C# applications.

System.Globalization Namespace
: Points out the classes in this namespace, which define culture-related information, including the language, the country/region, the calendars in use, the format patterns for dates, currency and numbers, and the sort order for strings.

System.Resources Namespace
: Points out the classes and interfaces in this namespace, which allows developers to create, store, and manage various culture-specific resources used in an application.

Decision Chart
: Provides a graphical, linked guide to decisions about designing your application.

Visual Basic Code Example Topics
: Lists quickstart-like code examples that help you perform common tasks in Visual Basic .NET.

Introduction to International Applications in Visual Basic and Visual C#

In Visual Studio .NET, there are two parts to creating a world-ready application: globalization, the process of designing applications that can adapt to different cultures, and localization, the process of translating resources for a specific culture. For an overview of globalization, localization, and related terminology, see Globalization and Localization Issues. For general information on designing applications for an international audience, see Planning World-Ready Applications.

The .NET localization model consists of a main assembly that contains both the application code and the fallback resources — strings, images, and other objects for the language in which the application is originally developed. Each localized application will have satellite assemblies, or assemblies which contain only the localized resources. Because the main assembly always contains the fallback resources, if a resource is not found in the localized satellite assembly, the ResourceManager will attempt to load it in a hierarchical manner, eventually falling back to the resource in the main assembly. The resource fallback system is explained in greater detail in Hierarchical Organization of Resources for Localization.

The project system for Windows Forms applications can generate resource files for both the fallback and each desired additional UI culture. The fallback resource file is built into the main assembly, and the culture-specific resource files are then built into satellite assemblies, one for each UI culture. When you build a project, the resource files are compiled from the Visual Studio XML format (.resx) to an intermediate binary format (.resources), which are then embedded in satellite assemblies.

The project system for both Windows Forms and Web Forms allows you to build resource files using an Assembly Resource File template, access the resources, and build your project. Satellite assemblies will be created along with the main assembly.

When a localized application executes, its appearance is determined by two culture values. (A *culture* is a set of user preference information related to the user's language, environment, and cultural conventions.) The UI culture setting determines which resources will be loaded. The UI culture is set as **UICulture** in Web.config files and page directives, and **CurrentUICulture** in Visual Basic or Visual C# code. The culture setting determines formatting of values such as dates, numbers, currency, and so on. The culture is set as **Culture** in Web.config files and page directives, **CurrentCulture** in Visual Basic or Visual C# code.

See Also

Globalizing and Localizing Applications | System.Globalization Namespace | System.Resources Namespace | Security and Localized Satellite Assemblies

Localizing Applications

Localization is the process of customizing your application for a given culture or locale. Localization consists primarily of translating the user interface.

In This Section

Walkthrough: Localizing Windows Forms
Provides a step-by-step example of how to localize a Windows Form.
Walkthrough: Localizing Web Forms Pages
Provides a step-by-step example of how to localize a Web Forms page.
Hierarchical Organization of Resources for Localization
Explains how localized resources are stored and accessed in Visual Studio.
Security and Localized Satellite Assemblies
Discusses signing satellite assemblies with public-private key pairs.
Version Numbers for Main and Localized Satellite Assemblies
Introduces the **SatelliteContractVersionAttribute** class, which determines which satellite assemblies work with an application's main assembly
Neutral Resources Languages for Localization
Introduces the **NeutralResourcesLanguageAttribute** class, which specifies the culture of the resources included in an application's main assembly.

Related Sections

Introduction to International Applications in Visual Basic and Visual C#
Discusses the concepts related to developing software for an international market using Visual Basic or Visual C#.
Globalizing Windows Forms
Provides links to topics about creating Windows applications that support multiple cultures.
Globalizing Web Forms Pages
Provides links to topics about creating Web applications that support multiple cultures.
Planning World-Ready Applications
Provides information on programming for an international audience, such as design issues and terminology.

Walkthrough: Localizing Windows Forms

The Visual Studio project system provides considerable support for localizing Windows Forms applications. There are two ways to generate resource files using the Visual Studio development environment: one is to have the project system generate the resource files for localizable UI elements such as text and images on the form. The resource files are then built into satellite assemblies. The second way is to add a resource file template and then edit the template with the XML Designer. A reason for doing the latter is to make localizable strings that appear in dialog boxes and error messages. You must then write code to access these resources.

This walkthrough topic demonstrates both processes in a single Windows Application project.

You can also convert a text file to a resource file; for more information, see Resources in Text File Format and Resource File Generator (Resgen.exe).

To have Visual Studio generate resource files for you

1. Create a new Windows Application named "WindowsApplication1". For details see Creating a Windows Application Project.
2. In the Properties window, set the form's **Localizable** property to **true**.

   The **Language** property is already set to **(Default)**.

3. Drag a **Button** control from the Windows Forms tab of the **Toolbox** to the form, and set its **Text** property to "Hello World".
4. Set the form's **Language** property to "German (Germany)".
5. Set the button's **Text** property to "Hallo Welt".
6. Set the form's **Language** property to "French (France)".
7. Set the button's **Text** property to "Bonjour le Monde". You can resize the button to accommodate the longer string, if necessary.
8. Save and build the solution.
9. Click the "Show All Files" button in Solution Explorer.

   The resource files appear underneath Form1.vb or Form1.cs. Form1.resx is the resource file for the fallback culture, which will be built into the main assembly. Form1.de-DE.resx is the resource file for German as spoken in Germany. Form1.fr-FR.resx is the resource file for French as spoken in France.

   In addition, you will see files appear named Form1.de.resx and Form1.fr.resx. Visual Studio automatically creates these files in order to work around a limitation in Visual SourceSafe having to do with adding new files to a project during a save operation. The .resx files are empty and contain no resources.

10. Press the F5 key or choose **Start** from the **Debug** menu.

    You will now see a dialog box with an English, French, or German greeting depending on the UI language of your operating system.

    > **Note** The UI language used in Windows is a function of the CurrentUICulture setting. If your copy of Windows has a Multilingual User Interface Pack (MUI) installed, you can change UI language in the Control Panel. For more information, see the Microsoft Globalization site. If you have no MUI installed, you can change the current UI culture programmatically, as explained next.

The following procedure shows you how to set the UI culture so the application displays your French resources. In real-world applications, you would not hard-code the UI culture in this way. The setting for the UI culture would depend instead on a user setting or an application setting.

To set the UI Culture to view specific resources

1. In the Code Editor, add the following code at the beginning of the module, before the Form1 declaration:

```
' Visual Basic
Imports System.Globalization
Imports System.Threading

// C#
using System.Globalization;
using System.Threading;
```

2. Add the following code. In Visual Basic, it should go in the **New** function, before calling the **InitializeComponent** function. In Visual C#, it should go in **Form1**() and also before calling the **InitializeComponent** function.

```
' Visual Basic
' Sets the UI culture to French (France)
Thread.CurrentThread.CurrentUICulture = New CultureInfo("fr-FR")

// C#
// Sets the UI culture to French (France)
Thread.CurrentThread.CurrentUICulture = new CultureInfo("fr-FR");
```

3. Save and build the solution.
4. Press the F5 key or choose **Start** from the **Debug** menu.

   Now the form will be always displayed in French. If you changed the size of the button earlier to accommodate the longer French string, notice that the button size has also been persisted in the French resource file.

To manually add resource files to the project and edit them

1. On the **Project** menu, click **Add New Item**.
2. In the Templates box, select the **Assembly Resource File** template. Type the file name "WinFormStrings.resx" in the **Name** box. The file WinFormStrings.resx will contain fallback resources in English. These resources will be accessed whenever the application cannot find resources more appropriate to the UI culture.

   The file is added to your project in Solution Explorer and automatically opens in the XML Designer in Data view.

3. In the **Data Tables** pane, select **data**.
4. In the **Data** pane, click an empty row and enter "strMessage" in the **name** column and "Hello World" in the **value** column.

   You do not need to specify the **type** or **mimetype** for a string; they are used for objects. The type specifier holds the data type of the object being saved. The MIME type specifier holds the base type (base64) of the binary information stored, if the object consists of binary data.

5. On the **File** menu, click **Save WinFormStrings.resx**.
6. Do steps 1-5 twice more to create two more resource files named "WinFormStrings.de-DE.resx" and "WinFormStrings.fr-FR.resx", with the string resources specified in the following table. The file WinFormStrings.de-DE.resx will contain resources that are specific to German as spoken in Germany. The file WinFormStrings.fr-FR.resx will contain resources that are specific to French as spoken in France.

resource file name	name	value
WinFormStrings.de-DE.resx	strMessage	Hallo Welt
WinFormStrings.fr-FR.resx	strMessage	Bonjour le Monde

To access the manually-added resources

1. In the Code Editor, import the `System.Resources` namespace at the beginning of the code module.

```
' Visual Basic
Imports System.Resources

// C#
using System.Resources;
```

2. In Design view, double-click the button to display the code for its **Click** event handler and add the following code. The ResourceManager constructor takes two arguments. The first is the root name of the resources — that is, the name of the resource file without the culture and .resx suffixes. The second argument is the main assembly.

   In this walkthrough, no namespaces are declared, so the first argument of the **ResourceManager** constructor can take the form of "ProjectName.ResourceFileRootName". However, in a real-world application, you would set the DefaultNamespace property. In that case, you would need to declare the resource manager with the fully qualified root name of the resource file, including its namespace. For example, if the default namespace is "MyCompany.MyApplication.MyComponent", the first argument of the **ResourceManager** constructor might be "MyCompany.MyApplication.MyComponent.WinFormStrings".

```
' Visual Basic
' Declare a Resource Manager instance
Dim LocRM As New ResourceManager("WindowsApplication1.WinFormStrings", GetType(Form1).Ass
embly)
' Assign the string for the "strMessage" key to a messagebox
MessageBox.Show(LocRM.GetString("strMessage"))

// C#
// Declare a Resource Manager instance
ResourceManager LocRM = new ResourceManager("WindowsApplication1.WinFormStrings",typeof(F
orm1).Assembly);
// Assign the string for the "strMessage" key to a messagebox
MessageBox.Show(LocRM.GetString("strMessage"));
```

Note By default, the **ResourceManager** object is case-sensitive. If you wish to do case-insensitive lookups so that "TXTWELCOME" retrieves the same resource as "txtWelcome", you can set the resource manager's IgnoreCase property to **true**. However, for performance reasons, it is best to always specify the correct case for your resource names. Doing case-insensitive resource lookups can cause performance problems.

3. Build and run the form. Click the button.

   The message box will display a string appropriate for the UI culture setting; or if it cannot find a resource for the UI culture, it will display a string from the fallback resources.

See Also

Setting the Culture and UI Culture for Windows Forms Globalization | Introduction to International Applications in Visual Basic and Visual C# | Walkthrough: Localizing Web Forms Pages | Globalizing and Localizing Applications | Security and Localized Satellite Assemblies

Walkthrough: Localizing Web Forms Pages

When you localize Web Forms pages, the Visual Studio project system does not automatically generate resource files. You must manually create and edit the XML resource files. This walkthrough topic demonstrates how to add a resource file template and then edit the template with the XML Designer to create resources in English, German, and German as spoken in Austria. This topic also discusses how to write code to access these resources.

You can also convert a text file to a resource file. For more information, see Resources in Text File Format and Resource File Generator (Resgen.exe).

To manually create and edit a resource file

1. Create a new ASP.NET Web Application named "LocProject". For details, see Creating Web Projects.

   Note The project must be named "LocProject"; some code later in this walkthrough relies on having that name.

2. On the **Project** menu, click **Add New Item**.
3. In the Templates box, select the **Assembly Resource File** template. Type the file name "strings.resx" in the **Name** box. The file strings.resx will contain fallback resources in English. These resources will be accessed whenever the application cannot find resources more appropriate to the UI culture.

   The file is added to your project in Solution Explorer and automatically opens in the XML Designer in Data view.

4. In the **Data Tables** pane, select **data**.
5. In the **Data** pane, click an empty row and enter "txtSearch" in the **name** column and "Search for the following text:" in the **value** column.

   You do not need to specify the **type** or **mimetype** for a string; they are used for objects. The type specifier holds the data type of the object being saved. The MIME type specifier holds the base type (base64) of the binary information stored, if the object consists of binary data.

6. Click another empty row and enter "txtWelcome" in the **name** column and "Welcome to Localization!" in the **value** column.
7. On the File menu, click **Save strings.resx**.
8. Do steps 2-7 twice more to create two more resource files named "strings.de.resx" and "strings.de-AT.resx", containing the string resources specified in the following table. The file strings.de.resx will contain German resources that are neutral — that is, they apply to any region where German is spoken. The file strings.de-AT.resx will contain German resources that are specific to Austria.

resource file name	name	value
strings.de.resx	txtSearch	Nach dem folgenden Text suchen:
strings.de.resx	txtWelcome	Willkommen zur Lokalisierung!
strings.de-AT.resx	txtWelcome	Grüßt's euch zur ASP.NET Lokalisierung!

   You can enter the accented characters in a variety of ways:

   - Copying and pasting from this help topic
   - Using a German keyboard layout
   - Using the character map application charmap.exe
   - Using the Windows On-Screen Keyboard in Windows 2000 and Windows XP

   For details about using a different keyboard layout, the character map application, or the on-screen keyboard, see Windows Help.

To access the resources

1. In the Code Editor, import the `System.Resources` and `System.Globalization` namespaces at the beginning of the code module. In Visual Basic, this code should occur before the class declaration for WebForm1. In Visual C#, this code should occur before the namespace declaration. The following examples show the statements in context.

```
' Visual Basic
Imports System.Resources
Imports System.Globalization

Public Class WebForm1
```

```
   Inherits System.Web.UI.Page

// C#
using System.Resources;
using System.Globalization;

namespace LocProject
```

2. Declare a variable of type **ResourceManager** as part of the page class:

```
' Visual Basic
Public Class WebForm1
   Inherits System.Web.UI.Page
   Protected LocRM As ResourceManager

// C#
namespace LocProject_CS
{
   public class WebForm1 : System.Web.UI.Page
   {
      protected ResourceManager LocRM;
```

3. Create an instance of the **ResourceManager** class by adding the following code to the **Page_Load** method in the WebForm1 code-behind class. The ResourceManager constructor takes two arguments. The first is the root name of the resources — that is, the name of the resource file without the culture and .resx suffixes. The second argument is the main assembly.

```
' Visual Basic
LocRM = New ResourceManager("LocProject.strings", GetType(WebForm1).Assembly)

// C#
LocRM= new ResourceManager("LocProject.strings", typeof(WebForm1).Assembly);
```

Note If you did not name your project "LocProject," you should substitute the name you did use as part of the root name parameter.

To display a static resource string in the page

1. In HTML view, add some static text from a resource file with the following HTML code. Place this code between the <form> and </form> tags.

```
<h3><%=LocRM.GetString("txtWelcome")%></h3>
```

Note By default, the **ResourceManager** object is case-sensitive. If you wish to do case-insensitive lookups so that "TXTWELCOME" retrieves the same resource as "txtWelcome", you can set the resource manager's IgnoreCase property to **true**. However, for performance reasons, it is best to always specify the correct case for your resource names. Doing case-insensitive resource lookups can cause performance problems.

To assign a resource string to the property of a control

1. In Design view, drag a **Button** control from the Web Forms tab of the Toolbox to your form.
2. In the Properties window, delete the value for the button's **Text** property
3. In the Code Editor, add the following code to the **Page_Load** method to retrieve the text for the button from the resource file.

```
' Visual Basic
Button1.Text = LocRM.GetString("txtSearch")

// C#
Button1.Text = LocRM.GetString("txtSearch");
```

4. In the Code Editor, add the following code to import the `System.Threading` namespace at the beginning of the code module.

```
' Visual Basic
' Import this namespace at the beginning of the code module
Imports System.Threading

// C#
// Import this namespace at the beginning of the code module.
using System.Threading;
```

5. Add the following code to the very beginning of the **Page_Load** method to specify that the Web Forms page should display resources and formatting according to the culture specified in the browser's Accept-Language header. This code is important because otherwise your Web application will default to the culture settings of the Web server, which may not be appropriate for the remote user viewing the Web Forms page.

```
' Visual Basic
' Set the culture and UI culture to the browser's accept language
Thread.CurrentThread.CurrentCulture = CultureInfo.CreateSpecificCulture(Request.UserLangu
ages(0))
Thread.CurrentThread.CurrentUICulture = New CultureInfo(Request.UserLanguages(0))

// C#
// Set the culture and UI culture to the browser's accept language
Thread.CurrentThread.CurrentCulture = CultureInfo.CreateSpecificCulture(Request.UserLangu
ages[0]);
Thread.CurrentThread.CurrentUICulture = new CultureInfo(Request.UserLanguages[0]);
```

6. Save the project.
7. On the **Build** menu, click the **Build LocProject** command.
8. Right-click the WebForm1.aspx file in Solution Explorer and choose **View in Browser**. If you are running on a German or Austrian operating system you will see the page in German or Austrian; otherwise you will see it in English.

   If you view the page first on a system configured for German and then on a system configured for Austrian, you will notice that the Welcome text has changed to the Austrian resources, but the text for the search button has not changed. This is an example of the resource fallback system: the Austrian resources only changed the welcome text, and the remaining text defaults to the neutral German resource.

See Also

Setting the Culture and UI Culture for Web Forms Globalization |
Introduction to International Applications in Visual Basic and Visual C# | Walkthrough: Localizing Windows Forms |
Globalizing and Localizing Applications

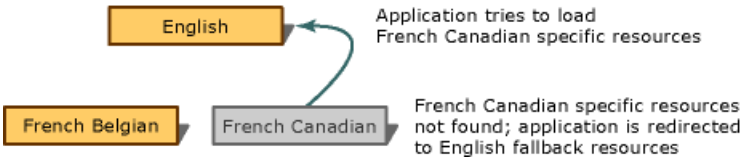
Hierarchical Organization of Resources for Localization

In Visual Studio .NET, localized resources (data such as strings and images appropriate to each culture) are stored in separate files and loaded according to the UI culture setting. To understand how localized resources are loaded, it is useful to think of them as organized in a hierarchical manner.

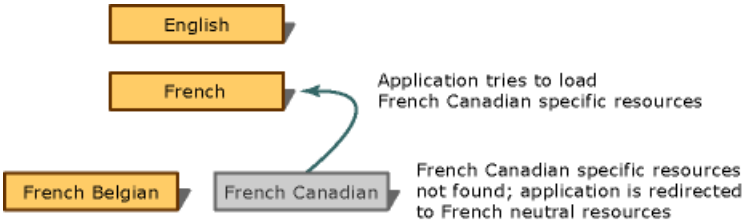
- At the top of the hierarchy sit the fallback resources for your default culture, for example English ("en"). These are the only resources that do not have their own file; they are stored in the main assembly.
- Below the fallback resources are the resources for any neutral cultures. A neutral culture is associated with a language but not a region. For example, French ("fr") is a neutral culture. (Note that the fallback resources are also for a neutral culture, but a special one.)
- Below those are the resources for any specific cultures. A specific culture is associated with a language and a region. For example, French Canadian ("fr-CA") is a specific culture.

If an application tries to load any localized resource, such as a string, and does not find it, it will travel up the hierarchy until it finds a resource file containing the requested resource.

The best way to store your resources is to generalize them as much as possible. That means to store localized strings, images, and so forth in resource files for neutral cultures rather than specific cultures whenever possible. For instance, if you have resources for the French Belgian ("fr-BE") culture and the resources immediately above are the fallback resources in English, a problem may result when someone uses your application on a system configured for the French Canadian culture. The system will look for a satellite assembly for "fr-CA", not find it, and load the main assembly containing the fallback resource, which is English, instead of loading the French resources. The following picture shows this undesirable scenario.



If you follow the recommended practice of placing as many resources as possible in a neutral resource file for the "fr" culture, the French Canadian user would not see resources marked for the "fr-BE" culture, but he or she would be shown strings in French. The following situation shows this preferred scenario.



See Also

Neutral Resources Languages for Localization | Security and Localized Satellite Assemblies | Localizing Applications | Globalizing and Localizing Applications | Setting the Culture and UI Culture for Windows Forms Globalization | Setting the Culture and UI Culture for Web Forms Globalization

Security and Localized Satellite Assemblies

If your main assembly uses strong naming, satellite assemblies must be signed with the same private key as the main assembly. If the public/private key pair does not match between the main and satellite assemblies, your resources will not be loaded. For information about strong names, see Strong Name Scenario. For more information on signing assemblies, see Signing an Assembly with a Strong Name.

In general, you may need to have your organization's signing group or an external signing organization sign with the private key. This is due to the sensitive nature of the private key: access is often restricted to a few individuals. You can use delayed signing during development. For more information, see Delay Signing an Assembly.

See Also

Assembly Security Considerations | Key Security Concepts |
Introduction to International Applications in Visual Basic and Visual C# | Localizing Applications |
Globalizing and Localizing Applications

Version Numbers for Main and Localized Satellite Assemblies

The main assembly uses versioning to determine whether to load resources. The SatelliteContractVersionAttribute class provides versioning support for a main assembly. Specifying the **SatelliteContractVersionAttribute** on an application's main assembly allows you to update and re-deploy a main assembly without updating its satellite assemblies. For your localized resources to be available, the satellite contract version of your main assembly must match the AssemblyVersionAttribute class of your satellite assemblies. You must specify an exact version number in the **SatelliteContractVersionAttribute**; wildcard characters such as "*" are not allowed. For more information, see Retrieving Resources in Satellite Assemblies.

The **SatelliteContractVersionAttribute** class allows you to update a main assembly without having to update your satellite assembly, or vice versa. When the main assembly is updated, its assembly version number is changed. If you want to continue using the existing satellite assemblies, change the main assembly's version number but leave the satellite contract version number the same. For example, in your first release your main assembly version may be 1.0.0.0. The satellite contract version and the assembly version of the satellite assembly will also be 1.0.0.0. If you need to update your main assembly for a service pack, you can change the assembly version to 1.0.0.1, while keeping the satellite contract version and the satellite's assembly version as 1.0.0.0.

If you need to update a satellite assembly but not your main assembly, you change the **AssemblyVersionAttribute** of the satellite assembly. Along with your satellite assembly, you will have to ship a policy assembly that states that your new satellite assembly is compatible with your old satellite assembly. For more information on policies, see How the Runtime Locates Assemblies.

The following code shows how to set the satellite contract version. The code can be placed in either a build script or in the AssemblyInfo.vb or AssemblyInfo.cs file.

```
' Visual Basic
<Assembly: SatelliteContractVersionAttribute("4.3.2.1")>

// C#
[assembly: SatelliteContractVersionAttribute("4.3.2.1")]
```

See Also

How the Runtime Locates Assemblies | Setting Assembly Attributes | Security and Localized Satellite Assemblies | Localizing Applications | Globalizing and Localizing Applications

Neutral Resources Languages for Localization

The NeutralResourcesLanguageAttribute class specifies the culture of the resources included in the main assembly. This attribute is used as a performance enhancement, so that the **ResourceManager** object does not search for resources that are included in the main assembly.

The following code shows how to set the neutral resources language. The code can be placed in either a build script or in the AssemblyInfo.vb or AssemblyInfo.cs file.

```
' Visual Basic
' Set neutral resources language for assembly.
<Assembly: NeutralResourcesLanguageAttribute("en")>

// C#
// Set neutral resources language for assembly.
[assembly: NeutralResourcesLanguageAttribute("en")]
```

See Also

ResourceManager Class | Introduction to International Applications in Visual Basic and Visual C# | Hierarchical Organization of Resources for Localization | Localizing Applications | Globalizing and Localizing Applications

Globalizing Applications

Globalization is the process of designing and developing a software product that functions for multiple cultures. The topics in this section apply to both Windows Forms and Web Forms pages.

In This Section

Culture-Specific Classes for Global Windows Forms and Web Forms
Discusses classes that format dates, time, numbers, currency, and other information according to the culture setting.
Globalizing Windows Forms
Provides links to topics about globalization of Windows Forms.
Globalizing Web Forms Pages
Provides links to topics about globalization of Web Forms pages.

Related Sections

Introduction to International Applications in Visual Basic and Visual C#
Discusses the essential concepts about developing software for an international market using Visual Basic or Visual C#.
Planning World-Ready Applications
Provides information on programming for an international audience, such as design issues and terminology.

Culture-Specific Classes for Global Windows Forms and Web Forms

Each culture has different conventions for displaying dates, time, numbers, currency, and other information. The System.Globalization namespace contains classes that can be used to modify how culture-specific values are displayed, such as **DateTimeFormatInfo**, **Calendar**, and **NumberFormatInfo**.

But most of the time you will use the culture setting, stored either in the application or in the **Regional Options** control panel, to automatically determine the conventions at run time and format the information accordingly. For more information on setting the culture, see Setting the Culture and UI Culture for Windows Forms Globalization or Setting the Culture and UI Culture for Web Forms Globalization. Classes that automatically format information according to the culture setting are called culture-specific. Some culture-specific methods are **IFormattable.ToString**, **Console.Writeline**, and **String.Format**. Some culture-specific functions (in the Visual Basic language) are **MonthName** and **WeekDayName**.

For example, the following code shows how you can use the **ToString** method to format currency for the current culture:

```
' Visual Basic
' Put the Imports statements at the beginning of the code module
Imports System.Threading
Imports System.Globalization
' Display a number with the culture-specific currency formatting
Dim MyInt As Integer = 100
Console.WriteLine(MyInt.ToString("C", Thread.CurrentThread.CurrentCulture))

// C#
// Put the using statements at the beginning of the code module
using System.Threading;
using System.Globalization;
// Display a number with the culture-specific currency formatting
int myInt = 100;
Console.WriteLine(myInt.ToString("C", Thread.CurrentThread.CurrentCulture));
```

If the culture is set to "fr-FR", you will see this in the output window:

```
100,00
```

If the culture is set to "en-US", you will see this in the output window:

```
$100.00
```

See Also

MonthName Function | WeekdayName Function | IFormattable.ToString Method | DateTimeFormatInfo Class | NumberFormatInfo Class | Calendar Class | Console.WriteLine Method | String.Format Method | Globalizing and Localizing Applications

Globalizing Windows Forms

Globalization is the process of designing and developing a software product that functions for multiple cultures. The topics in this section apply only to Windows Forms.

In This Section

Setting the Culture and UI Culture for Windows Forms Globalization
Explains how to set the two culture values for Windows Forms.
Encoding and Windows Forms Globalization
Describes full Unicode support and its implications.
International Fonts in Windows Forms and Controls
Explains when and how to select fonts for display of international characters on Windows Forms.
Displaying Right-to-Left Text in Windows Forms for Globalization
Provides instructions for displaying text that flows from right to left on Windows Forms.
Display of Asian Characters with the ImeMode Property
Introduces the **ImeMode** property, which is used to control the type of input a Windows Form or control accepts.

Related Sections

Introduction to International Applications in Visual Basic and Visual C#
Discusses the concepts related to developing software for an international market using Visual Basic .NET or Visual C# .NET.
Planning World-Ready Applications
Provides information on programming for an international audience, such as design issues and terminology.

Setting the Culture and UI Culture for Windows Forms Globalization

The two culture values of a Visual Basic or Visual C# application determine what resources are loaded for an application and how information like currency, numbers, and dates is formatted. The resources loaded are determined by the UI culture setting, and the formatting options are determined by the culture setting. The first place an application will look for culture values is the **CurrentCulture** and **CurrentUICulture** properties. They can be set in code as shown below.

The **CurrentCulture** property's default value is the system's User Locale, which is set in the **Regional Options** control panel. The **CurrentUICulture** property's default value is the system's UI Language, which is the language of your system UI. On Windows 2000 and Windows XP MultiLanguage Edition, the **CurrentUICulture** defaults to the current user UI language settings.

In some instances, you may want to have most of your application change according to the operating system's or user's culture settings, but also have a number or date that does not change. You can have culture-specific classes format the information with the invariant culture, which is associated with the English language but no particular region. For more information on these classes, see Formatting For Different Cultures and System.Globalization Namespace. For more information on the invariant culture, see CultureInfo.InvariantCulture Property. For a list of possible culture settings, see CultureInfo Class.

To set formatting options appropriate for a specific culture

1. Set the CurrentCulture and CurrentUICulture properties if you wish to override the settings of the user or operating system. In general, you would want to specify a culture so that every part of the application's user interface is appropriate to that culture. So you must set it before the **InitializeComponent** method is called.

```
' Visual Basic
' Put the Imports statements at the beginning of the code module
Imports System.Threading
Imports System.Globalization
' Put the following code before InitializeComponent()
' Sets the culture to French (France)
Thread.CurrentThread.CurrentCulture = new CultureInfo("fr-FR")
' Sets the UI culture to French (France)
Thread.CurrentThread.CurrentUICulture = new CultureInfo("fr-FR")

// C#
// Put the using statements at the beginning of the code module
using System.Threading;
using System.Globalization;
// Put the following code before InitializeComponent()
// Sets the culture to French (France)
Thread.CurrentThread.CurrentCulture = new CultureInfo("fr-FR");
// Sets the UI culture to French (France)
Thread.CurrentThread.CurrentUICulture = new CultureInfo("fr-FR");
```

   Note The culture value must always be a specific culture (such as "fr-FR"), not a neutral culture (such as "fr"). The reason is that a neutral culture such as "fr" can apply to all French-speaking cultures, and different currencies are used in France, Belgium, and Quebec.

2. Call formatting methods with the invariant culture for any strings that should appear unchanged regardless of the **CurrentCulture** property's value.

```
' Visual Basic
Dim MyInt As Integer = 100
Dim MyString As String = MyInt.ToString("C", CultureInfo.InvariantCulture)
MessageBox.Show(MyString)

// C#
int MyInt = 100;
string MyString = MyInt.ToString("C", CultureInfo.InvariantCulture);
MessageBox.Show(MyString);
```

See Also

Thread.CurrentCulture Property | Thread.CurrentUICulture Property | CultureInfo Class | Globalizing Applications | Globalizing and Localizing Applications | Formatting Issues

Encoding and Windows Forms Globalization

Windows Forms are entirely Unicode-enabled, meaning that each character is represented by a unique number, no matter what the platform, program, or language. For more information on Unicode, see the Web site of the Unicode consortium (http://www.unicode.org/).

Benefits of Unicode-enabled forms include the ability to work with scripts that are Unicode-only, such as Hindi. In addition, you can use multiple languages on a single form. In Unicode, all characters are two bytes long, so no special effort is needed to represent double-byte characters. You can also write a single set of code which will work on all platforms. This is a change from previous versions of Visual Basic, in which you had to write different code for different platforms such as Windows NT and Windows 98.

However, certain controls do not support Unicode in Windows 98 and Windows Millennium Edition. These controls, all of which inherit from the common control, will process data with the Windows code pages, as ANSI. These controls are: **TabControl**, **ListView**, **TreeView**, **DateTimePicker**, **MonthCalendar**, **TrackBar**, **ProgressBar**, **ImageList**, **ToolBar**, and **StatusBar**. The result of this is that you cannot display Unicode data in these controls on the listed platforms. For example, you cannot display Japanese characters on an English Windows 98 system.

See Also

Globalizing Windows Forms | Globalizing and Localizing Applications

International Fonts in Windows Forms and Controls

The recommended method of selecting fonts is to use font fallback wherever possible. Font fallback means that the system determines what script the character belongs to. To take advantage of this feature, do not set the **Font** property for your form or any other element. The application will automatically use the default system font, which differs from one localized language of the operating system to another. When the application runs, the system will automatically provide the correct font for the culture selected in the operating system.

There is an exception to the rule of not setting the font, which is for changing the font style. This might be important for an application in which the user clicks a button to make text in a text box appear in boldface. To do that, you would write a function to change the text box's font style to bold, based on whatever the form's font is. It is important to call this function in two places: in the button's **Click** event handler and in the **FontChanged** event handler. If the function is called only in the **Click** event handler and some other piece of code changes the font family of the entire form, the text box will not change with the rest of the form.

```
' Visual Basic
Private Sub MakeBold()
   ' Change the TextBox to a bold version of the form font
   TextBox1.Font = New Font(Me.Font, FontStyle.Bold)
End Sub

Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles
Button1.Click
   ' Clicking this button makes the TextBox bold
   MakeBold()
End Sub

Private Sub Form1_FontChanged(ByVal sender As Object, ByVal e As System.EventArgs) Handles My
Base.FontChanged
   ' If the TextBox is already bold and the form's font changes,
   ' change the TextBox to a bold version of the new form font
   If (TextBox1.Font.Style = FontStyle.Bold) Then
      MakeBold()
   End If
End Sub

// C#
private void button1_Click(object sender, System.EventArgs e)
{
   // Clicking this button makes the TextBox bold
   MakeBold();
}

private void MakeBold()
{
   // Change the TextBox to a bold version of the form's font
   textBox1.Font = new Font(this.Font, FontStyle.Bold);
}

private void Form1_FontChanged(object sender, System.EventArgs e)
{
   // If the TextBox is already bold and the form's font changes,
   // change the TextBox to a bold version of the new form font
   if (textBox1.Font.Style == FontStyle.Bold)
   {
      MakeBold();
   }
}
```

However, when you localize your application, the bold font may display poorly for certain languages. If this is a concern, you want the localizers to have the option of switching the font from bold to regular text. Since localizers are typically not developers and do not have access to source code, only to resource files, this option needs to be set in the resource files. To do this, you would set the **Font.Bold** property to **true**. This results in the font setting being written out to the resource files, where localizers can edit it. You then write code after the **InitializeComponent** method to reset the font based on whatever the form's font is, but using the font style specified in the resource file.

```
' Visual Basic
TextBox1.Font = New System.Drawing.Font(Me.Font, TextBox1.Font.Style)

// C#
textBox1.Font = new System.Drawing.Font(this.Font, textBox1.Font.Style);
```

See Also

Globalizing Windows Forms | Globalizing and Localizing Applications

Displaying Right-to-Left Text in Windows Forms for Globalization

Certain scripts, such as Arabic, are written from right to left. When you display these scripts, not only must the script flow in the reverse direction from the English language, but also visual alignment of form elements must be reversed. In this version, when you set a form's or control's **RightToLeft** property to **RightToLeft.Yes**, the left or right alignment is reversed for elements such as check boxes, vertical scroll bars on scrollable controls, and text in the title bars of forms. For a list of these effects, see Control.RightToLeft Property.

You do not need to change the values of properties such as the **TextAlign** value of a **TextBox** control: the value stays the same but the alignment is reversed.

To display right-to-left text

- Set the form's **RightToLeft** property to **RightToLeft.Yes**.

  All controls on the form will inherit the right-to-left setting, since the default value for the **RightToLeft** property of the **Control** class is **RightToLeft.Inherit**.

- If you wish to override the setting for a particular control, set that control's **RightToLeft** property to **RightToLeft.No**.

See Also

Control.RightToLeft Property | MainMenu.RightToLeft Property | ContextMenu.RightToLeft Property | RightToLeft Enumeration | Globalizing Windows Forms | Globalizing and Localizing Applications

Display of Asian Characters with the ImeMode Property

The **ImeMode** property is used by forms and controls to force a specific mode for an input method editor (IME). The IME is an essential component for writing Chinese, Japanese, and Korean scripts, since these writing systems have more characters than can be encoded for a regular keyboard. For example, you may want to allow only ASCII characters in a particular text box. In such a case you can set the **ImeMode** property to **Disable** and users will only be able to enter ASCII characters for that particular text box. The default value of the **ImeMode** property is **ImeMode.Inherit**, so if you set the property for a form, all controls on the form will inherit that setting. For more information, see Control.ImeMode Property and ImeMode Enumeration.

See Also

Globalizing Windows Forms | Globalizing and Localizing Applications

Globalizing Web Forms Pages

Globalization is the process of designing and developing a software product that functions for multiple cultures. The topics in this section apply only to Web Forms pages.

In This Section

Setting the Culture and UI Culture for Web Forms Globalization
Explains how to set the two culture values for Web Forms pages.
Selecting an Encoding for Web Forms Globalization
Describes how to set the encoding options for Web Forms pages.
Editing the Web.config File for Web Forms Globalization
Explains how to set encoding and culture in the globalization section of the Web.config file.
Displaying Right-to-Left Text Using HTML Tags for Globalization
Provides instructions for displaying text that flows from right to left on Web Forms pages.
HTML Layout Guidelines for Web Forms Globalization
Suggests best practices for designing Web Forms pages for globalization.

Related Sections

Introduction to International Applications in Visual Basic and Visual C#
Discusses the concepts about developing software for an international market using Visual Basic or Visual C#.
Planning World-Ready Applications
Provides background information on programming for an international audience, such as design issues and terminology.
Bi-directional Language Features in Visual Studio
Discusses the bi-directional features in Visual Studio, which support Arabic, Hebrew, Farsi, Urdu, and other languages that are written right-to-left.

Setting the Culture and UI Culture for Web Forms Globalization

By specifying a culture it is possible to use a set of common preferences for information like strings and date and number formats that correspond to users' cultural conventions. A Web Forms page has two culture values, **Culture** and **UICulture**. The **Culture** value determines or influences results of culture-dependent functions, such as displaying the date. The **UICulture** value determines how the resources are loaded for the form and is used for culture-specific lookup of resource data. The only purpose of the **UICulture** value is to indicate the language the resources should load, that is, determine which language the UI strings should use. The **Culture** value determines everything else — date formatting, number formatting, and so on.

The two culture settings do not need to have the same value. Depending on your application, it may be important to set them separately. An example is a Web auction site. The UI culture changes for each Web browser, but the culture stays constant, so prices are always displayed in the same currency and formatting.

The **Culture** value can only be set to specific cultures such as en-US or en-GB. This prevents the problem of identifying the correct currency symbol to use for "en", where en-US and en-GB have different currency symbols.

To set the Culture and UI Culture values

- In the Globalization section of the Web.config file, add lines similar to the code below. For more information on Web.config files, see Editing the Web.config File for Web Forms Globalization.

  ```
  culture="en-US"
  uiCulture="de"
  ```

  -or-

- In HTML view, add the **Culture** and **UICulture** values to the page directive.

  ```
  <%@ Page UICulture="de" Culture="en-US"...%>
  ```

  -or-

- In the Code Editor, add code to set the **CurrentCulture** and **CurrentUICulture** properties.

  The following example sets both the **CurrentCulture** and the **CurrentUICulture** properties to the culture specified in the browser's Accept-Language header. Note that the **CurrentCulture** property is set using the **CreateSpecificCulture** method, since this property must be set to a specific culture and not a neutral culture.

  ```
  ' Visual Basic
  ' Import these namespaces at the beginning of the code module
  Imports System.Globalization
  Imports System.Threading

  ' Set the culture and UI culture to the browser's accept language
  Thread.CurrentThread.CurrentCulture = CultureInfo.CreateSpecificCulture(Request.UserLangu
  ages(0))
  Thread.CurrentThread.CurrentUICulture = New CultureInfo(Request.UserLanguages(0))

  // C#
  // Import these namespaces at the beginning of the code module.
  using System.Globalization;
  using System.Threading;

  // Set the culture and UI culture to the browser's accept language
  Thread.CurrentThread.CurrentCulture = CultureInfo.CreateSpecificCulture(Request.UserLangu
  ages[0]);
  Thread.CurrentThread.CurrentUICulture = new CultureInfo(Request.UserLanguages[0]);
  ```

See Also

Globalizing Web Forms Pages | @ Page

Selecting an Encoding for Web Forms Globalization

Internally, the code behind Web Forms pages handles all string data as Unicode. You can set the **ResponseEncoding** attribute to set the encoding that the server uses to send data to the client, for example UTF-8. The Web Forms page also sets the **CharSet** attribute on the **Content-Type** of the HTTP header according to the value of **ResponseEncoding**. This enables browsers to determine the encoding without a meta tag or having to guess the correct encoding from the content.

The **RequestEncoding** attribute indicates the encoding the server will use to interpret data entered on the client and sent to the server. The **FileEncoding** attribute specifies the encoding that is used to interpret the data included in the aspx file. If the file encoding is specified in the Web.config file, the actual file must be saved in the same encoding. To select an encoding different from the system default, use the Advanced Save Options dialog box (available on the File menu).

> **Note** The name of your Web project may not display correctly on a remote user's system if the name of your Web project uses a double-byte character set (DBCS) that is not the language of the remote user's operating system. If you are using FrontPage server extensions, the project name must also match the language of the Web server's operating system in order to display correctly.

To specify encoding

- Set the appropriate encoding in the Web.config file. For more information on Web.config files, see Editing the Web.config File for Web Forms Globalization.

  -or-

- In HTML view, add the encoding values to the page directive.

```
<%@ Page ResponseEncoding="utf-8" RequestEncoding="utf-8"%>
```

> **Note** You can use the **Page** directive to specify any attribute except for **fileEncoding**, because it applies to the file itself. You can use the **globalization** section of the Web.config file to specify any attribute, including **fileEncoding**.

See Also

Globalizing Web Forms Pages

Editing the Web.config File for Web Forms Globalization

Web Forms configuration files, named Web.config, provide settings for every Web Forms page in the same directory as the configuration file. The settings are usually also inherited by subdirectories. Each file can contain a **globalization** section in which you can specify default encodings and cultures. Values are valid if they are accepted by the related Encoding and CultureInfo classes. For more information on the contents of the Globalization section, see <globalization> Element. For general information on Web.config files, see ASP.NET Configuration.

The Web.config file is case-sensitive, so specifying a value for "FileEncoding" will not have the desired effect, but specifying a value for "fileEncoding" will. For more information, see Format of ASP.NET Configuration Files.

To edit the Web.config file

- Set the values of **fileEncoding**, **requestEncoding** and **responseEncoding** to specify the way in which .aspx files, request data and response data are encoded. For more information on encoding values, see Selecting an Encoding for Web Forms Globalization

  If the file encoding is specified in the Web.config file, the actual file must be saved in the same encoding. To select an encoding different from the system default, use the Advanced Save Options dialog box (available on the File menu).

- Set the values of **culture** and **uiCulture**. For more information on culture values, see Setting the Culture and UI Culture for Web Forms Globalization.

```
<configuration>
   <system.web>
      <globalization
           fileEncoding="utf-8"
           requestEncoding="utf-8"
           responseEncoding="utf-8"
           culture="en-US"
           uiCulture="de-DE"
        />
   </system.web>
</configuration>
```

Note The attributes of the **globalization** section in the Web.config file can also be specified in the @ Page directive (with the exception of **fileEncoding**, which can only be specified in Web.config because it applies to the file itself). Settings in the Page directive are only valid for a specific page and override the settings of the Web.config file.

See Also

Globalizing Web Forms Pages

Displaying Right-to-Left Text Using HTML Tags for Globalization

When writing Web Forms pages, the best way to make text flow from right to left is to use the DIR (direction) attribute. It is usually placed in the <HTML> tag or the <BODY> tag, and then controls and HTML elements on the page inherit the specified direction.

Which tag the DIR attribute is applied to affects how the form appears:

- When direction is specified as RTL (right-to-left) in the <HTML> tag, the page will get the correct extended styles set so it displays as a right-to-left page on a bidirectional-enabled system and a vertical scrollbar appears on the left side.
- When direction is specified as RTL in the <BODY> tag, frames and captions will not inherit the right-to-left direction.

There are other ways to set the flow of text, including the <BDO> tag for bidirectional override, Unicode characters which have embedded directional information, and cascading style sheets. For more information, see Authoring HTML for Middle Eastern Content on the Global Software Development site on Microsoft.com (http://www.microsoft.com/globaldev/articles/mideast.asp).

To set the direction attribute in a Web Forms page

1. In Design view, select the document. In the Properties window, set the **dir** property to `ltr` or `rtl`.

   -or-

   In HTML view, edit the <HTML> or <BODY> tag to add the DIR attribute.

   ```
   <HTML dir="rtl">
   <!-- Or -->
   <body dir="rtl" >
   ```

2. In Design view, select any HTML server controls for which you wish to override the DIR attribute of the rest of the document. In the Properties window, set the **dir** property of those controls to `ltr` or `rtl`.

   -or-

   In HTML view, edit the tags of any HTML elements, HTML server controls, or Web Forms controls for which you wish to override the DIR attribute.

   ```
   <TABLE dir="ltr" >
   <asp:TextBox dir="ltr" >
   ```

See Also

ASP.NET Server Controls | dir Property | bdo Object | Globalizing Web Forms Pages | Globalizing and Localizing Applications

HTML Layout Guidelines for Web Forms Globalization

The best practice for HTML layout is to ensure that localizers only need to translate strings, since the user interface will automatically size all controls to fit. The step of resizing is removed from the localization process, thereby removing the errors and bugs that usually result. The following are recommendations for designing Web Forms for globalization.

- **Never use Cascading StyleSheet (CSS) properties for absolute positions.** Specifying absolute positions prevents elements from being automatically positioned and sized.

```
<!-- Do not do this -->
<DIV id = idFindWhatLabel style = "position: absolute; left: 0.98em; top: 1.2168em; width
: 4.8em; height: 1.2168em;">
```

- **Forms should take advantage of the available width and height.** There are two ways to do this. First, size major elements such as tables to width=100%. Second, size elements using CSS expressions based on the overall size of the form.

```
<!-- Table sized to take up entire width of the form-->
  <table width=100%>

<!-- div sized to take up half the width and height of the form -->
<div style='position:absolute; left: 0; top: 0;
  height: expression(document.body.clientHeight / 2);
  width: expression(document.body.clientWidth / 2); '>
```

- **Each control should be in a separate table cell.** This allows text to wrap independently and ensures correct alignment for cultures in which text layout flows from right to left.
- **Allow wrapping text.** Avoid the "nowrap" attribute for table cells that contain text. Only use this attribute when you want the text to stay on one line and there is enough room for the text to expand to accommodate any language.
- **Separate check boxes and radio buttons from their label text.** Place the labels for check boxes and radio buttons in separate cells from the controls. This allows the text to wrap correctly when the text grows longer. However, you should still design the form so that the text has a reasonable chance of staying on one line when it is translated.
- **Leave room for growth, and avoid fixed widths.** Text may expand when translated into other languages. A good rule of thumb is to leave room for 300% growth on short strings (fewer than 10 characters), 200% growth on medium strings (10-20 characters), and 100% growth on large strings (more than 20 characters).

  One acceptable solution is to set table cell width to a percentage of the form width so the cell can expand relative to the entire table. However, try to specify as few relative sizes (other than 100%) as possible, because you want the HTML layout engine to do most of the work for you.

```
<!-- Table cell sized to take up one quarter width of the table -->
 <td width=25%>
```

- **Size buttons in as few places as possible.** Often there is a set of buttons that should have the same size. Set this size in one place for the entire set of buttons, if possible. Allow room for the text to expand in list boxes and group boxes.
- **Set height as seldom as possible.** Do not set the cell height for any table cell or control that contains text. (You may set the line height for CSS elements.) Otherwise, the look of the page will change if the user sets the text size in the browser.
- **Do not use left or right align in HTML tags.** In general, avoid setting "align=left" or "align=right" for any table cell. These properties may interfere with the layout of forms for cultures in which text layout flows from right to left.
- **Avoid inline CSS values that may need to be changed.** Keep all CSS values that localizers may need to change in style sheets, rather than inline. This includes setting the font for a page.

See Also

Working With CSS Styles | Globalizing Web Forms Pages | Globalizing and Localizing Applications

Bi-directional Language Features in Visual Studio

Visual Studio includes support for *bi-directional* features, which support Arabic, Hebrew, Farsi, Urdu, and other languages that are written right-to-left. The following topics describe the features you can use to create applications for these languages.

In This Section

Bi-directional Features at Design Time in Visual Studio
Information on working in Visual Studio and how it does (and does not) support bi-directional features.
Arabic or Hebrew Text in Object Names
Information on how you can use bi-directional text to assign names to objects.
Saving and Opening Files with Encoding
How to set the character encoding in files and work with encoded files.
Creating Applications for Bi-directional Languages with Visual Studio
Provides overview information on how bi-directional text can be used in different types of Visual Studio applications.
Bi-directional Support for Windows Applications
Provides summary information of how bi-directional text is supported in Windows applications.
Creating Mirrored Windows Forms and Controls
Describes how to create Windows forms and controls that are properly reversed for use with bi-directional languages.
Creating Message Boxes for Bi-Directional Windows Applications
Describes how to display Windows message boxes correctly for bi-directional languages.
Bi-directional Support for Web Applications
Provides summary information of how bi-directional text is supported in Web applications.
Web Forms Controls in Bi-Directional Applications
Lists issues with using bi-directional text with Web Forms controls.

Related Topics

Globalizing and Localizing Applications
Lists topics in Visual Studio that pertain to creating international applications.
Localizing Applications
Lists topics that describe how to create applications with region-specific content.
Globalizing Applications
Lists topics that pertain to creating applications that can be localized.
Visual Studio .NET With Arabic Language Support
This whitepaper on the Microsoft.com Web site provides details on support for Arabic in applications created with Visual Studio. It includes Arabic-specific information about text formatting, calendar support, Windows Forms and controls, Web Forms, culture-specific classes in the .NET Framework, and Crystal Reports.

Bi-directional Features at Design Time in Visual Studio

Visual Studio allows you to work with bi-directional languages as you create applications. Visual Studio itself is not fully bi-directional, but it does allow you to create fully bi-directional applications.

> **Note** In order to enter and display bi-directional languages, you must be working with a version of Windows that is configured with the appropriate language. This can either be an English version of Windows with the appropriate language pack installed, or the appropriately localized version of Windows.

Features That Are Fully Supported

At design time in Visual Studio, you can use bi-directional languages in these ways:

- **Text entry** Visual Studio supports Unicode, so if your system is set to the appropriate locale and input language, you can enter text in Arabic or Hebrew. (Arabic support includes Kashida and Diacritics.)
- **Object names** You can use bi-directional languages to assign names to solutions, projects, files, folders, and so on. In code, you can use bi-directional languages for the names of variables, classes, object, attributes, metadata, and other elements. For details, see Arabic or Hebrew Text in Object Names.
- **File encoding** You can save and open files with a language-specific or Unicode encoding. For details, see Saving and Opening Files with Encoding.

Features with Limited or No Support

Other features common to bi-directional language applications are not fully supported in Visual Studio, or in some cases, not at all. These include:

- **Right-to-left reading order** By default, text-entry controls you use in Visual Studio use left-to-right reading order. In most cases, you can use standard Windows gestures to switch reading order. For example, you can press Ctrl+Right Shift to switch the Properties window to support right-to-left reading order for property values.

  However, right-to-left reading order is not supported everywhere in Visual Studio. Exceptions include:

  - Check boxes, drop-down lists, and other controls in Visual Studio dialog boxes always use left-to-right reading order.
  - The code editor (and text editor) does not support right-to-left reading order. You can enter text in a bi-directional language, but the reading order is always left-to-right.

See Also

Arabic or Hebrew Text in Object Names | Saving and Opening Files with Encoding | Globalizing and Localizing Applications | Visual Studio .NET With Arabic Language Support (this link goes to the Microsoft.com Web site)

Arabic or Hebrew Text in Object Names

You can use Arabic or Hebrew text to assign names to folders, variables, or other objects. When working with Arabic, you can use any Arabic characters including Kashida and Diacritics.

> **Note** Your Windows system and user locales must be set to the appropriate language in order to be able to enter Arabic or Hebrew into the dialog boxes displayed during installation.

The following elements can be named using Arabic or Hebrew and will be handled correctly in Visual Studio:

- Solution, project, and file names, including any folders you include in the project path. Solution Explorer will display solution and element names correctly.
- File contents. You can open or save files with Unicode encoding or with a selected code page.

  > **Note** The code editor is a special case. For details, see below.

- Data elements. Server Explorer will display these elements correctly and allow you to edit them.
- Elements copied to the Windows Clipboard.
- Attributes and metadata.
- Property values. You can use Arabic or Hebrew text in the Properties window. The window allows you to switch between right-to-left and left-to-right reading order using standard Windows keystrokes (Ctrl+RightShift for right-to-left, and Ctrl+LeftShift for left-to-right).
- Code and literal text. In the code editor (which is also the text editor), you can use Arabic or Hebrew to name classes, functions, variables, properties, string literals, attributes, and so on. However, the editor does not support right-to-left reading order; text always starts at the left margin.

  > **Tip** It is recommended that you place string literals in resource files instead of hard-coding them into your programs. For details, see Walkthrough: Localizing Windows Forms.

  > **Note** You must be consistent in how you refer to objects named in these languages. For example, if you use Kashida in naming an Arabic variable, you must always use Kashida when referring to that variable, or errors will result.

- Code comments. You can create comments in Arabic or Hebrew. You can also use these languages in the comment builder tool.

See Also

Bi-Directional Features at Design Time in Visual Studio | Saving and Opening Files with Encoding | Globalizing and Localizing Applications | Visual Studio .NET With Arabic Language Support (this link goes to the Microsoft.com Web site)

Saving and Opening Files with Encoding

You can save files with specific character encoding to support bi-directional languages. You can also specify an encoding when opening a file, so that Visual Studio displays the file correctly.

To save a file with encoding

1. From the File menu, choose **Save File As**, and then click the drop-down button next to the **Save** button.

   The **Advanced Save Options** dialog box is displayed.

2. Under **Encoding**, select the encoding to use for the file.
3. Optionally, under **Line endings**, select the format for end-of-line characters.

   This option is useful if you intend to exchange the file with users of a different operating system.

If you want to work with a file that you know is encoded in a specific way, you can tell Visual Studio to use that encoding when opening the file. The method you use depends on whether the file is part of your project.

To open an encoded file that is part of a project

1. In Solution Explorer, right-click the file and choose **Open With**.
2. In the **Open With** dialog box, choose the editor to open the file with.

   Many Visual Studio editors, such as the forms editor, will auto-detect the encoding and open the file appropriately. If you choose an editor that allows you to choose an encoding, the **Encoding** dialog box is displayed.

3. In the **Encoding** dialog box, select the encoding that the editor should use.

To open an encoded file that is not part of a project

1. On the **File** menu, point to **Open**, choose **File** or **File From Web**, and then select the file to open.
2. Click the drop-down button next to the **Open** button and choose **Open With**.
3. Follow Steps 2 and 3 from the preceding procedure.

See Also

Encoding and Windows Forms Globalization | Bi-Directional Features at Design Time in Visual Studio | Arabic or Hebrew Text in Object Names | Globalizing and Localizing Applications

Creating Applications for Bi-directional Languages with Visual Studio

You can use Visual Studio to create applications that correctly display text in languages written right-to-left, including Arabic and Hebrew. For some features, you can simply set properties. In other cases, you must implement features in code.

> **Note** In order to enter and display bi-directional languages, you must be working with a version of Windows that is configured with the appropriate language. This can either be an English version of Windows with the appropriate language pack installed, or the appropriately localized version of Windows.

Windows Applications

You can create fully bi-directional applications that include support for bi-directional text, right-to-left reading order, and mirroring (reversing the layout of windows, menus, dialog boxes, and so on). Except for mirroring, these features are available by default or as property settings. Mirroring is supported inherently for some features, such as message boxes. However, in other cases you must implement mirroring in code.

For details, see Bi-directional Support for Windows Applications.

Web Applications

Web applications rely on browsers for their user interface, so the degree of bi-directional support in a Web application is somewhat dependent on how well the user's browser supports those bi-directional features. In Visual Studio, you can create applications with support for Arabic or Hebrew text, right-to-left reading order, file encoding, and local culture settings.

Web services likewise support and receiving sending UTF-8 and Unicode text, making them suitable for applications involving bi-directional languages.

For details, see Bi-directional Support for Web Applications.

Console Applications

Console applications do not include text support for bi-directional languages. This is not directly a limitation of Visual Studio; it is a consequence of how Windows works with console applications.

See Also

Bi-directional Support for Windows Applications | Bi-directional Support for Web Applications | Globalizing Windows Forms | Localizing Applications

Bi-directional Support for Windows Applications

You can use Visual Studio to create Windows applications that support bi-directional languages such as Arabic and Hebrew. This includes standard forms, dialog boxes, MDI forms, and all the controls you can work with in these forms — that is, all the objects in the System.Windows.Forms.Control namespace.

Culture Support

Culture and UI culture settings determine how an application works with dates, times, currency, and other information. Support for culture and UI culture is the same for bi-directional languages as it is for any other languages. For details, see Culture-Specific Classes for Global Windows Forms and Web Forms.

RightToLeft Property

The base **Control** class (from which forms derive) includes a RightToLeft property that you can set to change the reading order of a form and its controls. If you set the form's **RightToLeft** property, by default controls on the form inherit this setting. However, you can also set the **RightToLeft** property individually on most controls. For details, see Displaying Right-to-Left Text in Windows Forms for Globalization.

The effect of the **RightToLeft** property can differ from one control to another. In some controls it only sets the reading order, as in the **Button**, **TreeView** and **ToolTip** controls. In other controls, the **RightToLeft** property changes both reading order and layout. This includes the **RadioButton**, **ComboBox** and **CheckBox** controls. The following table provides details on how the **RightToLeft** property affects individual Windows Forms controls.

Control/Component	Effect of RightToLeft property	Requires mirroring?
Button	Sets the RTL reading order	No
CheckBox	The check box is displayed on the right side of the text	No
CheckedListBox	All the check boxes are displayed on the right side of the text	No
ColorDialog	Not affected; depends on the language of the operating system	No
ComboBox	Items in combo box control are right-aligned	No
ContextMenu	Appears right-aligned with RTL reading order	No
DataGrid	Appears right-aligned with RTL reading order	No
DateTimePicker	Not affected; depends on the language of the operating system	No
DomainUpDown	Left-aligns the up and down buttons	No
ErrorProvider	Not supported	No
FontDialog	Depends on the language of the operating system	No
GroupBox	The caption is displayed right aligned. Child controls may inherit this property.	No
HScrollBar	Starts with the scroll box (thumb) right-aligned	No
ImageList	Not required	No
Label	Displayed right-aligned	No
LinkLabel	Displayed right-aligned	No
ListBox	Items are right-aligned	No
ListView	Sets the reading order to RTL; elements stay left-aligned	Yes
MainMenu	Displayed right-aligned with RTL reading order at run time (not at design time)	No
MonthCalendar	Not affected; depends on the language of the operating system	No
NotifyIcon	Not supported	No
NumericUpDown	Up and down buttons are left-aligned	No
OpenFileDialog	Not affected; depends on the language of the operating system	No
PageSetupDialog	Not affected; depends on the language of the operating system	No
Panel	Child controls may inherit this property	Yes
PictureBox	Not supported	No
PrintDialog	Not affected; depends on the language of the operating system	No
PrintDocument	The vertical scroll bar become left-aligned and the horizontal scroll bar starts from the left	No
PrintPreview	Not affect by this property	No
PrintPreviewDialog	Text displayed in a form's title bar is right-aligned	No

ProgressBar	Not affect by this property	Yes
RadioButton	The radio button is displayed on the right side of the text	No
RichTextBox	Control elements that include text are displayed from right to left with RTL reading order	No
SaveFileDialog	Not affected; depends on the language of the operating system	No
Splitter	Not supported	No
StatusBar	The text in the panels is right-aligned with RTL reading order	Yes
TabControl	Not affected by this property	Yes
TextBox	Displays text from right to left with RTL reading order	No
Timer	Not required	No
ToolBar	Not affected by this property	Yes
ToolTip	Sets the RTL reading order	No
TrackBar	The scroll or track starts from the right	No
TreeView	Sets the RTL reading order only	Yes
VScrollBar	Displayed on the left side rather than right side of scrollable controls	No

Encoding

Windows Forms support Unicode, so you can include any character set when creating them. However, not all Windows Forms controls support Unicode on all platforms. For details, see Encoding and Windows Forms Globalization.

GDI+

You can use GDI+ to draw text with right-to-left reading order. The DrawString method, which is used to draw text, supports a **StringFormat** parameter that you can set to the **DirectionRightToLeft** member of the StringFormatFlags enumeration in order to reverse the point of origin for the text.

Common Dialog Boxes

System tools such as the File Open dialog box are under the control of Windows. They inherit language elements from the operating system. If you are using a version of Windows with the correct language settings, these dialog boxes will work properly with bi-directional languages.

Similarly, message boxes go through the operating system and support bi-directional text. The captions on message box buttons are based on the current language setting. By default, message boxes do not use right-to-left reading order, but you can specify a parameter to change this when displaying the message box. For details, see
Creating Message Boxes for Bi-Directional Windows Applications.

Mirroring

Mirroring refers to reversing the layout of UI elements so that they flow from right to left. In a mirrored Windows Form, for example, the Minimize, Maximize, and Close buttons appear left-most on the title bar, not right-most.

Setting a form or control's **RightToLeft** property to **true** reverses the reading order of elements on a form, but it does not reverse the layout to be right-to-left — that is, it does not cause mirroring. For example, it does not move the Minimize, Maximize, and Close buttons in the form's title bar to the left side of the form. Similarly, some controls, such as the **TreeView** control, require mirroring in order to change their display to be appropriate for Arabic or Hebrew.

You cannot directly mirror a form or control. However, you can create mirroring in code. For forms, you can set an extended Window style to cause mirroring. For most controls that require mirroring, you can create a mirrored control by inheriting from the base control and setting the same Windows extended style. For details, see Creating Mirrored Windows Forms and Controls.

You can create mirrored versions of the following controls.

ListView	**Panel**	**StatusBar**
TabControl	**TabPage**	**ToolBar**
Treeview		

Some controls are sealed, so you cannot derive a new control from them. These include the **ImageList** and **ProgressBar** controls.

See Also

Creating Mirrored Windows Forms and Controls | Creating Message Boxes for Bi-Directional Windows Applications |

Bi-directional Support for Web Applications | Globalizing Windows Forms | Localizing Applications | Visual Studio .NET With Arabic Language Support (this link goes to the Microsoft.com Web site)

Creating Mirrored Windows Forms and Controls

Windows Forms do not directly support mirroring. However, the Windows operating system itself does. You can therefore create a mirrored form in Visual Studio and set a Windows-defined extended style so that a Windows Form displayed in a mirrored fashion.

To create a mirrored Windows Form or control

1. Create a new class for the form or control:
   - If you are creating a mirrored form, the class should derive from System.Windows.Forms.Form.
   - If you are creating a mirrored control, the class should derive from the control you want to mirror, such as System.Windows.Forms.TreeView or System.Windows.Forms.Panel.
2. Override the base class's CreateParams property. Because this is a read-only property, you only need to create the property's get method.
3. In the overridden property, create an instance of the System.Windows.Forms.CreateParams class and set its ExStyle property.

   The **ExStyle** property is a bit field in which specific bits control various style attributes of the window.
4. Using bit-wise OR operators, set the following bits in the **CreateParams** object's **ExStyle** property to **true**:
   - x400000 (WS_EX_LAYOUTRTL) This bit setting specifies that the form or control will have a right-to-left layout (hence, will be mirrored). Set this for both forms and controls.
   - x100000 (WS_EX_NOINHERITLAYOUT) This setting specifies that the form or control's layout is not inherited by child controls.
   - Mirroring for child controls of the mirrored form should be set using their RTL property or by other means.

   Tip The constants used for the bit-wise settings are a Windows convention, defined in the winuser.h file.

The following example illustrates how to create a Windows form that is mirrored.

```
' Visual Basic
Public Class MirroredForm
    Inherits System.Windows.Forms.Form
    Const WS_EX_LAYOUTRTL = &H400000
    Const WS_EX_NOINHERITLAYOUT = &H100000
    Protected Overrides ReadOnly Property CreateParams() _
        As System.Windows.Forms.CreateParams
        Get
            Dim CP As System.Windows.Forms.CreateParams = _
                MyBase.CreateParams
            If Not MyBase.DesignMode() Then
               CP.ExStyle = CP.ExStyle Or WS_EX_LAYOUTRTL Or _
                   WS_EX_NOINHERITLAYOUT
            End If
            Return CP
        End Get
    End Property
    ' Rest of Form code here
End Class

// C#
const int WS_EX_LAYOUTRTL = 0x400000;
const int WS_EX_NOINHERITLAYOUT = 0x100000;
protected override CreateParams CreateParams
{
    get
    {
        CreateParams CP = base.CreateParams;
        if (! base.DesignMode)
            CP.ExStyle = CP.ExStyle | WS_EX_LAYOUTRTL
                | WS_EX_NOINHERITLAYOUT;
        return CP;
```

```
    }
}
// Rest of form code here
```

In some situations, it is useful to create a property that allows you to turn mirroring on or off.

To create a property to control mirroring

1. Create a Boolean property to control mirroring. For details about creating properties, see Property Procedures (Visual Basic .NET) or Properties Tutorial (Visual C# .NET).
2. Optionally, include attributes that allow you to expose the property at design time. If you want to include attributes, you must also import the System.ComponentModel namespace. For details, see Attributes Overview.
3. In the property's set method, call the base class's OnRightToLeftChanged method so that derived classes are notified of the change to the property.
4. Override the base class's CreateParams property as described in the previous procedure. In the overridden property, set the extended Windows style if mirroring is true.

   The following example shows the relevant code from a mirrored TreeView control that implements a property called `mirrored`. The example includes some design-time attributes so that the `mirrored` property can be exposed in the Properties window.

```
' Visual Basic
Imports System.ComponentModel
Public Class MirroredTreeView
    Inherits System.Windows.Forms.TreeView
    Private _mirrored As Boolean = False
    <Description("Change to the right-to-left layout."), _
    DefaultValue(False), Localizable(True), _
    Category("Appearance"), Browsable(True)> _
    Public Property Mirrored() As Boolean
        Get
            Return _mirrored
        End Get
        Set(ByVal Value As Boolean)
            If _mirrored <> Value Then
                _mirrored = Value
                MyBase.OnRightToLeftChanged(EventArgs.Empty)
            End If
        End Set
    End Property

    Protected Overrides ReadOnly Property CreateParams() _
        As System.Windows.Forms.CreateParams
        Get
            Dim CP As System.Windows.Forms.CreateParams = _
                MyBase.CreateParams
            If Mirrored Then
                CP.ExStyle = CP.ExStyle Or &H400000
            End If
            Return CP
        End Get
    End Property

    ' Rest of control code here

End Class

// C#
public class MirroredTreeView : System.Windows.Forms.TreeView
{
```

```
    const int WS_EX_LAYOUTRTL = 0x400000;
    private bool _mirrored= false;
    [Description("Change to the right-to-left
       layout."),DefaultValue(false),
        Localizable(true),Category("Appearance"),Browsable(true)]
    public bool Mirrored
    {
        get
        {
            return _mirrored;
        }
        set
        {
            if (_mirrored != value)
            {
                _mirrored = value;
                base.OnRightToLeftChanged(EventArgs.Empty);
            }
        }
    }
}
protected override CreateParams CreateParams
{
    get
    {
        CreateParams CP;
        CP = base.CreateParams;
        CP.ExStyle = CP.ExStyle | WS_EX_LAYOUTRTL ;
        return CP;
    }
}
// Rest of control code here
```

See Also

Globalizing Windows Forms | Localizing Applications | Bi-directional Support for Windows Applications | Windows Forms Inheritance | Implementing Mirror-Aware Controls for Windows Application with Visual Studio .NET (this link goes to the Microsoft.com Web site)

Creating Message Boxes for Bi-Directional Windows Applications

By default, a message box appears with left-to-right reading order. You can create messages boxes with right-to-left reading order and mirroring by passing certain parameters when you display the message box.

To create bi-directional message boxes

- Call the **MessageBox** class's Show method, and in the *options* parameter, pass the logical OR combination of the **RtlReading** and **RightAlign** members of the MessageBoxOptions enumeration. (The **RightAlign** member makes sure that wrapped text is aligned properly.)

  > **Note** The captions of buttons on the message box (such as "OK") are established by the Windows system locale, and you cannot set these during the call.

  The following example shows how to display a message box with right-to-left reading order and mirroring:

```
' Visual Basic
MessageBox.Show("Text", "Caption", _
   MessageBoxButtons.OK, MessageBoxIcon.Question, _
   MessageBoxDefaultButton.Button1, MessageBoxOptions.RtlReading _
      Or MessageBoxOptions.RightAlign)

// C#
MessageBox.Show("Text", "Caption",
    MessageBoxButtons.OK, MessageBoxIcon.Question,
    MessageBoxDefaultButton.Button1,
    MessageBoxOptions.RtlReading
    | MessageBoxOptions.RightAlign);
```

See Also

Globalizing Windows Forms | Localizing Applications | Creating Mirrored Windows Forms and Controls | Bi-directional Support for Windows Applications

Bi-directional Support for Web Applications

Visual Studio allows you to create Web applications that support bi-directional languages. Web applications have some issues that do not arise with client-based applications. For example, because the user interface is a browser that is not necessarily under your control, when you create your application, you must be aware of how users will access it.

Object Names in Web Applications

If your copy of Windows is configured with the appropriate language, Visual Studio allows you to use a bi-directional language when creating solutions, projects, and Web Forms pages. However, if the application will be accessed by people who do not have the proper language configuration, the names will not be processed properly.

For example, if you create Web pages with Arabic names, people without the Arabic Windows language pack will not be able to enter those names or even display them properly.

It is therefore recommended that in Web applications, you name elements using text that will be processed properly on all computers.

Culture Support

Culture and UI culture settings determine how an application works with dates, times, currency, and other information. Support for culture and UI culture is the same for bi-directional languages as it is for any other languages. For details, see Culture-Specific Classes for Global Windows Forms and Web Forms.

Encoding

ASP.NET handles all text as Unicode, and therefore supports bi-directional languages inherently. However, you must still set encoding options to specify what encoding is used to exchange information with browsers, and what encoding to apply to the application's files. You can do this by setting the `responseEncoding`, `requestEncoding`, and `fileEncoding` options in the application's Web.config file. For details, see Editing the Web.config File for Web Forms Globalization. Alternatively, you can set encoding options for individual pages by setting the encoding attributes in the @ Page directive.

Right-To-Left Support

When you create Web Forms pages, you can set the **dir** (direction) attribute. This specifies whether you want the page to use uses left-to-right or right-to-left reading order. However, it is up to the user's browser to implement the appropriate behavior. For details, see Displaying Right-to-Left Text Using HTML Tags for Globalization.

Flow and Grid Layout

In Grid layout, controls are positioned on the page using x and y coordinates. When the page is set to use right-to-left reading order, the x and y coordinates use the top right corner as their origin. In some cases, the page displays in the browser somewhat differently than it does in Design view of the Web Forms Designer. Therefore, for greatest precision when laying out pages that will be displayed in right-to-left reading order, it is recommended that you use flow layout. For details, see Positioning HTML Elements in Design View.

Web Forms Controls

HTML and ASP.NET server controls support Unicode and bi-directional languages. For example, the controls support right-to-left reading order, and text-entry controls such as the **TextBox** control allow you to enter Arabic or Hebrew text. The **Calendar** control supports localized dates (such as Hijri for Arabic) and bi-directional text.

There are small issues with some controls when they are used with bi-directional languages. For details, see Web Forms Controls in Bi-Directional Applications.

See Also

Web Forms Controls in Bi-Directional Applications | Globalizing Web Forms Pages | Localizing Applications | Visual Studio .NET With Arabic Language Support (this link goes to the Microsoft.com Web site)

Web Forms Controls in Bi-Directional Applications

You can use HTML and ASP.NET Web controls in applications with bi-directional text.

Control Reading Order

By default, controls inherit the reading order from the page they are in (that is, from the setting of the page's DIR attribute). You can set the reading order for individual controls as well. HTML controls expose a **dir** property that you can set in the Properties window. For ASP.NET controls, you set the **dir** property in HTML view. For details, see Displaying Right-to-Left Text Using HTML Tags for Globalization.

Displaying Numerals

The **DropDownList** and **ListBox** controls always display Latin numerals, even if the regional settings normally display Hindi numerals.

Validation Controls

For the most part, the ASP.NET validation Web server controls support bi-directional languages. Note the following:

- The validation controls do not ignore diacritics for the Arabic language.
- In the **RegularExpressionValidator** control, add the \W specifier to support characters in bi-directional languages. For example, the following regular expression is used to validate an email expression:

  ```
  \w+([-+.]\w+)*@\w+([-.]\w+)*\.\w+([-.]\w+)*
  ```

  The corresponding regular expression for multilingual text would be:

  ```
  (\w+|\W+)([-+.](\w+|\W+))*@(\w+|\W+)([-.](\w+|\W+))*\.(\w+|\W+)([-.](\w+|\W+))*
  ```

  For more details about regular expressions, see .NET Framework Regular Expressions.

Clipping

Under some circumstances, controls might clip text when set to right-to-left reading order. You can overcome this problem by padding the control's display with a style setting such as the following:

```
style="PADDING-RIGHT: 5px"
```

See Also

Bi-directional Support for Web Applications | Globalizing Web Forms Pages | Localizing Applications | Visual Studio .NET With Arabic Language Support (this link goes to the Microsoft.com Web site)

Creating and Retrieving Resources

In addition to containing executable code, applications will frequently contain nonexecutable data in the form of resources. Resources represent embedded data, such as strings or images, which can be retrieved at run time and displayed in the user interface. A common use for resources is localization of Windows Forms applications. Although resource management is handled in localization by the automatic creation of satellite assemblies, you can also manually retrieve and display resources at run time. The topics in this section demonstrate how to create resource files and resource assemblies, and how to retrieve them at run time.

In This Section

Creating and Editing Resource Files with ResEditor
Explains how to use the ResEditor application to create and edit resource files.
Embedding Resource Files in Applications
Describes how to add a resource file to an application.
Creating Resource Assemblies
Describes how to create resource-assemblies that can be updated without needing to recompile the solution.
Retrieving Resources with the ResourceManager Class
Explains how to use the **ResourceManager** class to retrieve resources at run time.

Related Sections

Globalization and Localization Namespaces in Visual Studio
Provides links to topics that describe how to use automatic globalization and localization for Windows Forms.
Common Resource Concepts
Provides detailed information about additional resource-related classes in the .NET Framework.

Creating and Editing Resource Files with ResEditor

The .NET Framework includes a sample application called ResEditor that facilitates creating and editing resource files. The ResEditor application allows you to create binary resource files (.resources) as well as XML resource files (.resX).

Building ResEditor

ResEditor is provided as source code with the .NET Framework SDK. Before you can use ResEditor, you must build it using the supplied batch file.

To build the ResEditor Application

1. Start Windows, and then click the **Start** button, point to **All Programs**, then **Visual Studio .NET 2003**, then **Visual Studio .NET Tools**, then **Visual Studio .NET Command Prompt** to open the Visual Studio .NET command prompt.
2. At the command prompt, change the directory to the SDK\v1.1\Samples\Tutorials\resourcesandlocalization\reseditor folder as shown below:

   cd \Program Files\Microsoft Visual Studio .NET 2003\SDK\v1.1\Samples\Tutorials\resourcesandlocalization\reseditor

3. At the command prompt, type **Build**, then press the ENTER key to build the application.

   ResEditor is built and the executable is placed in the \Program Files\Microsoft Visual Studio .NET\FrameworkSDK\Samples\Tutorials\resourcesandlocalization\reseditor folder.

Creating Resource Files

After ResEditor has been built, you can use it to create resource files.

To create a resource file with ResEditor

1. In **Windows Explorer**, browse to the directory that contains the **ResEditor** executable file.
2. Double-click the **ResEditor** icon to start the application.
3. From the **Add** drop-down menu, choose the type of resource you want to add.
4. Type the name of the resource in the **Add** text box, then click the **Add** button to add the resource entry to the file.
5. In the main pane, click the cell next to the resource name to specify a value.

   - For **String** resources, type the appropriate string into the box.
   - For **Image** and other types of resources, browse to the appropriate file.

6. Repeat steps 2, 3, and 4 for each resource you want to add to your file.
7. On the **File** menu, click **Save As** to save your file. You can save your file as either a .resources file or a .resX file.

Editing Existing Resource Files

You can also use ResEditor to edit exisitng resource files. You can use ResEditor to edit either .resources files or .ResX files.

To edit an existing resource file with ResEditor

1. In **Windows Explorer**, browse to the directory that contains the **ResEditor** executable file.
2. Double-click the **ResEditor** icon to start the application.
3. On the **File** menu, click **Open**.
4. Browse to the appropriate resource file in the **Open a Resource File** dialog box.

   The resource file opens and the resources it contains are displayed in the main pane.

5. If you want to change the value of any resource, click the cell next to the resource name and specify the correct value.

   - For **String** resources, type the appropriate string into the box.
   - For **Image** and other types of resources, browse to the appropriate file.

6. If you want to rename any of the resources, do the following:
   a. Highlight the resource you want to rename by clicking it.
   b. Type the new name in the **Rename** text box.

   c. Click the **Rename** button to apply the new name.
7. If you want to delete a resource, highlight the resource by clicking it and choose **Delete** from the **Resources** menu.
8. When you have finished editing your resource file, choose **File** then **Save As** to save your file.

See Also

Retrieving Resources with the ResourceManager Class | Embedding Resource Files in Applications | Creating Resource Assemblies

Embedding Resource Files in Applications

After you have created resource files, it is easy to add them to your application. Binary resource files (.resources) or XML resource files (.resX) can be added directly to your project. When your project is compiled, the resources files will be compiled as well. You can retrieve embedded resources (resources that have been compiled into a .dll file) by using the **ResourceManager** class. For more information, see Retrieving Resources with the ResourceManager Class.

If you want to frequently update resources in a program without having to recompile the entire solution, create a resource assembly. For more information, see Creating Resource Assemblies.

To add a resource file to your project

1. Create the resource file. For more information, see Creating and Editing Resource Files with ResEditor.
2. From the **Project** menu, choose **Add Existing Item**.

   The **Add Existing Item** dialog box opens.

3. Browse to the resource file that you want to add to your project. This can be either a .resources file or a .resX file. Select the appropriate file.
4. From the **Build** menu, choose **Build Solution** to embed your resource file in your compiled project.

See Also

Retrieving Resources with the ResourceManager Class | Creating and Editing Resource Files with ResEditor | Creating Resource Assemblies

Embedding Image Resources in a Project

Managing image resources as separate files can introduce complexities at deployment time. If you like the convenience of packaging images with your assemblies, you can add images to your project as embedded resources.

The following procedures show how to embed an image in your project and display it on a form.

To add an image to your project as an embedded resource

1. On the **Project** menu, choose **Add Existing Item**.
2. Navigate to the image you want to add to your project. Click the **Open** button to add the image to your project's file list.
3. Right-click the image in your project's file list and choose **Properties**. The Properties window appears.
4. Find the **Build Action** property in the Properties window. Change its value to **Embedded Resource**.

At this point, you can build the project. Your image will be compiled into your project's assembly.

To load the image programmatically and display it on a form

1. Import the System.Reflection and System.IO namespaces.
2. Choose an appropriate place in your code to load the image. The following example uses the form's **Load** event handler.

   The example code first uses reflection to get a reference to the assembly that is currently running.

   Then the example opens a stream on the embedded image. The name used to refer to the image takes this form:

   <namespace>.<image name>.<format>

   where **namespace** is the namespace for the project, **image name** is the name of the image, and **format** is the encoding format (for example, "bmp" or "jpg"). The example uses the string "EmbeddingExample.image1.bmp" to load the image.

   Finally, the example creates a **Bitmap** object from the image stream, assigns it to a **PictureBox** control, sizes the form to fit the image, and adds the **PictureBox** control to the form.

```
' Visual Basic
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As _
System.EventArgs) Handles MyBase.Load

        Dim myAssembly As System.Reflection.Assembly = _ System.Reflection.Assembly.GetEx
ecutingAssembly()
        Dim myStream As Stream = _ myAssembly.GetManifestResourceStream("EmbeddingExample
.image1.bmp")
        Dim image As New Bitmap(myStream)

        Me.ClientSize = New Size(image.Width, image.Height)

        Dim pb As New PictureBox
        pb.Image = image
        pb.Dock = DockStyle.Fill
        Me.Controls.Add(pb)
End Sub

// C#
private void Form1_Load(System.Object sender, System.EventArgs e)
    {
        System.Reflection.Assembly myAssembly = System.Reflection.Assembly.GetExecutingAs
sembly();
        Stream myStream = myAssembly.GetManifestResourceStream("EmbeddingExample.image1.b
mp");
        Bitmap image = new Bitmap(myStream);

        this.ClientSize = new Size(image.Width, image.Height);
```

```
        PictureBox pb = new PictureBox();
        pb.Image = image;
        pb.Dock = DockStyle.Fill;
        this.Controls.Add(pb);
    }
```

At this point, you can build and run the project.

See Also

Embedding Resource Files in Applications | Resource File Naming Conventions

Creating Resource Assemblies

Resource assemblies are useful when you need to frequently update resources in a program without having to recompile the entire solution. By placing all of your resources into a resource-only assembly, you can update the assembly as needed without having to recompile your application. Resources compiled into resource assemblies can be retrieved using the **ResourceManager** class. For more information, see Retrieving Resources with the ResourceManager Class.

To create a resource-only assembly

1. Create a new application with the **Empty Project** template.
2. From the **Project** menu, choose **Add Existing Item** to add your resource files to your project.
3. In **Solution Explorer**, right-click your project and choose **Properties** from the shortcut menu to open the **Project Property Pages**.
4. In the **Output Type** drop-down menu, change the output type of your project to **Class Library**.

   This will cause your assembly to be compiled as a .dll file.

5. From the **Build** menu, choose **Build <your project>** where <your project> is the name of your project.

   Your resources are compiled into the assembly.

See Also

Retrieving Resources with the ResourceManager Class | Embedding Resource Files in Applications | Creating and Editing Resource Files with ResEditor | Assemblies

Retrieving Resources with the ResourceManager Class

You can use the ResourceManager Class to retrieve *embedded resources* (resources that have been compiled into an application or class library) at run time. Each instance of the **ResourceManager** class is associated with an assembly and manages retrieval of the resources embedded in that assembly.

To retrieve a resource

1. Create a reference to the assembly that contains the resources you want to access. If the resource-containing assembly has not been loaded, it must be loaded at this time.
2. Create an instance of **ResourceManager** class to retrieve the resources.
3. Specify the base name of the embedded file and provide a reference to the assembly that contains the resources.

   Note The base name of an embedded resource file is the name of the namespace that the resource file is embedded and the name of the file without any extensions. For example, the base name of a file named Resource1.resX in the myApplication namespace would be myApplication.Resource1.

4. Call the ResourceManager.GetString Method or ResourceManager.GetObject Method to retrieve your resources.

   - To retrieve **String** resources, call the **GetString** method.
   - To retrieve resources of other types, call the **GetObject** method. Resources retrieved by means of the **GetObject** method must be explicitly converted to the appropriate type.

Example

```
' Visual Basic .NET
' Gets a reference to the same assembly that
' contains the type that is creating the ResourceManager.
Dim myAssembly As System.Reflection.Assembly
myAssembly = Me.GetType.Assembly

' Gets a reference to a different assembly.
Dim myOtherAssembly As System.Reflection.Assembly
myOtherAssembly = System.Reflection.Assembly.Load("ResourceAssembly")

' Creates the ResourceManager.
Dim myManager As New _
   System.Resources.ResourceManager("ResourceNamespace.myResources", _
   myAssembly)

Dim myString As System.String
Dim myImage As System.Drawing.Image
myString = myManager.GetString("StringResource")
myImage = CType(myManager.GetObject("ImageResource"), _
    System.Drawing.Image)

// C#
// Gets a reference to the same assembly that
// contains the type that is creating the ResourceManager.
System.Reflection.Assembly myAssembly;
MyAssembly = this.GetType().Assembly;

// Gets a reference to a different assembly.
System.Reflection.Assembly myOtherAssembly;
myOtherAssembly = System.Reflection.Assembly.Load("ResourceAssembly");

// Creates the ResourceManager.
System.Resources.ResourceManager myManager = new
   System.Resources.ResourceManager("ResourceNamespace.myResources",
   myAssembly);

// Retrieves String and Image resources.
System.String myString;
System.Drawing.Image myImage;
```

```
myString = myManager.GetString("StringResource");
myImage = (System.Drawing.Image)myManager.GetObject("ImageResource");
```

See Also

Creating and Editing Resource Files with ResEditor | Embedding Resource Files in Applications | Creating Resource Assemblies | ResourceManager Class

Securing Applications Portal

Security is a watchword for application development, especially in applications that will be accessible to users across a network. To create a secure application, you must understand the security risks that your application is subject to. You must also be familiar with the techniques used to prevent unauthorized access to your application and ensure data integrity.

In the Visual Basic and Visual C# Documentation

Introduction to Web Application Security
Provides an overview of security for Web applications, describing what types of issues you need to think about when creating applications in Visual Studio.
Windows Forms Security
Describes the security model for Windows applications, including code-based security and security schemes from the operating system.
Security Considerations for ASP.NET Web Applications
Links to the topics that provide an introduction to security in Web applications.
Setting Web Permissions
Describes how to control who can gain access to your Web project files and what level of access individuals have to the application.
Code Security in Components
Explains the issues you must deal with when designing secure components: how to ensure that your code will run where it is deployed, how to protect your component against malicious use, how to request permissions for component code, and how to assure users of your component's source and authenticity.
Adding Imperative Security Checks to Components
Describes how to protect specific blocks of code by requiring appropriate permissions for them.
Adding Security Attributes to Components
Describes how to use attributes to request permissions for your assembly before any code members are modified.
Security Ramifications of Event Logs
Describes the access rights needed to perform common tasks with the **EventLog** component.
Message Queue Security
Presents an overview of the security features of Microsoft Message Queueing.
Specifying the Security Context for Services
Describes how to determine the security context in which your services will run.
Introduction to the EventLog Component
Describes how to interact with the Windows Security log, which tracks security changes and possible breaches.
Security and Localized Satellite Assemblies
Describes strong naming requirements for your localized applications.

Additional Information

Securability Overview
Discusses how to provide security to an application and its data.
Best Practices for Securability
Presents best practices for creating securable applications.
Securing Applications
Provides information from the *.NET Framework Developer's Guide* about classes and services that enable developers to write secure code and enable system administrators to customize the access that code has to protected resources.
Access Control
Presents and overview of what access controls lists (ACLs) are and their role in securing applications.
Code Access Security Basics
Defines code access security in .NET, and describes the main tasks involved in implementing and working with it.
Decision Chart
Provides a graphical, linked guide to decisions about designing your application.
Security Namespaces in Visual Studio
Lists the major namespaces for security and provides links to .NET Framework reference topics about the types (classes, structures, enumerations, delegates, and interfaces) in the namespaces, as well as links to topics about using them in Visual Basic and Visual C# applications.
.NET Framework Security
Provides information from the *.NET Framework Developer's Guide* about Web application security, code-access security, and role-based security.
Security Model

Describes the design choices you face when designing the architecture of your distributed application, including choices that affect authentication, authorization, impersonation, and database security.

Macros Security and Sharing Issues

Describes the main security considerations for macro development, including intellectual property protection and virus avoidance and protection.

Visual Basic Code Example Topics

Lists quickstart-like code examples that help you perform common tasks in Visual Basic .NET.

Troubleshooting Problems Portal

As you develop Visual Studio applications, you might run into problems or errors that you do not immediately know how to resolve. In the topics listed below, you can find solutions to problems that might arise during your development process.

Project Troubleshooting

Troubleshooting Web Projects
Describes solutions to common problems that may occur when creating or opening Web projects in Visual Studio .NET.
Troubleshooting Broken References
Describes solutions for various problems that you might encounter when adding references to files and components.
Web Access Failed Dialog Box
Provides help for error messages that may arise when creating and opening Web projects.

Language Troubleshooting

Visual Basic Compiler Errors
Discusses the Visual Basic compiler errors that occur when the compiler encounters problems in your code.
Visual Basic Run-Time Errors
Discusses the Visual Basic run-time errors that occur when an application attempts to perform an action that the system cannot execute.
Upgrading from Visual Basic 6.0
Describes the changes you need to be aware of when upgrading your applications to Visual Basic .NET.
Language Changes in Visual Basic
Describes the language-specific changes between Visual Basic 6.0 and Visual Basic .NET.

Data Troubleshooting

Troubleshooting Data Access in Visual Studio .NET
Suggests how to fix common problems that arise when working with data access in Visual Studio .NET.

Components and Component Authoring Troubleshooting

Troubleshooting FileSystemWatcher Components
Describes solutions for various situations you might encounter when working with the **FileSystemWatcher** component.
Troubleshooting MessageQueue Components
Describes the solutions for a situation you might encounter when working with the **MessageQueue** component.
Troubleshooting Control and Component Authoring
Explains how to fix common problems when creating your own control or component.
Troubleshooting Inherited Event Handlers in Visual Basic .NET
Explains how to fix common issues with event handlers in inherited components.

Web Applications Troubleshooting

Troubleshooting Web Projects
Describes solutions to common problems that may occur when creating or opening Web projects in Visual Studio .NET.
Web Access Failed Dialog Box
Provides help for error messages that may arise when creating and opening Web projects.

Windows Applications Troubleshooting

Troubleshooting: Debugging Windows Services
Describes how you can debug a Windows service application using all of the standard debugging functionality of Visual Studio.

Deployment Troubleshooting

Troubleshooting Deployment
Describes solutions for various problems that you might encounter when creating deployment projects and deploying applications.

Extensibility Troubleshooting

Troubleshooting Visual Basic and Visual C# Extensibility
Describes solutions to common problems that may occur when developing macros, add-ins, and wizards.

Upgrading Troubleshooting

Troubleshooting Applications Upgraded from Visual Basic 6.0
Describes some known issues that are not detected by the upgrading tools and how to address them.
Upgrading from Visual Basic 6.0
Describes the changes you need to be aware of when upgrading your applications to Visual Basic .NET.
Language Changes in Visual Basic
Describes the language-specific changes between Visual Basic 6.0 and Visual Basic .NET.
Upgrading Visual InterDev 6.0 Applications to Visual Studio .NET
Describes how to upgrade a Visual InterDev 6.0 project so you can take advantage of features provided by the .NET Framework.

Additional Information

Visual Basic Code Example Topics
Lists quickstart-like code examples that help you perform common tasks in Visual Basic .NET.
Getting Help from Microsoft Product Support Services
Describes various ways to get technical support.
Visual Studio .NET Software Requirements
Lists the features supported in various versions of the Windows operating system, as well as solutions for some common problems.
Visual Studio .NET Editions
Lists the features supported in the Professional, Enterprise Developer, Enterprise Architect, and Academic editions of Visual Studio.
Visual Basic Standard Edition Features
Lists the features supported in this edition of Visual Basic, which may explain why something is not working as expected.
Visual C# Standard Edition Features
Lists the features supported in this edition of Visual C#, which may explain why something is not working as expected.

Testing Scalability and Performance Portal

An important part of developing Visual Studio applications is to design and build it for optimal performance. This requires that you understand the performance implications of your application design. In addition, you must know how to test your application to get objective measures of its performance under a variety of conditions.

Visual Basic and C# Scalability and Performance Topics

Monitoring Performance Thresholds
Provides basic information on monitoring Microsoft Windows 2000 performance thresholds.
Tracing and Instrumenting Applications in Visual Basic and Visual C#
Provides information on how to use trace statements to monitor your application's performance.

Visual Studio Scalability and Performance Topics

Scalability
Presents information on designing your application to allow for maximum increase in capacity as resources change, testing your application's scalability, and following best practice recommendations.
Performance
Presents information on designing your application for maximum performance, testing your application's performance, and following best practice recommendations.
Testing for Performance
Describes how to test to determine whether your application meets the performance guidelines you have set.
Testing for Reliability
Describes how to test to determine whether your application meets the reliability guidelines you have set.
Testing for Scalability
Describes how to test to determine whether your application meets the scalability guidelines you have set.
Instrumentation Guidelines
Describes how to use Visual Studio Analyzer to help track your application's performance.

Accessing Data

Most Visual Basic and Visual C# applications revolve around reading and updating information in databases. To allow data integration in distributed, scalable applications, Visual Studio .NET provides support for a new generation of data access technology: ADO.NET.

In This Section

Data Walkthroughs
Lists topics that illustrate how to perform common data-related tasks.
Code for Data Access
Provides code examples for common data tasks.
Troubleshooting Data Access in Visual Studio .NET
Suggests how to fix common problems that arise when working with data access in Visual Studio .NET.
Introduction to Distributed Applications and Data Integration
Provides a general introduction to the concepts of data access in applications you create with Visual Basic and Visual C#.
Connecting to Data Sources with ADO.NET
Provides topics about creating and working with connection objects in Visual Studio.
ADO.NET Data Adapters
Provides topics about defining the SQL statement or stored procedure calls that you use to fill datasets and write data back to the database.
ADO.NET Datasets
Provides topics about what datasets are and how to use them.
Filtering and Sorting in Datasets
Provides topics about viewing the data in a dataset in different ways by selecting subsets of records or sorting.
Relationships in ADO.NET Datasets
Provides topics about working with dataset tables that have a parent-child relationship.
Performing Database Operations Directly
Provides topics about how to execute SQL statements and stored procedures directly and work with the results they return.
Dataset Updates in Visual Studio .NET
Provides topics about making changes in datasets and then writing those changes back to a data source.
XML Schemas and Data
Provides topics about creating XML schemas that describe the structure of datasets.

Related Sections

Data Design
Provides topics that discuss the fundamental concepts of designing data storage and access for applications.
Visual Basic Code Example Topics
Lists quickstart-like code examples that help you perform common tasks in Visual Basic .NET.
Decision Chart
Provides an interactive guide to help you choose which approach or technology to use, as well as links to more information.
What's New in Data
Discusses data access features that are new in Visual Studio .NET.
Windows Forms Data Architecture
Provides topics on how data binding works in Windows Forms and how to bind controls to a data source.
Data Access in Web Forms Pages
Provides topics on the issues associated with data binding in Web Forms pages and how to bind controls to a data source.
Creating and Filling Datasets Example Topics
Presents a series of specific example code topics for common tasks in creating and filling datasets with Visual Basic .NET.
Manipulating Data Example Topics
Presents a series of specific example code topics for common tasks in working with data within your application using Visual Basic .NET.
Data Access Technologies
Describes factors to consider when deciding between ADO.NET, ADO, and OLE DB.
Debugging SQL
Provides information on debugging SQL Server stored procedures, functions, and triggers.
Common Tasks in Visual Basic and Visual C# (Data)
Lists the most common data tasks and provides links to more information.
Data Namespaces in Visual Studio
Lists the major namespaces for data and XML and provides links to .NET Framework reference topics about the types (classes,

structures, enumerations, delegates, and interfaces) in the namespaces, as well as links to topics about using them in Visual Basic and Visual C# applications.

XML in Visual Studio

Provides links to sections of the Visual Studio documentation that discuss XML.

Overview of Visual Database Tools

Introduces the Query Designer and other data tools.

Crystal Reports

Provides links to the Crystal Reports for Visual Studio .NET documentation.

Performing Transactions

Introduces transactions for users of the .NET Framework SDK.

Modeling Your Application and Data

Provides an overview of how you identify and implement the data and process requirements for your application, including an overview of what tools you can use.

Deploying Applications and Components

Provides links to procedural, conceptual, and step-by-step topics about component deployment.

Data Walkthroughs

These step-by-step topics cover common data and XML scenarios.

> **Tip** If you are new to ADO.NET, try one of the Windows Forms or Web Forms data walkthroughs before trying one of the general data walkthroughs.

In the Visual Basic and Visual C# Documentation

Windows Forms Data-Access Walkthroughs

Simple Data Access in a Windows Form
Describes the basic steps required for accessing SQL Server data via a dataset in a read-write Windows Form.
Displaying Data in a Windows Form Using a Parameterized Query
Describes how to create a dataset containing only selected records, based on criteria that users provide at run time in a Windows Form.
Creating a Master-Detail Windows Form
Describes how to create a parent-child relationship in a dataset and display related records in a Windows Form.

Web Forms Data-Access Walkthroughs

Displaying Data in a Web Forms Page
Illustrates a basic data-access scenario, providing step-by-step instructions for how to create a Web Forms page that displays data in a grid control.
Creating Read-Only Data Access in a Web Forms Page
Describes how to use an ADO.NET data command and data reader to provide optimized access to read-only data in a Web Forms page.
Updating Data Using a Database Update Query in Web Forms
Describes the basic techniques of creating a Web Forms page that uses a data command to update the database.
Using a DataGrid Web Control to Read and Write Data
Gives step-by-step details on how to use a grid control to allow users to view and edit data.
Creating Paged Data Access Using a Web Forms Page
Describes how to create a Web Forms page that allows users to view data records a few records at a time, paging back and forth to view earlier and later records.

General Data Walkthroughs

Creating a Distributed Application
Illustrates how to create an application with a middle-tier component (an XML Web service) that is used to get and update data.
Creating a Dataset with Tables, Keys, and One-to-Many Relationships
Describes the steps to create a dataset and use it to validate data in a one-to-many or keyed relationship.
Handling a Concurrency Exception
Describes how to use the **DBConcurrencyException** object to identify concurrency exceptions and the actual record that caused the error.
Mapping Data Source Tables to Dataset Tables
Describes how to load data from a database into a dataset based on a different schema using data adapter table and column mappings.

XML Data and XML Schema Walkthroughs

Creating an XML File with an Associated XML Schema
Describes how to create an XML file and a schema, associating the two, and working on the XML file in the designer's XML view.
Creating an XML Schema with the XML Designer
Describes the steps to create an XML purchase order schema as part of a Windows application.
Reading XML Data into a Dataset
Shows how to load XML data into a dataset, display it in a Windows Forms **DataGrid** control, then display an XML Schema for the XML file in a text box.
Displaying an XML Document in a Web Forms Page using Transformations
Gives step-by-step details on how to read an XML file and display it in different ways on a Web Forms page.

Additional Information

Samples and Walkthroughs

Provides access to all of the sample applications and step-by-step topics in Visual Studio.

Accessing Data

Provides links to information about working with ADO.NET data in Visual Basic and Visual C#.

Walkthroughs in Visual Basic and Visual C#

Shows a complete list of walkthroughs related to managed development in Visual Basic and Visual C#.

Visual Basic Code Example Topics

Shows quickstart-like code examples that help you perform common tasks in Visual Basic .NET.

Code for Data Access

This topic is designed to help you find code that demonstrates how to perform common data-programming tasks in Visual Basic .NET. The following categories are represented in the list below:

• Datasets • Data Views • Updating Datasets and Data Sources	• Data Commands • Concurrency • Relations • Related Code Examples

You can also find sample code for data access in these locations:

- Creating and Filling Datasets Example Topics
- Manipulating Data Example Topics
- Visual Basic Sample Applications
- Visual Basic QuickStarts and Other Samples
- Accessing Data with ADO.NET

Datasets

Example of...

Accessing columns in datasets

Filling datasets

Filling datasets with a parameterized query

Creating a simple Windows application that uses a dataset

Creating a simple Windows application that uses a parameterized query

Creating a simple Web application that uses a dataset

Creating a simple distributed application that exchanges information in datasets

Creating a simple application that maps data tables to a dataset

Sorting a data table in a dataset

Filtering (selecting) a data table in a dataset

Data Views

Example of...

Sorting a dataset using a data view

Filtering (selecting) in a dataset using a data view

Finding records in a data view

Reading records using a data view

Updating records using a data view

Inserting records using a data view

Deleting records using a data view

Working with multiple data views

Updating Datasets and Data Sources

Example of...

Updating existing dataset records

Creating a simple Windows application that updates a database

Creating a simple Web application that updates a database

Creating a simple Web application that updates a database using a grid

Inserting records into a dataset

Deleting records from a dataset

Merging datasets

Turning off dataset constraints

Committing dataset changes

Sending dataset changes to the data source

Handling data-update errors

Data Commands

Example of...

Creating parameterized queries

Creating a simple Web application that displays information in a grid

Executing queries that return a result set

Creating a simple Web application that displays paged data

Executing update, insert, or delete queries

Executing database (DDL) commands

Executing commands that return a scalar value

Concurrency

Example of...

Creating a simple application that implements concurrency checking

Handling concurrency errors

Relations

Example of...

Working with data in related tables

Creating a simple Windows application that displays master-detail information

Related Code Examples

Accessing Data (Web Forms)

Accessing Data (Windows Forms)

DataGrid Control (Web Forms)

DataGrid Control (Windows Forms)

Code Examples for Visual Basic Programmers

Troubleshooting Data Access in Visual Studio .NET

This topic lists the following common issues that arise when working with data access in Visual Studio .NET.

Cannot Access a Data Base at Run Time that I Can Access at Design Time

After Reconfiguring a Data Adapter, Filling a Dataset Throws an Exception

Getting the 'Connection property has not been initialized' Error

Incorrect or Missing Output Parameters are Returned When Executing a SQL Command

Getting the 'The .NET Data SQL Provider (System.Data.SqlClient) requires Microsoft Data Access Components (MDAC) version 2.6 or later' Error

Getting an Unexpected Exception When Attempting to Commit or Roll Back a Transaction

Getting an InvalidOperationException When Attempting to Remove Tables from a Dataset

Cannot Access a Database at Run Time that I Can Access at Design Time

This is most likely the result of the user name and password passed at run time being denied access to the database. Resolution depends on the authentication and authorization methods being used in the database, as well as the connection string.

Some common configurations that cause this problem are:

- Selecting the **Use a specific username and password** option and not selecting the **Allow saving password** option when completing the Data Link Properties Dialog Box. In this scenario you are prompted for a username and password when attempting to connect to the data source at design time. For run-time connectivity, you need to incorporate functionality into your application to pass the proper username and password to the data source.
- Not setting up permission for the ASPNET user in your data source. For information on how ASP.NET requests permissions to data, see Access Permissions for Web Applications.

Things to check:

- Verify the username and password that your application is using have proper permissions to access the data source.
- Verify the user has the correct permission settings in the database.
- Verify there is network connectivity between the machine running the application and the data source.

After Reconfiguring a Data Adapter, Filling a Dataset Throws an Exception

Reconfiguring a data adapter does not automatically regenerate the dataset. The dataset needs to be synchronized with the new data-adapter configuration. Regenerating the dataset should fix this problem.

To regenerate a Dataset from a configured data adapter

1. Select the form that contains the data adapter and open it in **Design** view.
2. On the **Data** menu, click **Generate Dataset**.

   The Generate Dataset Dialog Box opens.

3. In the **Choose which table(s) to add to the dataset** area, select the table associated with the reconfigured data adapter.
4. Click **OK**.

The following table provides links to related information.

For more information on...	See...
DataSet Class	Introduction to Datasets
DataAdapter Class	Introduction to Data Adapters

Getting the 'Connection property has not been initialized' Error

This is most likely the result of command objects that no longer contain a reference to a valid data connection object. This is typically caused by deleting and recreating an existing connection object.

Applications with Data Adapters

Reconfigure the data adapter to set the **Connection** property for the adapter's **Command** objects. For more information about data adapters, connections, and commands, see Using .NET Data Providers to Access Data.

To reconfigure a data adapter

1. Select the form that contains the data adapter(s) and open it in **Design** view.
2. Select the data adapter in the component tray.

3. On the **Data** menu, click **Configure Data Adapter** .

   The Data Adapter Configuration Wizard opens.

4. Complete the wizard. For more information, see Data Adapter Configuration Wizard.
5. Repeat this process from step 2 for each affected data adapter.

Applications with Individual Command Objects

Set the **Connection** property to a valid connection object.

To set the Connection property of a Command object

1. Select the form that contains the command(s) and open it in **Design** view.
2. Select the command object in the component tray.
3. Set its **Connection** property to a valid connection object.
4. Repeat this process from step 2 for each affected command.

Incorrect or Missing Output Parameters Are Returned When Executing a SQL Command

This most likely results from incorrect setting of the parameter's **Direction** property. The parameter's **Direction** property is set to a value defined in the ParameterDirection Enumeration.

- Verify the parameter's **Direction** property is set to a value that that can receive data (Output for output parameters).
- Verify the parameter's data type is the same as the data type of the expected return value.

> **Note** Output parameters are returned at the end of the data stream, so if you are using the **DataReader** object, you must close it or read to the end of the data before the output parameters are visible.

For more information on output parameters not being returned, as well as a code sample, search for Q308051 on the Microsoft Product Support Services Web site (http://support.microsoft.com). Select **Article ID** as the Search option.

Getting the 'The .NET Data SQL Provider (System.Data.SqlClient) requires Microsoft Data Access Components (MDAC) version 2.6 or later' Error

The Microsoft .NET Framework SDK and the .NET Framework redistributable package do not include the MDAC installation. All .NET applications that use data-access functionality require MDAC 2.6 or later (MDAC 2.7 is recommended). The latest version of MDAC is available as a download from the Microsoft Web site (http://www.microsoft.com).

Because Visual Studio .NET installs MDAC 2.7 by default, this error is most likely to occur when deploying to a machine that does not have Visual Studio .NET installed.

When you deploy your application, you can have the setup check the version of MDAC on the computer being deployed to. For more information, see Adding a Launch Condition for Microsoft Data Access Components.

Getting an Unexpected Exception When Attempting to Commit or Roll Back a Transaction

When an error at the data source causes a transaction to be immediately rolled back, one of the following errors may be raised in your application:

- "The COMMIT TRANSACTION request has no corresponding BEGIN TRANSACTION"
- "The ROLLBACK TRANSACTION request has no corresponding BEGIN TRANSACTION"

Although this behavior is by design, it is always best to call methods that access external data sources (such as the **Commit** and **Rollback** methods) from within a **Try...Catch** statement. For more information, see Using the Try/Catch Block to Catch Exceptions.

Getting an InvalidOperationException When Attempting to Remove Tables from a Dataset

When a dataset is bound to a XmlDataDocument calling the **Clear** method throws the following exception:

System.InvalidOperationException: Cannot add or remove tables from the DataSet once the DataSet is mapped to a loaded XML document.

Clearing the data from a dataset bound to a **XmlDataDocument** object requires navigating the table and removing the individual DataRow objects. For more information on removing data rows from a data table, see Deleting Records in a Dataset and DataRowCollection.RemoveAt Method.

See Also

Accessing Data | Overview of ADO.NET

Introduction to Distributed Applications and Data Integration

Virtually all distributed applications revolve around reading and updating information in databases. To allow data integration in distributed, scalable applications, Visual Studio .NET provides support for a new generation of data access technology: ADO.NET.

> **Note** When you deploy an application that includes Visual Studio data-access components, you must make sure that the user installing the application has version 2.7 or later of the Microsoft Data Access Components (MDAC). For more information, see Adding a Launch Condition for Microsoft Data Access Components.

In This Section

Introduction to Data Access with ADO.NET
Provides a high-level overview of the most important concepts of data access in ADO.NET.
Benefits of ADO.NET
Lists the advantages that ADO.NET offers for application development in Visual Studio .NET.
Comparison of ADO.NET and ADO
Describes differences between ADO.NET and ADO.

Related Sections

Design Goals for ADO.NET
Discusses the motivation and design goals behind creating ADO.NET.
ADO.NET Architecture
Provides an overview of the architecture and components of ADO.NET.
ADO.NET Datasets
Lists topics that pertain to datasets, including an overview and instructions on how to create and work with them.
ADO.NET Data Adapters
Lists topics that pertain to data adapters, including an overview and instructions on how to create adapters, work with parameters, and create table mappings.
Walkthrough: Simple Data Access in a Windows Form
Illustrates a basic data-access scenario using a Windows Form that displays data from a SQL Server database.
Walkthrough: Displaying Data in a Windows Form Using a Parameterized Query
Describes how to create a Windows Form with individually bound controls that display different data depending on what the user enters into a text box.
Walkthrough: Creating a Distributed Application
Illustrates how to create a multi-tier application with a Windows Form and Web Forms page as the front end, an XML Web service component, and a database back end.
Walkthrough: Creating a Master-Detail Windows Form
Describes how to create a parent-child relationship in a dataset and display related records in a Windows Form.

Introduction to Data Access with ADO.NET

As you develop applications using ADO.NET, you will have different requirements for working with data. In some cases, you might simply want to display data on a form. In other cases, you might need to devise a way to share information with another company.

No matter what you do with data, there are certain fundamental concepts that you should understand about the data approach in ADO.NET. You might never need to know some of the details of data handling — for example, you might never need to directly edit an XML file containing data — but it is very useful to understand the data architecture in ADO.NET, what the major data components are, and how the pieces fit together.

This introduction presents a high-level overview of these most important concepts. The topic deliberately skips over many details — for example, there is much more to datasets than what is mentioned here — in favor of simply introducing you to ideas behind data integration in ADO.NET.

> **Note** When you deploy an application that includes Visual Studio data-access components, you must make sure that the user installing the application has version 2.7 or later of the Microsoft Data Access Components (MDAC). For more information, see Adding a Launch Condition for Microsoft Data Access Components.

ADO.NET Does Not Depend On Continuously Live Connections

In traditional client/server applications, components establish a connection to a database and keep it open while the application is running. For a variety of reasons, this approach is impractical in many applications:

- Open database connections take up valuable system resources. In most cases, databases can maintain only a small number of concurrent connections. The overhead of maintaining these connections detracts from overall application performance.
- Similarly, applications that require an open database connection are extremely difficult to scale up. An application that does not scale up well might perform acceptably with four users but will likely not do so with hundreds. ASP.NET Web applications in particular need to be easily scalable, because traffic to a Web site can go up by orders of magnitude in a very short period.
- In ASP.NET Web applications, the components are inherently disconnected from each other. The browser requests a page from the server; when the server has finished processing and sending the page, it has no further connection with the browser until the next request. Under these circumstances, maintaining open connections to a database is not viable, because there is no way to know whether the data consumer (the client) requires further data access.
- A model based on always-connected data can make it difficult and impractical to exchange data across application and organizational boundaries using a connected architecture. If two components need to share the same data, both have to be connected, or a way must be devised for the components to pass data back and forth.

For all these reasons, data access with ADO.NET is designed around an architecture that uses connections sparingly. Applications are connected to the database only long enough to fetch or update the data. Because the database is not holding on to connections that are largely idle, it can service many more users.

Database Interactions Are Performed Using Data Commands

To perform operations in a database, you execute SQL statements or stored procedures (which include SQL statements). You use SQL statements or stored procedures to read and write rows and perform aggregate functions, such as adding or averaging. You also use SQL statements or stored procedures to create or modify tables or columns, to perform transactions, and so on.

In ADO.NET you use data commands to package a SQL statement or stored procedure. For example, if you want to read a set of rows from the database, you create a data command and configure it with the text of a SQL Select statement or the name of a stored procedure that fetches records.

When you want to get the rows, you do the following:

1. Open a connection.
2. Call an execute method of the command, which in turn:
   a. Executes the SQL statement or stored procedure referenced by the command.
   b. Then closes the connection.

   The connection stays open only long enough to execute the statement or stored procedure.

When you call a command's execute method, it returns a value. Commands that update the database return the number of rows affected; other types of commands return an error code. If the command queries the database with a SELECT statement, the

command can return a set of rows. You can fetch these rows using a data reader, which acts as a very fast read-only, forward-only cursor. For more information, see Retrieving Data Using the DataReader.

> **Security Note** When using data commands with a CommandType property set to **Text**, carefully check information that is sent from a client before passing it to your database. Malicious users might try to send (inject) modified or additional SQL statements in an effort to gain unauthorized access or damage the database. Before you transfer user input to a database, you should always verify that the information is valid; it is a best practice to always use parameterized queries or stored procedures when possible. For more information, see Scripting Exploits.

If you need to perform more than one operation — for example, read some rows and then update them — you use multiple data commands, one for each operation. Each operation is performed separately. For example, to read the rows, you open the connection, read the rows, and then close the connection. When you want to update data, you again open the connection, perform the update, and then close the connection again.

Data commands can include parameters (specifically, a collection of parameter objects) that allow you to create parameterized queries such as the following:

```
Select * From customers Where (customer_id = @customerid)
```

You can then set the parameters at run time and execute the command to return or update the data you want.

Data Can Be Cached in Datasets

The most common data task is to retrieve data from the database and do something with it: display it, process it, or send it to another component. Very frequently, the application needs to process not just one record, but a set of them: a list of customers or today's orders, for example. Often the set of records that the application requires comes from more than one table: my customers and all their orders; all authors named "Smith" and the books they have written; and other, similar, sets of related records.

Once these records are fetched, the application typically works with them as a group. For example, the application might allow the user to browse through all the authors named "Smith" and examine the books for one Smith, then move to the next Smith, and so on.

In many cases, it is impractical to go back to the database each time the application needs to process the next record. (Doing so can undo much of the advantage of minimizing the need for open connections.) A solution, therefore, is to temporarily store the records retrieved from the database and work with this temporary set.

This is what a dataset is. A dataset is a cache of records retrieved from a data source. It works like a virtual data store: A dataset includes one or more tables based on the tables in the actual database, and it can include information about the relationships between those tables and constraints on what data the tables can contain.

The data in the dataset is usually a much-reduced version of what is in the database. However, you can work with it in much the same way you do the real data. While you are doing so, you remain disconnected from the database, which frees it to perform other tasks.

Of course, you often need to update data in the database (although not nearly as often as you retrieve data from it). You can perform update operations on the dataset, and these can be written through to the underlying database.

An important point is that the dataset is a passive container for the data. To actually fetch data from the database and (optionally) write it back, you use data adapters. A data adapter contains one or more data commands used to populate a single table in the dataset and update the corresponding table in the database. (A data adapter typically contains four commands, one each to select, insert, update, and delete rows in the database.) Therefore, a data adapter's **Fill** method might execute a SQL statement such as `SELECT au_id, au_lname, au_fname FROM authors` whenever the method is called.

Because a dataset is effectively a private copy of the database data, it does not necessarily reflect the current state of the database. If you want to see the latest changes made by other users, you can refresh the dataset by calling the appropriate **Fill** method.

One of the advantages of using datasets is that components can exchange them as required. For example, a business object in the middle tier might create and populate a dataset, then send it to another component elsewhere in the application for processing. This facility means that components do not have to individually query the database.

Datasets Are Independent of Data Sources

Although a dataset acts as a cache for data drawn from a database, the dataset has no actual relationship with the database. The dataset is a container; it is filled by SQL commands or stored procedures executed from a data adapter.

Because a dataset is not directly tied to a data source, it is a good integration point for data coming from multiple sources. For

example, some of the data in a dataset might come from your database, whereas other parts of it might come from a different database or a non-database source such as a spreadsheet. Some of the data in a dataset might arrive in a stream sent by another component. Once the data is in a dataset, you can work with it using a consistent object model, regardless of its original source.

Data Is Persisted as XML

Data needs to be moved from the data store to the dataset, and from there to various components. In ADO.NET the format for transferring data is XML. Similarly, if data needs to be persisted (for example, into a file), it is stored as XML. If you have an XML file, you can use it like any data source and create a dataset out of it.

In fact, in ADO.NET, XML is a fundamental format for data. The ADO.NET data APIs automatically create XML files or streams out of information in the dataset and send them to another component. The second component can invoke similar APIs to read the XML back into a dataset. (The data is not stored in the dataset as XML — for example, you cannot parse data in a dataset using an XML parser — but instead in a more efficient format.)

Basing data protocols around XML offers a number of advantages:

- XML is an industry-standard format. This means that your application's data components can exchange data with any other component in any other application, as long as that component understands XML. Many applications are being written to understand XML, which provides an unprecedented level of exchange between disparate applications.
- XML is text-based. The XML representation of data uses no binary information, which allows it to be sent via any protocol, such as HTTP. Most firewalls block binary information, but by formatting information in XML, components can still easily exchange the information.

For most scenarios, you do not need to know XML in order to use data in ADO.NET. ADO.NET automatically converts data into and out of XML as needed; you interact with the data using ordinary programming methods.

Schemas Define Data Structures

Although you do not need to know anything about XML to read and write to the database and work with datasets, there are situations in which working with XML is precisely the goal you are after. These are situations in which you are not accessing data, but instead, working with the design of data. To put it another way, in ADO.NET you use XML directly when you are working with metadata.

Datasets are represented as XML. The structure of the dataset — the definition of what tables, columns, data types, constraints, and so on are in the dataset — is defined using an XML Schema based on the XML Schema definition language (XSD). Just as data contained by a dataset can loaded from and serialized as XML, the structure of the dataset can be loaded from and serialized as XML Schema.

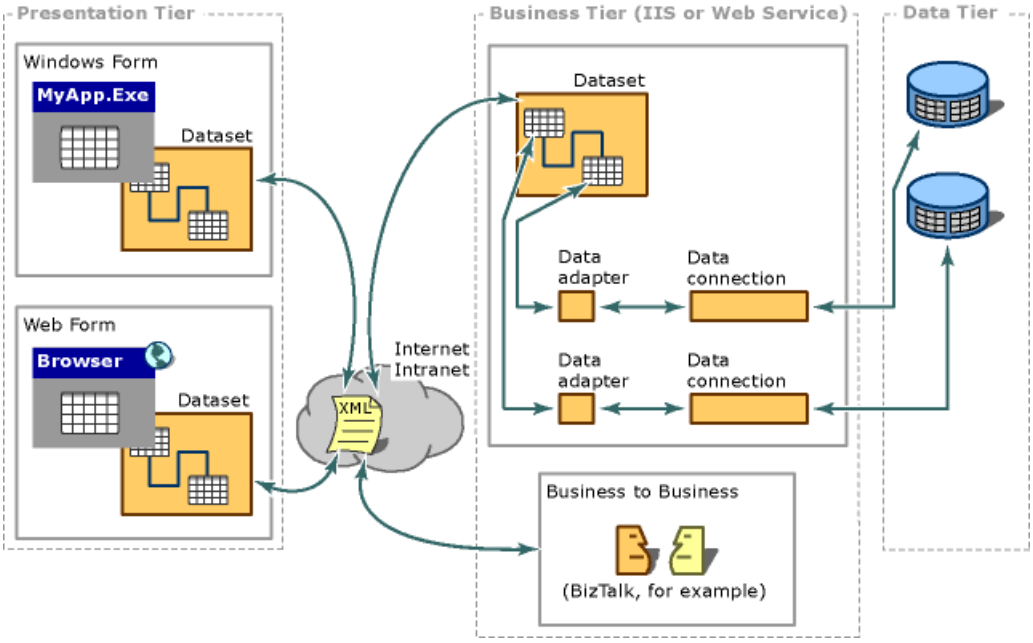
For most of the work you do with data in ADO.NET, you do not have to delve deeply into schemas. Typically, the Visual Studio .NET tools will generate and update schemas as needed, based on what you do in visual designers. For example, when you use the tools to create a dataset representing tables in your database, Visual Studio .NET generates an XML Schema describing the structure of the dataset. The XML Schema is then used to generate a typed dataset, in which data elements (tables, columns, and so on) are available as first-class members. For more information about typed datasets, see Introduction to Datasets.

However, there are times when you want to create or edit schemas yourself. A typical example is developing a schema in conjunction with a partner or client. In that case, the schema serves as a contract between you and the partner regarding the shape of the XML-based data you will exchange. In that situation, you will often have to map the schema elements to the structure of your own database. For more information about designing schemas, see XML Schemas and Data.

Components of ADO.NET

The following illustration shows the major components of an ADO.NET application.

ADO.NET Data Components



The following table summarizes the ADO.NET data components illustrated above and provides links to more information.

Component or object	For more information see
Dataset	Introduction to Datasets
Data adapter	Introduction to Data Adapters
Data connection	Introduction to ADO.NET Connection Design Tools
Windows Form	Introduction to Windows Forms
Web Forms page	Introduction to Web Forms Pages
BizTalk	BizTalk Web page (http://www.microsoft.com/biztalk)

See Also

Benefits of ADO.NET | Comparison of ADO.NET and ADO | ADO.NET Datasets | ADO.NET Data Adapters | Accessing Data with ADO.NET | XML in Visual Studio | XML Schemas and Data

Benefits of ADO.NET

ADO.NET offers several advantages over previous versions of ADO and over other data access components. These benefits fall into the following categories:

Interoperability

ADO.NET applications can take advantage of the flexibility and broad acceptance of XML. Because XML is the format for transmitting datasets across the network, any component that can read the XML format can process data. In fact, the receiving component need not be an ADO.NET component at all: The transmitting component can simply transmit the dataset to its destination without regard to how the receiving component is implemented. The destination component might be a Visual Studio application or any other application implemented with any tool whatsoever. The only requirement is that the receiving component be able to read XML. As an industry standard, XML was designed with exactly this kind of interoperability in mind.

Maintainability

In the life of a deployed system, modest changes are possible, but substantial, architectural changes are rarely attempted because they are so difficult. That is unfortunate, because in a natural course of events, such substantial changes can become necessary. For example, as a deployed application becomes popular with users, the increased performance load might require architectural changes. As the performance load on a deployed application server grows, system resources can become scarce and response time or throughput can suffer. Faced with this problem, software architects can choose to divide the server's business-logic processing and user-interface processing onto separate tiers on separate machines. In effect, the application server tier is replaced with two tiers, alleviating the shortage of system resources.

The problem is not designing a three-tiered application. Rather, it is increasing the number of tiers after an application is deployed. If the original application is implemented in ADO.NET using datasets, this transformation is made easier. Remember, when you replace a single tier with two tiers, you arrange for those two tiers to trade information. Because the tiers can transmit data through XML-formatted datasets, the communication is relatively easy.

Programmability

ADO.NET data components in Visual Studio encapsulate data access functionality in various ways that help you program more quickly and with fewer mistakes. For example, data commands abstract the task of building and executing SQL statements or stored procedures.

Similarly, ADO.NET data classes generated by the tools result in typed datasets. This in turn allows you to access data through typed programming. For example, consider the following line of code that accesses a data member in an untyped dataset:

```
' Visual Basic
If TotalCost > DataSet1.Tables("Customers").Rows(i)("AvailableCredit") Then

// C#
if (TotalCost > DataSet1.Tables["Customers"].Rows[i]["AvailableCredit"]) { }
```

The equivalent line to access a data member in a typed dataset looks like the following:

```
' Visual Basic
If TotalCost > DataSet1.Customers("Jones").AvailableCredit Then

// C#
if(TotalCost > DataSet1.Customers("Jones").AvailableCredit) { }
```

The code for the typed dataset is easier to read. It is also easier to write, because statement completion is provided. For example, "AvailableCredit" is among the list of choices for completing the following statement:

```
IF TotalCost > Customer.
```

Finally, the code for the typed dataset is safer, because it provides for the compile-time checking of types. For example, suppose that `AvailableCredit` is expressed as a currency value. If the programmer erroneously assigns a string value to `AvailableCredit`, the environment would report the error to the programmer during compile time. When working with untyped datasets, the programmer would not learn of the error until run time.

For more information about typed and untyped datasets, see Introduction to Datasets.

Performance

For disconnected applications, ADO.NET datasets offer performance advantages over ADO disconnected recordsets. When using COM marshalling to transmit a disconnected recordset among tiers, a significant processing cost can result from converting the values in the recordset to data types recognized by COM. In ADO.NET, such data-type conversion is not necessary.

Scalability

Because the Web can vastly increase the demands on your data, scalability has become critical. Internet applications have a limitless supply of potential users. Although an application might serve a dozen users well, it might not serve hundreds —or hundreds of thousands — equally well. An application that consumes resources such as database locks and database connections will not serve high numbers of users well, because the user demand for those limited resources will eventually exceed their supply.

ADO.NET accommodates scalability by encouraging programmers to conserve limited resources. Because any ADO.NET application employs disconnected access to data, it does not retain database locks or active database connections for long durations.

See Also

Introduction to Data Access with ADO.NET | Comparison of ADO.NET and ADO | Data Access Technologies

Comparison of ADO.NET and ADO

You can understand the features of ADO.NET by comparing them to particular features of ActiveX Data Objects (ADO).

In-memory Representations of Data

In ADO, the in-memory representation of data is the recordset. In ADO.NET, it is the dataset. There are important differences between them.

Number of Tables

A recordset looks like a single table. If a recordset is to contain data from multiple database tables, it must use a JOIN query, which assembles the data from the various database tables into a single result table.

In contrast, a dataset is a collection of one or more tables. The tables within a dataset are called data tables; specifically, they are DataTable objects. If a dataset contains data from multiple database tables, it will typically contain multiple **DataTable** objects. That is, each **DataTable** object typically corresponds to a single database table or view. In this way, a dataset can mimic the structure of the underlying database.

A dataset usually also contains relationships. A relationship within a dataset is analogous to a foreign-key relationship in a database —that is, it associates rows of the tables with each other. For example, if a dataset contains a table about investors and another table about each investor's stock purchases, it could also contain a relationship connecting each row of the investor table with the corresponding rows of the purchase table.

Because the dataset can hold multiple, separate tables and maintain information about relationships between them, it can hold much richer data structures than a recordset, including self-relating tables and tables with many-to-many relationships.

Data Navigation and Cursors

In ADO you scan sequentially through the rows of the recordset using the ADO **MoveNext** method. In ADO.NET, rows are represented as collections, so you can loop through a table as you would through any collection, or access particular rows via ordinal or primary key index. **DataRelation** objects maintain information about master and detail records and provide a method that allows you to get records related to the one you are working with. For example, starting from the row of the `Investor` table for "Nate Sun," you can navigate to the set of rows of the `Purchase` table describing his purchases.

A *cursor* is a database element that controls record navigation, the ability to update data, and the visibility of changes made to the database by other users. ADO.NET does not have an inherent cursor object, but instead includes data classes that provide the functionality of a traditional cursor. For example, the functionality of a forward-only, read-only cursor is available in the ADO.NET **DataReader** object. For more information about cursor functionality, see Data Access Technologies.

Minimized Open Connections

In ADO.NET you open connections only long enough to perform a database operation, such as a Select or Update. You can read rows into a dataset and then work with them without staying connected to the data source. In ADO the recordset can provide disconnected access, but ADO is designed primarily for connected access.

There is one significant difference between disconnected processing in ADO and ADO.NET. In ADO you communicate with the database by making calls to an OLE DB provider. In ADO.NET you communicate with the database through a data adapter (an OleDbDataAdapter, SqlDataAdapter, OdbcDataAdapter, or OracleDataAdapter object), which makes calls to an OLE DB provider or the APIs provided by the underlying data source. The important difference is that in ADO.NET the data adapter allows you to control how the changes to the dataset are transmitted to the database — by optimizing for performance, performing data validation checks, or adding any other extra processing.

> **Note** Data adapters, data connections, data commands, and data readers are the components that make up a .NET Framework data provider. Microsoft and third-party providers can make available other .NET Framework data providers that can be integrated into Visual Studio. For information on the different .NET Data providers, see .NET Data Providers.

Sharing Data Between Applications

Transmitting an ADO.NET dataset between applications is much easier than transmitting an ADO disconnected recordset. To transmit an ADO disconnected recordset from one component to another, you use COM marshalling. To transmit data in ADO.NET, you use a dataset, which can transmit an XML stream.

The transmission of XML files offers the following advantages over COM marshalling:

Richer data types

COM marshalling provides a limited set of data types — those defined by the COM standard. Because the transmission of datasets in ADO.NET is based on an XML format, there is no restriction on data types. Thus, the components sharing the dataset can use whatever rich set of data types they would ordinarily use.

Performance

Transmitting a large ADO recordset or a large ADO.NET dataset can consume network resources; as the amount of data grows, the stress placed on the network also rises. Both ADO and ADO.NET let you minimize which data is transmitted. But ADO.NET offers another performance advantage, in that ADO.NET does not require data-type conversions. ADO, which requires COM marshalling to transmit records sets among components, does require that ADO data types be converted to COM data types.

Penetrating Firewalls

A firewall can interfere with two components trying to transmit disconnected ADO recordsets. Remember, firewalls are typically configured to allow HTML text to pass, but to prevent system-level requests (such as COM marshalling) from passing.

Because components exchange ADO.NET datasets using XML, firewalls can allow datasets to pass.

See Also

Introduction to Data Access with ADO.NET | Benefits of ADO.NET | Data Access Technologies

Connecting to Data Sources with ADO.NET

The following topics provide information about how to create and manage ADO.NET connection objects in Visual Studio.

In This Section

Introduction to ADO.NET Connection Design Tools
Provides an overview of connection objects and how to work with them.
Creating ADO.NET Connection Objects
Explains how to create connection objects on forms and components.
Creating Connections to SQL Server
Explains how to establish a connection between an application and a SQL Server database.
Creating Connections to Access Databases
Explains how to establish a connection between an application and a Microsoft Access database.
Creating Connections to ODBC Data Sources
Explains how to establish a connection between an application and an ODBC data source.
Creating Connections to Oracle Databases
Explains how to establish a connection between an application and an Oracle database.

Related Sections

Adding New Data Connections in Server Explorer
Explains how to create design-time connections in Visual Studio, which you can use to browse available data sources.
Connecting to a Data Source Using ADO.NET
Lists topics in the .NET Framework documentation that describes how to create and manage connection objects programmatically.
Configuring Applications Using Dynamic Properties
Provides conceptual information about storing property values in a configuration file that can be changed at run time, which is particularly useful for connection information such as server name and user name.
Connection Pooling for the .Net Data Provider for SQL Server
Provides instructions on how to manually manage connection pooling if you are using the **SqlConnection** class.

Introduction to ADO.NET Connection Design Tools

To move data between a data store and your application, you must first have a connection to the data store.

> **Note** Data adapters, data connections, data commands, and data readers are the components that make up a .NET Framework data provider. Microsoft and third-party providers can make available other .NET Framework data providers that can be integrated into Visual Studio. For more information, see .NET Data Providers.

In ADO.NET you create and manage connections using connection objects:

- SqlConnection - an object that manages a connection to a SQL Server version 7.0 or later. It is optimized for use with SQL Server 7.0 or later by (among other things) bypassing the OLE DB layer.
- OleDbConnection - an object that manages a connection to any data store accessible via OLE DB.
- OdbcConnection - an object that manages a connection to a data source created by using a connection string or ODBC data source name (DSN).
- OracleConnection - an object that manages a connection to Oracle databases.

Connection Strings

All connection objects expose roughly the same members. However, the specific members available with a given **OleDbConnection** object depend on what data source it is connected to; not all data sources support all members of the **OleDbConnection** class.

The primary property associated with a connection object is the **ConnectionString** property, which consists of a string with attribute/value pairs for information required to log on to a database and point to a specific database. A typical **ConnectionString** property might look like the following:

```
Provider=SQLOLEDB.1;Data Source=MySQLServer;Initial Catalog=NORTHWIND;Integrated Security=SSP
I
```

This particular connection string specifies that the connection should use Windows integrated security (NT authentication). A connection string can instead include a user id and password, but this is not recommended, because these attributes are then compiled into your application and therefore a potential security breach. For more information, see Access Permissions for Web Applications and Security Model.

> **Security Note** Storing connection-string details (such as the server name, user name, and password) can have implications for the security of your application. Using Windows Integrated Security is a more secure way to control access to a database. For more information, see Database Security.

The most common attribute/value pairs used by OLE DB are also represented separately by an individual property, such as **DataSource** and **Database**. When working with a connection object, you can either set the **ConnectionString** property as a single string, or you can set individual connection properties. (If your data source requires connection-string values that are not represented by individual properties, then you must set the **ConnectionString** property.) Alternatively, you can also set the **ConnectionString** property to the path of a Microsoft Data Link (.udl) file. For more information about data link files, see Data Link API Overview.

> **Note** The **SQLConnection** object does not support — or allow you to specify — the **Provider** attribute.

Opening and Closing Connections

The two primary methods for connections are **Open** and **Close**. The **Open** method uses the information in the **ConnectionString** property to contact the data source and establish an open connection. The **Close** method shuts the connection down. Closing connections is essential, because most data sources support only a limited number of open connections, and open connections take up valuable system resources.

If you are working with data adapters or data commands, you do not have to explicitly open and close a connection. When you call a method of these objects (for example, the data adapter's **Fill** or **Update** method), the method checks whether the connection is already open. If not, the adapter opens the connection, performs its logic, and closes the connection again.

Methods such as **Fill** only open and close the connection automatically if it is not already open. If the connection is open, the methods use it but do not close it. This gives you the flexibility to open and close data commands yourself. You might do this if you have multiple data adapters that share a connection. In that case, it is inefficient to have each adapter open and close the connection when you call its **Fill** method. Instead, you can open the connection, call the **Fill** method of each adapter, and then

close the connection when you are done.

Pooling Connections

Applications often have different users performing the same type of database access. For example, in ASP.NET Web applications many users might be querying the same database to get the same data. In those cases, the performance of the application can be enhanced by having the application share, or "pool," connections to the data source. The overhead of having each user open and close a separate connection can otherwise have an adverse effect on application performance.

If you are using the **OleDbConnection**, **OdbcConnection**, or **OracleConnection** class, connection pooling is handled automatically by the provider, so you do not need to manage it yourself.

If you are using the **SqlConnection** class, connection pooling is managed implicitly, but also provides options that allow you to manage pooling yourself. For more information, see Connection Pooling for the .Net Framework Data Provider for SQL Server.

Transactions

Connection objects support transactions with a **BeginTransaction** method that creates a transaction object (for example, a SqlTransaction object). The transaction object in turn supports methods that allow you to commit or roll back the transactions.

Transactions are managed in code. For more information, see Performing Transactions.

Configurable Connection Properties

In many applications, connection information cannot be determined at design time. For example, in an application that will be distributed to many different customers, you might not be able to determine at design time the connection information (such as the name of a server).

Connection strings are therefore often specified as dynamic properties. Because dynamic properties are stored in a configuration file (and not compiled into the application binary files), they can be changed without having to recompile the application.

A typical strategy is to specify connection properties as dynamic properties, provide a way (such as a Windows Form or Web Form page) for users to specify the relevant details, and then update the configuration file. The dynamic-property mechanism built into the .NET Framework automatically gets the values from the configuration file when the property is read, and updates the file when the value is updated.

Connection Information and Security

Because opening a connection involves getting access to an important resource — a database — there are often security issues in configuring and working with a connection.

How you secure the application and its access to the data source depends on the architecture of your system. In a Web-based application, for example, users typically get anonymous access to Internet Information Services (IIS), and therefore do not provide security credentials. In that case, your application maintains its own logon information and uses it (rather than any specific user information) to open the connection and access the database.

> **Security Note** Storing connection-string details (such as the server name, user name, and password) can have implications for the security of your application. Using Windows Integrated Security is a more secure way to control access to a database. For more information, see Database Security.

The ASP.NET Web application binary files and configuration file are secured from Web access by the inherent ASP.NET security model, which prevents access to these files using any Internet protocol (HTTP, FTP, and so on). To prevent access to the Web server computer itself via other methods (for example, from within your internal network), you use the security features of Windows.

In intranet or two-tier applications, you can take advantage of the integrated security option provided by Windows, IIS, and SQL Server. In that model, a user's authentication credentials for the local network are also used to access database resources, and no explicit user name or password are used in the connection string. (Typically, permissions are established on the database server computer via groups, so that you do not need to establish individual permissions for every user who might access the database.) In this model, you do not need to store logon information for the connection at all, and there are no extra steps required to protect connection string information.

For more information about security, see the following topics:

- Best Practices for Securability
- .NET Framework Security
- Access Permissions for Web Applications

- ASP.NET Web Application Security

Design-Time Connections in Server Explorer

Server Explorer provides a way for you to create design-time connections to data sources. This permits you to browse available data sources; display information about the tables, columns, and other elements they contain; and edit and create database elements.

Your application does not directly use the connections you create this way. Generally, the information provided by a design-time connection is used to set properties for a new connection object that you add to your application.

For example, at design time you might use Server Explorer to create a connection to the SQL server MyServer and the database Northwind. Later, when designing a form, you can browse the Northwind database, select columns from a table, and drag them onto the form. This creates a data adapter for your form. It also creates a new form-specific connection, copying connection string information from the design-time connection to your new connection. When the application runs, your new connection is used to access the data source.

Information about design-time connections is stored on your local computer independently of a specific project or solution. Therefore, once you have established a design-time connection while working in an application, it appears in Server Explorer whenever you work in Visual Studio (as long as the server to which the connection points is available). For more information about using Server Explorer and creating design-time connections, see Adding New Data Connections in Server Explorer.

> **Note** The data connection you create in Server Explorer determines the objects that get created when dragging database elements onto a designer. For example, if you are connecting to a SQL Server at design time, and your data connection was originally set up by selecting the **OLE DB Provider for SQL Server**, then dragging a table from Server Explorer onto a designer will create a **SqlConnection** object. If your application requires an ODBC connection, be sure to select the **OLE DB Provider for ODBC** when creating your data connection in Server Explorer.

Connection Design Tools in Visual Studio

You usually do not need to directly create and manage connection objects in Visual Studio. When you use tools such as the Data Adapter Wizard, the tools typically prompt you for connection information (that is, connection-string information) and automatically create connection objects for you on the form or component you are working with.

However, if you want, you can add connection objects yourself to a form or component and set their properties. This is useful if you are not working with datasets (and therefore not with data adapters), but instead simply reading data. You might create connection objects yourself if you will be using transactions.

See Also

Creating ADO.NET Connection Objects | Connecting to a Data Source Using ADO.NET |
Connection Pooling for the .NET Framework Data Provider for SQL Server

Creating ADO.NET Connection Objects

If you are using data design tools in Visual Studio, you often do not need to explicitly create a connection object for your form or component. However, in some instances you might find it convenient to create a connection alone.

Your options for creating a connection are:

- Use any of the following data design tools, which create a connection object as part of the task they are performing:
  - Data Adapter Configuration Wizard - this wizard prompts you for information to create a connection that is in turn linked to a data adapter. For more information, see Data Adapter Configuration Wizard.
  - Data Form Wizard - this wizard creates the connection object as part of the form it is configuring. For more information, see Data Form Wizard.
- Drag a table, selection of columns, or stored procedure from Server Explorer onto a form or component. By dragging one of these elements onto a form, you create both a data adapter and a connection. For more information, see Adding New Data Connections in Server Explorer.
- Create a standalone connection. This option creates a connection object on your form or component whose properties you configure manually. This strategy is useful if you intend to set connection properties at run time or if you simply prefer setting the properties in the Properties window. The procedure is described below.
- Create a connection programmatically. For more information about creating and managing connections in code, see Connecting to SQL Server Using ADO.NET and Connecting to an OLE DB Data Source Using ADO.NET.

To create a connection

1. From the **Data** tab of the Toolbox, drag a connection object onto your form or component. For more information about which connection object to use, see Introduction to ADO.NET Connection Design Tools.
2. Select the connection in the designer and then use the **Properties** window to set its connection string.

   - You can set the **ConnectionString** property as a single unit. (You cannot specify a **Provider** attribute for a **SqlConnection** object.)

     – or –

   - You can set individual properties (**DataSource**, **Database**, **UserName**, and so on). If you set individual properties, the connection string is built for you.

     > **Security Note** Storing connection-string details (such as the server name, user name, and password) can have implications for the security of your application. Using Windows Integrated Security is a more secure way to control access to a database. For more information, see Database Security.

3. If you want to rename the connection object, set its **Name** property.

   > **Note** If connection information needs to be set at run time, you can configure connection properties as dynamic properties. For more information, see Configuring Applications Using Dynamic Properties.

4. If you want to set properties at run time without recompiling the application, specify connection properties. For more information about making connection properties dynamic, see Introduction to ADO.NET Connection Design Tools. For an overview of dynamic properties, see Configuring Applications Using Dynamic Properties.

See Also

Creating Connection to SQL Servers | Creating Connections to Access Databases | Creating Connections to ODBC Data Sources | Creating Connections to Oracle Databases | Introduction to ADO.NET Connection Design Tools | Connecting to a Data Source Using ADO.NET | Connection Pooling for the .NET Data Provider for SQL Server

Creating Connections to SQL Server

When connecting to SQL Server use the **Microsoft OLE DB Provider for SQL Server**.

> **Security Note** Storing connection-string details (such as the server name, user name, and password) can have implications for the security of your application. Using Windows Integrated Security is a more secure way to control access to a database. For more information, see Database Security.

There are two ways to connect to SQL Server:

- Visually, with design-time tools.
- Programmatically.

Connecting to SQL Server in Server Explorer

Establishing a connection to SQL Server in Server Explorer allows you to easily create SqlConnection, SqlDataAdapter, and SqlCommand objects by dragging data objects from Server Explorer onto forms or designers.

To create a connection in Server Explorer

1. In Server Explorer, right-click **Data Connections** and click **Add Connection**.

   The **Data Link Properties** dialog box opens.

2. The default provider is **Microsoft OLE DB Provider for SQL Server**.
3. Select a server name from the drop-down list, or type the location of the server where the database you want to access is located.

   > **Note** Selecting the database on the server is a separate action. Refresh the list by clicking **Refresh**.

4. Based on the requirements of your database or application, select either Windows NT Integrated security or use a specific user name and password to log on to the SQL Server. For more information, see Data Link Properties Dialog Box.
5. Select the database you want to connect to from the drop-down list.
6. Click **OK**.

Connecting to SQL Server from within your application

Creating connections to data sources from within your application can be accomplished visually using Visual Studio .NET designers, or programmatically by creating data connection objects in code.

Connecting to SQL Server Visually

Visual Studio supports creating data objects by dragging items from Server Explorer or the Data tab of the Toolbox onto your form or designer.

To create a connection from Server Explorer

1. Create a **Data Connection** in Server Explorer to your SQL Server. For more information, see the above section titled "Connecting to SQL Server in Server Explorer."
2. Drag the connection onto your form or designer.

   A **SqlConnection** object appears in the component tray.

3. Advance to the section "Next Steps" later in this topic.

To create a connection from the Data tab of the Toolbox

1. Drag a **SqlConnection** onto your form or designer.

   An unconfigured **SqlConnection** object appears in the component tray.

2. In the Properties window, click the **ConnectionString** property.
3. Select an existing connection in the drop-down list, or click **New Connection** to open the Data Link Properties Dialog Box and configure a new connection.
4. Advance to the section "Next Steps" later in this topic.

Connecting to SQL Server Programmatically

You can create a **SqlConnection** object directly in code.

To programmatically create a connection between your application and a SQL Server database

- The following code creates a SqlConnection object, sets the SqlConnection.ConnectionString property, and opens the connection.

```
' Visual Basic
Public Sub ConnectToSql()
    Dim conn As New SqlClient.SqlConnection
    ' TODO: Modify the connection string and include any
    ' additional required properties for your database.
    conn.ConnectionString = & _
    "integrated security=SSPI;data source=SQL Server Name;" & _
    "persist security info=False;initial catalog=northwind"
    Try
        conn.Open()
        ' Insert code to process data.
    Catch ex As Exception
        MessageBox.Show("Failed to connect to data source")
    Finally
        conn.Close()
    End Try
End Sub

// C#
public void ConnectToSql ()
{
    System.Data.SqlClient.SqlConnection conn =
        new System.Data.SqlClient.SqlConnection ();
    // TODO: Modify the connection string and include any
    // additional required properties for your database.
    conn.ConnectionString =
     "integrated security=SSPI;data source=SQL Server Name;" +
     "persist security info=False;initial catalog=northwind";
    try
    {
        conn.Open();
        // Insert code to process data.
    }
        catch (Exception ex)
    {
        MessageBox.Show("Failed to connect to data source");
    }
    finally
    {
        conn.Close();
    }
}
```

Next Steps

Now that you have established a connection between your application and the SQL Server database, you can create objects to fetch and update data. The following table provides links to some relevant topics.

To	See
Fetch read-only data	Retrieving Data Using the DataReader
Fill a dataset with data	ADO.NET Datasets

Execute SQL statements directly	Performing Database Operations Directly

See Also

Connecting to Data Sources with ADO.NET | Adding New Data Connections in Server Explorer | Databases in Server Explorer | SQL Server Databases | Visual Database Tools | Data Link Properties Dialog Box

Creating Connections to Access Databases

When connecting to an Access database, use the **Microsoft Jet 4.0 OLE DB Provider**.

> **Security Note** Storing connection-string details (such as the server name, user name, and password) can have implications for the security of your application. Using Windows Integrated Security is a more secure way to control access to a database. For more information, see Database Security.

There are two ways to connect to an Access database:

- Visually, with design-time tools.
- Programmatically.

Connecting to an Access Database in Server Explorer

Establishing a connection to an Access database in Server Explorer allows you to easily create OleDbConnection, OleDbDataAdapter, and OleDbCommand objects by dragging data objects from Server Explorer onto forms or designers.

To create a connection in Server Explorer

1. In Server Explorer, right-click **Data Connections** and click **Add Connection**.
2. In the **Data Link Properties** dialog box, click the **Provider** tab.
3. Click the **Microsoft Jet 4.0 OLE DB Provider**, then click **Next**.
4. Click the ellipsis (...) button to browse to the Microsoft Access (.mdb) file you want to access.
5. Type the **User ID** to use for authentication when you log on to the data source.
6. Type the **Password** to use for authentication when you log on to the data source.
7. Click **OK**.

Connecting to an Access Database from Within Your Application

Creating connections to data sources from within your application can be accomplished visually using Visual Studio .NET design-time tools (including Server Explorer and the Toolbox), or programmatically by creating data connection objects in code.

Connecting to an Access Database Visually

Visual Studio supports creating data objects by dragging items from Server Explorer or the Data tab of the Toolbox onto your form or designer.

To create a connection from Server Explorer

1. Create a **Data Connection** in Server Explorer to your Access database. For more information, see the above section titled "Connecting to an Access database in Server Explorer."
2. Drag the connection from Server Explorer onto your form or designer.

   An **OleDbConnection** object appears in the component tray.

3. Advance to the section "Next Steps" later in this topic.

To create a connection from the Data tab of the Toolbox

1. Drag an **OleDbConnection** object onto your form or designer.

   An unconfigured **OleDbConnection** object appears in the component tray.

2. In the Properties window, click the **ConnectionString** property.
3. Do one of the following:

   - Click an existing connection in the drop-down list.

     - or -

   - Click **New Connection** to open the Data Link Properties Dialog Box and configure a new connection.

4. Advance to the section "Next Steps" later in this topic.

Connecting to an Access Database Programmatically

You can create a **OleDbConnection** object directly in code.

To programmatically create a connection between your application and an Access database

- The following code creates an OleDbConnection object, sets the OleDbConnection.ConnectionString property, and opens the connection.

```
' Visual Basic
Public Sub ConnectToAccess()
    Dim conn As New System.Data.OleDb.OleDbConnection()
    ' TODO: Modify the connection string and include any
    ' additional required properties for your database.
    conn.ConnectionString = & _
    "Provider=Microsoft.Jet.OLEDB.4.0;Data source=" & _
    "C:\Documents and Settings\username\My Documents\dbFile.mdb"
    Try
       conn.Open()
       ' Insert code to process data.
    Catch ex As Exception
       MessageBox.Show("Failed to connect to data source")
    Finally
       conn.Close()
    End Try
End Sub

// C#
public void ConnectToAccess()
{
    System.Data.OleDb.OleDbConnection conn = new
        System.Data.OleDb.OleDbConnection();
    // TODO: Modify the connection string and include any
    // additional required properties for your database.
    conn.ConnectionString = @"Provider=Microsoft.Jet.OLEDB.4.0;" +
        @"Data source= C:\Documents and Settings\username\" +
        @"My Documents\AccessFile.mdb";
    try
    {
        conn.Open();
        // Insert code to process data.
    }
        catch (Exception ex)
    {
        MessageBox.Show("Failed to connect to data source");
    }
    finally
    {
        conn.Close();
    }
}
```

Next Steps

Now that you have established a connection between your application and the Access database, you can create objects to fetch and update data. The following table provides links to some relevant topics.

To	See
Fetch read-only data	Retrieving Data Using the DataReader
Fill a dataset with data	ADO.NET Datasets

Execute SQL statements directly	Performing Database Operations Directly

See Also

Connecting to Data Sources with ADO.NET | Adding New Data Connections in Server Explorer | Databases in Server Explorer | Visual Database Tools | Data Link Properties Dialog Box | OleDbConnection Class

Creating Connections to ODBC Data Sources

When connecting to a DSN, use the **Microsoft OLE DB Provider for ODBC Drivers**.

> **Security Note** Storing connection-string details (such as the server name, user name, and password) can have implications for the security of your application. Using Windows Integrated Security is a more secure way to control access to a database. For more information, see Database Security.

There are two ways to connect to an ODBC data source:

- Visually, with design-time tools.
- Programmatically.

Connecting to an ODBC Data Source in Server Explorer

Establishing a connection to an ODBC data source in Server Explorer allows you to easily create OdbcConnection, OdbcDataAdapter, and OdbcCommand objects by dragging data objects from Server Explorer onto forms or designers.

To create a connection in Server Explorer

1. In Server Explorer, right-click **Data Connections** and click **Add Connection**.
2. In the **Data Link Properties** dialog box, click the **Provider** tab.
3. Click the **Microsoft OLE DB Provider for ODBC Drivers**, then click **Next**.
4. Select from the list, or type the ODBC data source name you want to access.
5. Type the **User ID** to use for authentication when you log on to the data source.
6. Type the **Password** to use for authentication when you log on to the data source.
7. Click **OK**.

Connecting to an ODBC Data Source from Within Your Application

Creating connections to data sources from within your application can be accomplished visually using Visual Studio .NET designers, or programmatically by creating data connection objects in code.

Connecting to an ODBC Data Source Visually

Visual Studio supports creating data objects by dragging items from Server Explorer or the Data tab of the Toolbox onto your form or designer.

To create a connection from Server Explorer

1. Create a **Data Connection** in Server Explorer to your ODBC data source. For details, see the above section titled "Connecting to an ODBC data source in Server Explorer."
2. Drag the connection onto your form or designer.
3. An **OdbcConnection** object appears in the component tray.
4. Advance to the section "Next Steps" later in this topic.

To create a connection from the Data tab of the Toolbox

1. Drag an **OdbcConnection** onto your form or designer.

   An unconfigured **OdbcConnection** object appears in the component tray.
2. In the Properties window, click the **ConnectionString** property.
3. Select an existing connection in the drop-down list or click **New Connection** to open the Data Link Properties Dialog Box and configure a new connection.
4. Advance to the section "Next Steps" later in this topic.

Connecting to an ODBC Data Source Programmatically

You can create an **OdbcConnection** object directly in code.

To programmatically create a connection between your application and a DSN

- The following code creates an OdbcConnection object, sets the OdbcConnection.ConnectionString property, and opens the connection.

```
' Visual Basic
    Public Sub ConnectToOdbc()
        Dim conn As New System.Data.Odbc.OdbcConnection
        ' TODO: Modify the connection string and include any
        ' additional required properties for your database.
        conn.ConnectionString = & _
            "FIL=MS Access;DSN=valid data source name"
        Try
            conn.Open()
            ' Insert code to process data.
        Catch ex As Exception
            MessageBox.Show("Failed to connect to data source")
        Finally
            conn.Close()
        End Try
    End Sub

// C#
public void ConnectToData()
{
    System.Data.Odbc.OdbcConnection conn =
        new System.Data.Odbc.OdbcConnection ();
    // TODO: Modify the connection string and include any
    // additional required properties for your database.
    conn.ConnectionString = "FIL=MS Access;DSN=valid data source name";
    try
    {
        conn.Open();
        // Process data here.
    }
    catch (Exception ex)
    {
        MessageBox.Show("Failed to connect to data source");
    }
    finally
    {
        conn.Close();
    }
}
```

Next Steps

Now that you have established a connection between your application and an ODBC data source, you can create objects to fetch and update data. The following table provides links to some relevant topics.

To	See
Fetch read-only data	Retrieving Data Using the DataReader
Fill a dataset with data	ADO.NET Datasets
Execute SQL statements directly	Performing Database Operations Directly

See Also

Connecting to Data Sources with ADO.NET | Adding New Data Connections in Server Explorer | Databases in Server Explorer | Visual Database Tools | Data Link Properties Dialog Box | OdbcConnection Class

Creating Connections to Oracle Databases

When connecting to an Oracle database use the **Microsoft OLE DB Provider for Oracle**.

> **Note** To enable drag and drop support from **Server Explorer** for Oracle databases, the data connection must use the Microsoft OLE DB Provider for Oracle.
>
> **Security Note** Storing connection-string details (such as the server name, user name, and password) can have implications for the security of your application. Using Windows Integrated Security is a more secure way to control access to a database. For more information, see Database Security.

There are two ways to connect to an Oracle database:

- Visually, with design-time tools.
- Programmatically.

Connecting to an Oracle Database in Server Explorer

To create a connection in Server Explorer

1. In Server Explorer, right-click **Data Connections** and click **Add Connection**.
2. In the **Data Link Properties** dialog box, click the **Provider** tab.
3. Click the **Microsoft OLE DB Provider for Oracle**, then click **Next**.
4. Type the server where the database you want to access is located.
5. Type the password to use for authentication when you log on to the data source.
6. Click **OK**.

Connecting to an Oracle Database from within your application

Creating connections to data sources from within your application can be accomplished visually using Visual Studio .NET designers, or programmatically by creating data connection objects in code.

Connecting to an Oracle Database Visually

Visual Studio supports creating data objects by dragging items from Server Explorer or the Data tab of the Toolbox onto your form or designer.

To create a connection from Server Explorer

1. Create a **Data Connection** in Server Explorer to your Oracle database. For more information, see the above section titled "Connecting to an Oracle database in Server Explorer."
2. Drag the connection onto your form or designer.
3. An **OracleConnection** object appears in the component tray.
4. Advance to the section "Next Steps" later in this topic.

To create a connection from the Data tab of the Toolbox

1. Drag an **OracleConnection** onto your form or designer.

   An unconfigured **OracleConnection** object appears in the component tray.

2. In the Properties window, click the **ConnectionString** property.
3. Select an existing connection in the drop-down list, or click **New Connection** to open the Data Link Properties Dialog Box and configure a new connection.
4. Advance to the section "Next Steps" later in this topic.

Connecting to an Oracle Database Programmatically

You can create an **OracleConnection** object directly in code.

To programmatically create a connection between your application and an Oracle database

- The following code creates an OracleConnection object and sets the OracleConnection.ConnectionString property.

```
' Visual Basic
Public Sub ConnectToOracle()
    Dim conn As New OracleClient.OracleConnection()
    ' TODO: Modify the connection string and include any
    ' additional required properties for your database.
    conn.ConnectionString = & _
    "Data Source=<oracle data source name>;Integrated Security=yes"
    Try
        conn.Open()
        ' Insert code to process data.
    Catch ex As Exception
        MessageBox.Show("Failed to connect to data source")
    Finally
        conn.Close()
    End Try
End Sub

// C#
public void ConnectToOracle()
{
    OracleClient.OracleConnection conn =
        new OracleClient.OracleConnection ();
    // TODO: Modify the connection string and include any
    // additional required properties for your database.
    conn.ConnectionString = "Data Source=" +
        "<oracle data source name>;Integrated Security=yes";
    try
    {
        conn.Open();
        // Insert code to process data.
    }
        catch (Exception ex)
    {
        MessageBox.Show("Failed to connect to data source");
    }
    finally
    {
        conn.Close();
    }
}
```

Next Steps

Now that you have established a connection between your application and an Oracle database, you can create objects to fetch and update data. The following table provides links to some relevant topics.

To	See
Fetch read-only data	Retrieving Data Using the DataReader
Fill a dataset with data	ADO.NET Datasets
Execute SQL statements directly	Performing Database Operations Directly

See Also

Connecting to Data Sources with ADO.NET | Adding New Data Connections in Server Explorer | Databases in Server Explorer | Oracle Databases | Visual Database Tools | Data Link Properties Dialog Box

ADO.NET Data Adapters

The following topics provide information about ADO.NET data-adapter objects and how to work with them in Visual Studio. Adapters are used to exchange data between a data source and a dataset.

In This Section

Introduction to Data Adapters
Provides an overview of what data adapters are, what they are used for, and how they are structured.
Parameters in Data-Adapter Commands
Describes the structure and use of parameters that adapters pass at run time to SQL statements or stored procedures.
Table Mapping in Data Adapters
Describes options for how to map column names between a source table and a dataset table.
Creating Data Adapters
Lists different ways of creating data adapters in Visual Studio.
Configuring Parameters for Data Adapters
Explains how to create and set properties of parameters that will be passed at run time to data-adapter commands.
Mapping Data-Source Columns to Dataset Data-Table Columns
Provides instructions for how to establish and modify the mapping between the column names of source tables and dataset tables.
Previewing Data-Adapter Results
Provides instructions for testing how a data adapter will populate a dataset.

Related Sections

Introduction to Data Access with ADO.NET
Provides a general introduction to the concepts of data access in Visual Studio.
Introduction to Datasets
Provides background information on the structure and use of datasets.
Creating and Filling Datasets Example Topics
Presents a series of specific example code topics for common tasks in creating and filling datasets with Visual Basic .NET.
Manipulating Data Example Topics
Presents a series of specific example code topics for common tasks in working with data within your application using Visual Basic .NET.

Introduction to Data Adapters

Data adapters are an integral part of ADO.NET managed providers, which are the set of objects used to communicate between a data source and a dataset. (In addition to adapters, managed providers include connection objects, data reader objects, and command objects.) Adapters are used to exchange data between a data source and a dataset. In many applications, this means reading data from a database into a dataset, and then writing changed data from the dataset back to the database. However, a data adapter can move data between any source and a dataset. For example, there could be an adapter that moves data between a Microsoft Exchange server and a dataset.

> **Note** For more information about data providers, see Using .NET Data Providers to Access Data.

Generally, adapters are configurable to allow you to specify what data to move into and out of the dataset. Often this takes the form of references to SQL statements or stored procedures that are invoked to read or write to a database.

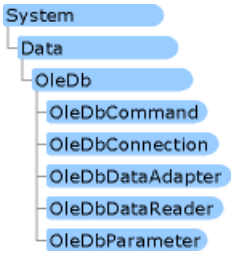
Visual Studio makes these data adapters available for use with databases:

- The OleDbDataAdapter object is suitable for use with any data source exposed by an OLE DB provider.
- The SqlDataAdapter object is specific to SQL Server. Because it does not have to go through an OLE DB layer, it is faster than the **OleDbDataAdapter**. However, it can only be used with SQL Server 7.0 or later.
- The OdbcDataAdapter object is optimized for accessing ODBC data sources.
- The OracleDataAdapter object is optimized for accessing Oracle databases.

> **Note** Data Adapters, Data Connections, Data Commands, and Data Readers are the components that make up a .NET Framework data provider. Microsoft and third-party providers can make available other .NET data providers that can be integrated into Visual Studio. For more information on the different .NET Framework data providers, see .NET Framework Data Providers.

You can create and manipulate adapters using the following portions of the .NET Framework managed-provider namespaces.

OleDb Managed-Provider Namespace



SqlClient Managed-Provider Namespace

Generally, each data adapter exchanges data between a single data-source table and a single DataTable object in the dataset. If the dataset contains multiple data tables, the usual strategy is to have multiple data adapters feeding data to it and writing its data back to individual data-source tables.

When you want to populate a table in a dataset, you call an adapter method that executes an SQL statement or stored procedure. The adapter creates a data reader object (SqlDataReader, OleDbDataReader, OdbcDataReader, or OracleDataReader) to read the data into a dataset.

> **Note** You can read data from the database without having to store it in a dataset, which can be very efficient for situations involving read-only data. For details, see "Read-Only Data" below. You can also execute SQL statements directly without using them to populate a dataset. For details, see Performing Database Operations Directly.

Similarly, when you want to update the database, you invoke an adapter method that calls an appropriate SQL statement or stored procedure to make the actual update in the database.

Data Adapters and Related Tables

An implication of having separate tables in the dataset is that a data adapter typically does not reference SQL commands or stored procedures that join tables. Instead, information from the related tables is read separately into the dataset by different adapters. Then a **DataRelation** object is used to manage constraints between the dataset tables (such as cascading updates) and to allow you to navigate between related master and child records.

For example, imagine that you are working with two related tables in the Northwind database, Customers and Orders. Rather than specifying a join to combine both tables into a single result set, you would most commonly define two adapters, one to populate a Customers table in the dataset and a second adapter to read Order records into a different dataset table. The individual adapters would probably include selection criteria to limit the number of records in the data tables.

In the dataset you would also define a **DataRelation** object specifying that order records are related to customer records by the CustomerID field. You can still manage the tables individually, which would not be possible if you had joined tables before fetching records from the data source. For situations where you wanted to work with related records, you can invoke properties and methods of the **DataRelation** object.

For more information about data relations, see Relationships in ADO.NET Datasets.

Connection Objects

A data adapter needs an open connection to a data source to read and write data. Therefore, an adapter uses connection objects (SqlConnection, OleDbConnection, OdbcConnection, or OracleConnection)to communicate with a data source. (The adapter can contain up to four connection references, one for each type of action it can perform: Select, Update, Insert, and Delete.)

The following table lists connection objects in the Data tab of the Toolbox:

Connection Object	Description
SqlConnection	A connection to a SQL Server 7.0 or later database.
OleDbConnection	A connection to any OLE DB data source.
OdbcConnection	A connection to an ODBC data source.
OracleConnection	A connection to an Oracle database.

In all cases, the connection object represents a unique session within the data source. All connection objects provide properties to establish and modify connection details (such as user ID and password, and connection-timeout setting). They also provide methods to begin, commit, and roll back database transactions. For details about connection objects, see Connecting to Data Sources with ADO.NET.

> **Security Note** Storing connection-string details (such as the server name, user name, and password) can have implications for the security of your application. Using Windows integrated security is a more secure way to control access to a database. For more information, see Database Security.

ADO.NET Command Objects

Using an adapter, you can read, add, update, and delete records in a data source. To allow you to specify how each of these operations should occur, an adapter supports the following four properties:

- **SelectCommand** – reference to a command (SQL statement or stored procedure name) that retrieves rows from the data store.
- **InsertCommand** – reference to a command for inserting rows into the data store.
- **UpdateCommand** – reference to a command for modifying rows in the data store.
- **DeleteCommand** – reference to a command for deleting rows from the data store.

The properties are themselves objects — they are instances of the SqlCommand, OleDbCommand, OdbcCommand, or OracleCommand class. The objects support a **CommandText** property containing a reference to an SQL statement or stored procedure.

> **Note** The command class must match the connection class. For example, if you are using a **SqlConnection** object to communicate with a SQL Server, you must also use commands that derive from the **SqlCommand** class.

Although you can explicitly set the text of a command object, you do not always need to; in many situations, Visual Studio will generate the SQL statements needed. In addition, the adapter can automatically generate appropriate SQL statements at run time if the **UpdateCommand**, **InsertCommand**, or **DeleteCommand** objects are not specified. For details, see Automatically Generated Commands.

However, you can manipulate command objects at design time and run time in order to have more direct control over how the commands are executed. For example, you can create or modify the command associated with a **SelectCommand** object just before it is executed.

You can also execute commands yourself, independently of the data adapter. This allows you to pass arbitrary SQL commands through the data adapter, such as those used to define or modify database definitions. You can also call stored procedures directly that do not return record sets — for example, a stored procedure that validates a user entry against a database. For details, see Executing a Command.

> **Security Note** When using data commands with a CommandType property set to **Text**, carefully check information that is sent from a client before passing it to your database. Malicious users might try to send (inject) modified or additional SQL statements in an effort to gain unauthorized access or to damage the database. Before you transfer user input to a database, you should always verify that the information is valid; it is a best practice to always use parameterized queries or stored procedures when possible. For more information, see Scripting Exploits.

Command Parameters

The commands in a data adapter are usually parameter-driven. The command for the **SelectCommand** property, for example, often has a parameter in its WHERE clause so you can specify at run time what records to get from the database. The other commands use parameters that allow you to pass at run time the data to write into a record and what record in the database to update. For more information about how parameters are used in data adapters, see Parameters in Data-Adapter Commands.

Reading and Updating with Data Adapters

The primary purpose of the data adapter is to communicate data between a data store and a dataset. The adapter supports specific methods to move the data back and forth between the two.

> **Note** If you just want to read data (not update it), you do not have to store it in a dataset. Instead, you can read directly out of the database and into an application. For details, see "Read-Only Data" below.

You can use a data adapter to perform the following operations:

- Retrieve rows from a data store into corresponding data tables within the dataset.

  To retrieve rows into a dataset, use the **Fill** method on a data adapter object (SqlDataAdapter, OleDbDataAdapter, OdbcDataAdapter, or OracleDataAdapter). When you invoke the **Fill** method, it transmits an SQL SELECT statement to the data store.

- Transmit changes made to a dataset table to the corresponding data store.

  To transmit a dataset table of the dataset to the data store, use the adapter's **Update** method. When you invoke the method, it executes whatever SQL INSERT, UPDATE or DELETE statements are needed, depending on whether the affected record is new, changed, or deleted.

  For details about how updates are performed using data adapters, see Dataset Updates in Visual Studio .NET.

Read-Only Data

If your program needs to perform a sequential, read-only pass through a query result, you can use a data reader object in place of filling a dataset. A data reader object fetches the data from the data source and passes it through directly to your application. Typically, data reader objects are used for read-only, forward-only access to the data at times when you do not need to cache data in a dataset. (The data adapter itself uses a data reader object to populate a dataset.) An example is a Web Forms page that displays database information; because the Web Forms page is recreated with each roundtrip, it is often not useful to store the data in a dataset.

Visual Studio supplies four data reader objects, a SqlDataReader, OleDbDataReader, OdbcDataReader, and OracleDataReader. For details about using the data reader object for efficient read-only access, see Executing a Data Command That Returns a Result Set.

Table Mappings

By default, when you use Visual Studio tools to generate a dataset from database tables, the names of the tables and columns are the same in the dataset as in the database. However, you might find this impractical. For example, you might find the names used in the database too terse or verbose; or the names might be in a foreign language. If you are working with an existing schema, you might also find that the names defined in the schema do not match those used in your database.

Therefore, the names in the database and dataset do not have to match. Instead, you can create new table and column names in the dataset command, and then map those to the names used in the database. Adapters use the TableMappings collection to

maintain the correspondence between structures of the dataset (data tables and data columns) and structures of the data store (tables and columns). For details about table mappings, see Table Mapping in Data Adapters.

See Also

Introduction to Data Access with ADO.NET | Introduction to Datasets | Creating Data Adapters | Table Mapping in Data Adapters | Previewing Data-Adapter Results | Configuring Parameters for Data Adapters |
Mapping Data-Source Columns to Dataset Data-Table Columns

Parameters in Data-Adapter Commands

A data adapter's commands — those defined in the **CommandText** property of the **SelectCommand**, **InsertCommand**, **UpdateCommand**, and **DeleteCommand** objects — often involve parameters. At run time, parameters are used to pass values to the SQL statements or stored procedures represented by the commands.

Parameters are used in two contexts:

- Selection parameters - in production applications, you frequently fetch only a subset of the data in a database. To do so, you use an SQL statement or stored procedure that includes a WHERE clause with a parameter for selection criteria that you get at run time. In addition, when you update or delete records, you use a WHERE clause that pinpoints the record or records to be changed. The values used in the WHERE clause are usually derived at run time.
- Update parameters - when you update an existing record or insert a new one, the values for the columns in the changed or new record are established at run time. In addition, values used during optimistic concurrency checking are established using parameters.

> **Note** For Oracle, when using named parameters in an SQL statement or stored procedure, you must precede the parameter name with a colon (:). However, when referring to a named parameter elsewhere in your code (for example, when calling Add), do not precede the named parameter with a colon (:). The data provider supplies the colon automatically. For more information, see OracleParameter Class.

Selection Parameters

When selecting records to fill a dataset, you often include one or more parameters in the WHERE clause so you can specify at run time which records to fetch. For example, users might search a book database for a specific title keyword that they type into a Web page. To allow that, you might specify an SQL statement like the following as the **CommandText** property of a **SelectCommand**. Parameters are indicated either with a placeholder (a question mark) or with a named parameter variable. Parameters for queries involving **OleDbCommand and OdbcCommand** objects use question marks; queries that use **SqlCommand** objects use named parameters that begin with an @ symbol, whereas **OracleCommand** objects use named parameters that begin with a colon (:).

A query that uses placeholders might look like the following:

```
SELECT BookId, Title, Author, Price from BOOKS
WHERE (Title LIKE ?)
```

A query that uses **SqlCommand** named parameters might look like the following:

```
SELECT BookId, Title, Author, Price from BOOKS
WHERE (Title LIKE @title)
```

A query that uses **OracleCommand** named parameters might look like the following:

```
SELECT BookId, Title, Author, Price from BOOKS
WHERE (Title LIKE :title)
```

In your application, you prompt the user for a title keyword. You then set the parameter value and run the command.

> **Note** Occasionally, you might want to get the entire contents of a database table — for example, if you are setting up a lookup table — but generally, you want to fetch only the data you need in order to keep your application efficient.

In Visual Studio you can build SQL statements with parameters using the Query Builder. If you drag elements from Server Explorer, Visual Studio can configure parameters in some cases but not all, and you will need to complete the configuration manually.

Update Parameters

Whether or not an adapter's **SelectCommand** object contains a parameterized command, the commands for the **UpdateCommand**, **InsertCommand**, and **DeleteCommand** properties always do.

The commands for the **UpdateCommand** and **InsertCommand** properties need parameters for every column in the database to be updated. In addition, the **UpdateCommand** and **DeleteCommand** statements require a parameterized WHERE clause that

identifies the record to be updated, similar to the way the **SelectCommand** object is often configured.

Imagine an application where users can buy books. As users shop, they maintain a shopping cart, which is implemented as a data table. In the ShoppingCart table, users maintain a record for each book they want to buy, with the book ID and customer ID together acting as the key to the shopping cart record.

When users add a book to their shopping cart, the application might invoke an SQL INSERT statement. In the adapter, the statement's syntax might look like the following:

```
INSERT INTO ShoppingCart
   (BookId, CustId, Quantity)
Values (?, ?, ?)
```

The three question marks represent parameter placeholders that will be filled in at run time with values for the customer ID, book ID, and quantity. If you are using named parameters, the same query might look like this:

```
INSERT INTO ShoppingCart
   (BookId, CustId, Quantity)
Values (@bookid, @custid, @quantity)
```

If the user decides to change something about an item in the shopping cart — for example, by changing the quantity — the application might invoke an SQL UPDATE statement. The syntax of the statement might be as follows:

```
UPDATE ShoppingCart
   SET (BookId = ?, CustId = ?, Quantity = ?)
WHERE (BookId = ? AND CustId = ?)
```

Or if you are using named parameters, it might be as follows:

```
UPDATE ShoppingCart
   SET (BookId = @bookid, CustId = @custid, Quantity = @quantity)
WHERE (BookId = @bookid AND CustId = @custid)
```

In this statement, the parameters in the SET clause are filled in with updated values for the changed record. The parameters in the WHERE clause identify which record to update and are filled in with the original values from the record.

A user might also remove an item from the shopping cart. In that case, the application might invoke an SQL DELETE statement with syntax such as the following, if you are using parameter placeholders:

```
DELETE FROM ShoppingCart
WHERE (BookId = ? AND CustId = ?)
```

Or the following if you are using named parameters:

```
DELETE FROM ShoppingCart
WHERE (BookId = @bookid AND CustId = @custid)
```

The Parameters Collection and Parameter Objects

To allow you to pass parameter values at run time, each of the four command objects for a data adapter supports a **Parameters** property. The property contains a collection of individual parameter objects that correspond one-to-one with the placeholders in a statement.

The following table shows the corresponding parameter collection for each data adapter:

Data adapter	Parameter collection
SqlDataAdapter	SqlParameterCollection
OleDbDataAdapter	OleDbParameterCollection
OdbcDataAdapter	OdbcParameterCollection
OracleDataAdapter	OracleParameterCollection

Note For Oracle, when using named parameters in an SQL statement or stored procedure, you must precede the parameter name with a colon (:). However, when referring to a named parameter elsewhere in your code (for example, when calling Add), do not precede the named parameter with a colon (:). The .NET Framework Data Provider for Oracle supplies the colon automatically.

By using the parameters collection, you save yourself the trouble of having to manually construct a SQL command as a string with run-time values; in addition, you get the benefit of type checking in your parameters.

If you use the Data Adapter Configuration Wizard to configure the adapter, the parameters collection is set up and configured automatically for all four adapter commands. If you drag elements from Server Explorer onto the form or component, Visual Studio can perform the following configurations:

- If you drag a table or some columns onto the designer, Visual Studio generates a **SelectCommand** object (specifically, an SQL SELECT statement) with no parameters, and parameterized **UpdateCommand**, **InsertCommand**, and **DeleteCommand** objects. If you want the **SelectCommand** object statement to have parameters, you must configure them manually.
- If you drag a stored procedure onto the designer, Visual Studio generates a **SelectCommand** object, with parameters as required by the stored procedure. However, if you need them, you must configure the **UpdateCommand**, **InsertCommand**, and **DeleteCommand** objects yourself, along with their parameters.

Generally speaking, if you want to create parameterized queries for the adapter, you should use the Data Adapter Configuration Wizard. However, should you need to, you can configure parameters manually using the Properties window.

Structure of the Parameters Collection

The items in a command's parameters collection correspond one-to-one to parameters required for the corresponding command object. If the command object is an SQL statement, the items in the collection correspond to the placeholders (question marks) in the statement. The following UPDATE statement requires a collection of five parameter items:

```
UPDATE ShoppingCart
   SET (BookId = ?, CustId = ?, Quantity = ?)
WHERE (BookId = ? AND CustId = ?)
```

Here is the same statement with named parameters:

```
UPDATE ShoppingCart
   SET (BookId = @bookid, CustId = @custid, Quantity = @quantity)
WHERE (BookId = @bookid AND CustId = @custid)
```

If the command object references a stored procedure, the number of parameter items in the collection is determined by the procedure itself. The parameters might not correspond exactly to the placeholders in an SQL statement.

In stored procedures, parameters can also be named. In that case, the position of a parameter in the collection is not important. Instead, each parameter item in the collection has a **ParameterName** property that is used to match it to the corresponding parameter in the stored procedure.

If you are configuring the parameters collection manually, you must understand exactly which parameters the stored procedure requires. Many stored procedures return a value; if so, the value is passed back to your application in the parameters collection, so you must allow for that. In addition, some stored procedures include multiple SQL statements, and you must be sure that the parameters collection reflects all the values passed to all statements in the procedure.

If parameters are not named (as in stored procedures), the items in the collection map positionally to the parameters required by the command. If the command is a stored procedure and it returns a value, the first item in the collection (item zero) is reserved for this return value.

You can therefore reference individual parameter objects by index position in the collection. However, parameter objects also support a **ParameterName** property that provides a way to reference parameters independently of their order. For example, the following two statements might be equivalent (assuming that the second parameter in the collection is named `Title_Keyword`):

```
' Visual Basic
' Encloses the keyword in SQL wildcard characters.
titleKeyword = "%" & txtTitleKeyword.Text & "%"
OleDbDataAdapter1.SelectCommand.Parameters(1).Value = titleKeyword
OleDbDataAdapter1.SelectCommand.Parameters("Title_Keyword").Value = titleKeyword
```

```
// C#
// Encloses the keyword in SQL wildcard characters.
string titleKeyword = "%" + txtTitleKeyword.Text + "%";
this.OleDbDataAdapter1.SelectCommand.Parameters[1].Value = titleKeyword;
this.OleDbDataAdapter1.SelectCommand.Parameters["Title_Keyword"].Value = titleKeyword;
```

Using a parameter name is generally a much better programming practice than referencing parameters by index value, because it reduces the need for maintenance if the number of parameters changes and frees you from having to remember whether a stored procedure returns a value. There is slight additional overhead in referencing a parameter by name rather than index value, but this can be offset by ease of programming and by application maintainability.

Establishing Parameter Values

There are two ways that you can establish the value of a parameter:

- By setting the parameter's **Value** property explicitly.
- By mapping parameters to columns in a dataset table, so that the values can be extracted from data rows when needed.

You set the parameter value explicitly when you are filling a dataset or calling a command — that is, for selection parameters. For example, in the example above of searching for books, the application might have a text box where users enter a title keyword. You would then explicitly set the value of the parameter to the text of the text box before calling the adapter's **Fill** method. Code to do this might look like the following, which establishes the contents of a text box as a parameter before filling a dataset.

```
' Visual Basic
' Encloses the keyword in SQL wildcard characters.
titleKeyword = "%" & txtTitleKeyword.Text & "%"
OleDbDataAdapter1.SelectCommand.Parameters("Title_Keyword").Value = titleKeyword
OleDbDataAdapter1.Fill(dsAuthors1)

// C#
// Encloses the keyword in SQL wildcard characters.
titleKeyword = "%" + txtTitleKeyword.Text + "%";
this.OleDbDataAdapter1.SelectCommand.Parameters["Title_Keyword"].Value = titleKeyword;
this.OleDbDataAdapter1.Fill(dsAuthors1);
```

Mapped parameter values are used during updates. When you call an adapter's **Update** method, the method walks through the records in a dataset table, individually making the appropriate update (update, insert, delete) for each record. In that case, the parameter values are already available as columns in the dataset records. For example, when the update process gets to a new record in the dataset table — a record for which it must call an INSERT statement in the database — the values for the INSERT statement's VALUE clause can be read directly out of the record.

These are typical scenarios, but not the only ones. Stored procedures sometimes return data using out parameters or via the procedure's return value. If so, the returned values should be mapped to columns in a dataset table.

It is possible to set update parameters explicitly as well. The adapter supports a **RowUpdating** event that is called each time a row is being updated. You can create a handler for this event and set parameter values there. This gives you very precise control over parameter values, and allows you to perform processes such as creating parameter values dynamically before they are written to a database record.

See Also

Introduction to Data Adapters | Introduction to Datasets | Creating Data Adapters | Configuring Parameters for Data Adapters | Code: Calling a Parameterized Stored Procedure (Visual Basic)

Table Mapping in Data Adapters

When a data adapter reads data from a data source, it determines where to put the data in the corresponding dataset table (or tables) using a table mapping. The mapping links the names of columns in the source with those in the dataset table. For example, information from a column called `au_id` in the data source might belong in a column called `author_id_number` in the dataset table.

By default, when you use Visual Studio tools to generate a dataset from information in a data source, the dataset elements have the same names as they have in the source. However, there are times when the names in the data source and the data set will not match:

- The dataset was created from an existing schema that uses different names.
- You want to change data element names in the dataset for convenience, readability, translation to or from a foreign language, or another reason.
- You want to control the names of typed data members when you generate a dataset from the adapter.

Table Mappings Structure

Table mappings are established using the adapter's TableMappings property, which is a collection of items of type DataTableMapping. There is one **DataTableMapping** object for each set of tables being mapped. Normally there is just one such item, because adapters are normally associated only with a single source table and a single dataset table. (However, a stored procedure can return multiple row sets; if so, the second and subsequent tables are mapped using the second and subsequent mappings in the **TableMappings** collection.)

Each mapping item in turn contains properties to identify the data-source table, the dataset table, and a **ColumnMappings** property containing items that represent the actual mappings.

Table Mapping in Action

When you call a data adapter's **Fill** method, the adapter goes through the following process to determine where in the dataset to write the data:

1. The adapter looks up each source column name in the **TableMappings** object.
2. When it finds the source column name, it gets the matching (mapped) name for the corresponding column in the dataset table.
3. Using the name it got from Step 2, the adapter writes the data from the source column to the corresponding dataset column.

There are a variety of conditions that can prevent the adapter from following the process outlined above. The two primary ones are:

- No mapping can be found for a source column. This might be because there is nothing defined for the **TableMappings** property, or because a specific column is not mapped.
- The column to be written is not defined in the dataset's schema, whether or not it is mapped in the **TableMappings** property.

The adapter supports two properties that allow you specify what should happen when either of these situations occurs. The situations are not necessarily error conditions, because the adapter can still fill the dataset even if either situation occurs.

The MissingMappingAction property allows you to specify what action the adapter should take if the mapping is missing. Possible settings are:

- **Passthrough** - the adapter attempts to load the column into a dataset column of the same name. If there is no dataset column of that name, the behavior depends on the **MissingSchemaAction** setting (see below).
- **Ignore** - columns that are not mapped properly are not loaded into the dataset.
- **Error** - raises an error.

The MissingSchemaAction property allows you to specify what should happen when the adapter tries to write data to a column that is not defined in the dataset's schema. Possible values are:

- **Add** - the table or column is added to the schema and to the dataset.
- **AddWithKey** - the table or column is added to the dataset and schema along with information about the primary key.
- **Ignore** - the table or column not represented in the dataset schema is not added to the dataset.

- **Error** - the adapter raises an error.

You generally set both properties in conjunction to accommodate your application's specific needs. Setting the **MissingMappingAction** property to **Passthrough** and the **MissingSchemaAction** to **Add** has the effect of automatically duplicating the table and column names from the source in the dataset.

Conversely, you might specify error checking in applications where the dataset schema is strictly defined. In such a case, getting data from a source without a clearly defined target column in the dataset might constitute violation of a business rule or other error.

You specify **Ignore** when you want to ensure that the only data loaded into the dataset is that which you have explicitly defined in the schema or mapped in the **TableMappings** property. This is useful if the adapter calls a stored procedure or SQL statement that returns more columns than you need in the dataset.

See Also

Introduction to Data Adapters | Mapping Data-Source Columns to Dataset Data-Table Columns | Table Mappings Dialog Box

Creating Data Adapters

Data adapters define the means by which you can get information from one or more tables in the data source into a dataset and back again. For an overview, see Introduction to Data Adapters.

In Visual Studio there are a number of ways you can create and configure data adapters: using Server Explorer, a wizard, "manually" (configuring it with the Properties window), and in code. The topics listed below provide information on each method.

In This Section

Creating Data Adapters Using Server Explorer
Explains how you can drag database elements onto a form or component and generate the adapter automatically.
Creating Data Adapters Using a Wizard
Explains how the Data Adapter Configuration Wizard prompts you for all the information required to configure the data adapter (including parameters) and, if necessary, create a connection.
Creating and Configuring Data Adapters Manually
Explains how to drag a data adapter from the Toolbox, and then configure it yourself using the Properties window.
Populating a DataSet from a DataAdapter
Explains how to create an instance of an adapter in code and use it to fill a dataset.

Related Sections

Introduction to Data Adapters
Provides an overview of what data adapters are, what they are used for, how they are structured.
Introduction to Data Access with ADO.NET
Provides a general introduction to the concepts of data access in Visual Studio.
Introduction to Datasets
Provides background information on the structure and use of datasets.
Creating and Filling Datasets Example Topics
Presents a series of specific example code topics for common tasks in creating and filling datasets with Visual Basic .NET.
Manipulating Data Example Topics
Presents a series of specific example code topics for common tasks in working with data within your application using Visual Basic .NET.

Creating Data Adapters Using Server Explorer

This procedure describes creating data adapters from Server Explorer. Often, you will want to make configuration changes to the adapter after creating it. For details, see Creating and Configuring Data Adapters Manually.

To create a data adapter with Server Explorer

1. Open the form or component in the appropriate designer and then open Server Explorer. For details, see Accessing and Initializing Server Explorer.
2. If the connection you need does not already exist, create it. For details, see Adding New Data Connections in Server Explorer.

   Security Note Storing connection-string details (such as the server name, user name, and password) can have implications for the security of your application. Using Windows integrated security is a more secure way to control access to a database. For more information, see Database Security.

3. Open the connection node and then expand any of the element nodes (**Tables**, **Stored Procedures**, **Views**, or **Functions**). If you are working with tables, expand the table name node to display the table columns.

   Note If you are working with stored procedures, you will need to manually configure the **UpdateCommand**, **InsertCommand**, and **DeleteCommand** properties of the adapter after creating it. For details, see Creating and Configuring Data Adapters Manually.

4. Select one or more columns (under **Table**) or a stored procedure, and then drag the selection onto the designer.

   Note You can drag the table name to the designer instead of selecting and dragging individual columns. Doing so results in an SQL SELECT * FROM tablename query. Selecting individual columns is generally more efficient, because you limit the amount of data that is transferred between the database and the dataset.

   When you drop the selection onto the designer, Visual Studio creates an instance of a connection object for that data source (if there is not already one there) and a data adapter object.

   If you dragged a table or individual columns onto the designer, the adapter is configured for both reading and updating data. If you dragged a stored procedure onto the designer, the adapter is configured for reading data. For details about configuring the commands for updating, see Creating and Configuring Data Adapters Manually.

5. If you want to see how the data adapter will fill a dataset, preview the results. For details, see Previewing Data-Adapter Results.

   At this point, you can create a dataset based on information in the adapter. For details, see Creating Typed Datasets with the Component Designer.

Dragging data elements from Server Explorer is easy, but does not give you the opportunity to create a parameterized query or to create new stored procedures to access data. For that, you can use the Data Adapter Configuration Wizard. For details, see Creating Data Adapters Using a Wizard.

See Also

Creating and Configuring Data Adapters Manually | Populating a DataSet from a DataAdapter | Previewing Data-Adapter Results | Configuring Parameters for Data Adapters

Creating Data Adapters Using a Wizard

The Data Adapter Configuration Wizard provides the simplest and most flexible way to create data adapters. For details, see Data Adapter Configuration Wizard.

To create a data adapter using the Data Adapter Configuration Wizard

1. Open the form or component you are creating in the appropriate designer.
2. From the **Data** Tab of the Toolbox, drag an OleDbDataAdapter, SqlDataAdapter, OdbcDataAdapter, or OracleDataAdapter object onto the design surface.

   The designer adds an instance of the adapter to the form or component and launches the Data Adapter Configuration Wizard.

3. In the wizard, do the following:
   a. In the second page, create or choose a connection. For more information about creating connections, see Creating ADO.NET Connection Objects.

      Security Note Storing connection-string details (such as the server name, user name, and password) can have implications for the security of your application. Using Windows Integrated Security is a more secure way to control access to a database. For more information, see Database Security.

   b. In the third page, specify whether the adapter should use SQL statements or stored procedures to read and write data. Choose one of the options listed in the following table.

Option	Description
Use SQL statement	You will define an SQL SELECT statement that the adapter will use to populate a table in the dataset. Based on your SELECT statement, the wizard will also generate UPDATE, INSERT, and DELETE statements for updating the data source.
Use newly created stored procedures	You specify a SELECT statement, and the wizard creates stored procedures from the statement to read and update the data source. If the provider does not support this option, it is grayed out.
Use existing stored procedures	You will specify existing stored procedures that the adapter will use to read and update the data source.

   The choice you make determines what appears on the following pane.

   c. In the fourth page, either create the SELECT statement or select the existing stored procedures. For assistance building the SQL statement, click **SQL Builder** to launch the **Query Builder**.

      Note You can create a query with a parameter placeholders in the WHERE clause, which will be filled with information you gather at run time. For details, see Parameters in Data-Adapter Commands.

   When you have finished the wizard, it generates an instance of a connection object with the information you specified in the second page.

4. If you want to change the name of the adapter and connection, select them individually in the designer and give them each a new name in the Properties window.
5. If you want to see what data the adapter will fill into the dataset, you can preview it. For details, see Previewing Data-Adapter Results.

You can also create and configure an adapter manually. To do so, you must already have a connection object available. Part of configuring the data adapter involves establishing table mappings between tables in the data source and the dataset. For details, see Creating and Configuring Data Adapters Manually.

See Also

Creating Data Adapters Using Server Explorer | Creating and Configuring Data Adapters Manually | Populating a DataSet from a DataAdapter | Previewing Data-Adapter Results | Configuring Parameters for Data Adapters

Creating and Configuring Data Adapters Manually

You can create data adapters "manually" — that is, without using a wizard. No matter how you have created the adapter, you can always make configuration changes to it.

To create a data adapter manually

1. Make sure a connection object is available to the form or component that you are working with. For details about adding a standalone connection, see Creating ADO.NET Connection Objects.
2. From the **Data** Tab of the Toolbox, drag an OleDbDataAdapter, SqlDataAdapter, OdbcDataAdapter, or OracleDataAdapter object onto the design surface.

   The designer adds an instance of the adapter to the form or component and launches the Data Adapter Configuration Wizard.

3. Close the wizard.

To configure a data adapter manually

1. Select the adapter, and then in the Properties window, configure the commands used to read and update data. You must configure the **SelectCommand** object. If the adapter will be used to update a data source, you must also configure the **UpdateCommand**, **DeleteCommand**, and **InsertCommand** objects.

   For each command object, set the following properties.

Property	Description
ActiveConnection	Set to reference a connection object. You can create a connection object from the **ActiveConnection** property in the Properties window. Typically, each command object will reference the same connection object, but you have the option of using different connections for each command if your application requires it.
CommandText	Either the text of an SQL statement or the name of a stored procedure. For some providers, you can include multiple statements or stored procedure names in the **CommandText** property, delimited with semicolons (;). The statements or procedures are executed sequentially. This is useful for situations such as issuing a SELECT statement after an update or INSERT statement in order to get back refreshed records and any auto-generated values (such as default or auto-increment values). Not all providers support this facility. For details about updating from datasets, see Dataset Updates in Visual Studio .NET.
CommandType	A value indicating how the value of the **CommandText** property should be interpreted: • **Text** - the command is an SQL statement. • **StoredProcedure** - the command is a reference to a stored procedure. • **TableDirect** - the text is not a command, but the name of a table.
Parameters	A collection of objects of type **Parameter** that you configure to pass values to the command. In the **SelectCommand** object, you create a parameters collection only if the command includes a parameter placeholder. **UpdateCommand**, **InsertCommand**, and **DeleteCommand** objects always require parameters. For details, see Configuring Parameters for Data Adapters.

2. If you do not want the same column names in the data source and dataset, change the mappings. The default value for the **MissingMappingAction** property (Passthrough) will generate the same names in the data source and the dataset automatically. For details, see Mapping Data-Source Columns to Dataset Data-Table Columns.
3. From the **Data** menu, choose **Generate DataSet**.

   Tip If you do not see the **Data** menu, click anywhere in the form or component designer; the menu only appears if the designer has the focus.

4. If you want to see how the data adapter will fill a dataset, preview the results. For details, see Previewing Data-Adapter Results.

See Also

Creating Data Adapters Using Server Explorer | Creating Data Adapters Using a Wizard | Populating a DataSet from a DataAdapter | Previewing Data-Adapter Results | Configuring Parameters for Data Adapters | Code: Creating a DataAdapter (Visual Basic):

Configuring Parameters for Data Adapters

Under most circumstances, a data adapter's **Command** objects (**SelectCommand**, **InsertCommand**, **UpdateCommand**, and **DeleteCommand**) reference SQL statements or stored procedures that require parameters. If you have used the Data Adapter Configuration Wizard to create the adapter, then the parameters for these commands are configured for you automatically. However, you might want to configure parameters manually or make small changes to an existing parameters collection.

> **Tip** To configure parameters after making changes to the SQL statement for a command, you can choose the **Regenerate parameters collection for this command** option in the Query Builder.

To configure parameters for a data adapter

1. Create a data adapter. For details, see Creating Data Adapters.
2. In the form or component designer, select the data adapter and then open the Properties window.
3. Expand the command object for which you want to configure parameters (for example, the **UpdateCommand**) and then in the **Parameters** property, click the ellipsis button (...) to open the **Parameter Collection Editor** window.
4. To create a new parameter object, click **Add**.
5. Place the new parameter at the correct index location in the collection for that parameter by clicking the arrows under **Sort** to move it.
6. To set parameter properties, select the parameter in the **Members** list, and then use the property grid on the right-hand side. The following table lists properties that you typically set.

Property	Description
SourceColumn	The name of a column in a dataset table from which the parameter value will be read. This property is used with parameters that fill values in an UPDATE, INSERT, or DELETE statement (or equivalent stored procedures).
SourceVersion	If the parameter value is being derived via the **SourceColumn** property, **SourceVersion** specifies which version of a dataset record should be used as the parameter value source: • **Original** - the value of the column as it was read from the database. You use this option when you need the value of a column before it has changed. An example is when a primary key is changing but you need the old key to find the record in the database. • **Current** - the value of the column after a record has been updated. This is usually the value to be written to the database. • **Default** - the value to which a column is set if no value is explicitly supplied. For details about different versions of records in the dataset, see Dataset Updates in Visual Studio .NET.
Value	An explicit value to set for the parameter. This property is often set at run time rather than statically at design time. The **Value** property takes precedence over the **SourceColumn** property if both are set.
NamedParameter	An Boolean value indicating whether the parameter corresponds to a variable (typically with the format @*parametername*) or to a placeholder (typically a question mark). If you are working with a **SqlConnection** object and are using named parameters, set this to **true**.
DBType, Precision, Scale, Size	Information about the native data type (in the data store) of the parameter value. The parameter value is converted into or out of the type you specify.
Direction	A specification for whether the parameter value is being passed to the command or back: • **Input** - the value is being passed to the command. This is the default. • **Output** - the value is being passed back from a stored procedure. (Not applicable for SQL statements.) • **InputOutput** - the value is passed to a stored procedure and passed back, usually with a modified value. • **ReturnValue** - the value is an explicit return value. By default, the returned value is the first item (index value zero) in the parameters collection.
Parameter Name	A name by which you can refer to the parameter in the collection, instead of referencing it by its index value. Although not required, a parameter name is recommended for ease of maintenance.

7. Click **OK** to close the **Parameter Collection Editor** window.
8. Repeat Steps 3 through 7 to configure parameters for other command objects.

See Also

Parameters in Data-Adapter Commands

Mapping Data-Source Columns to Dataset Data-Table Columns

Creating a mapping in a data adapter allows you to establish a correspondence between columns in the data source and columns in a dataset table. For details, see Table Mapping in Data Adapters.

To map data-source columns to a dataset table

1. Create a data adapter. For details, see Creating Data Adapters.
2. In a form or component designer, select the data adapter and open the Properties window.
3. Next to the **TableMappings** property, click the ellipsis button (...).

   The **Mappings** dialog box opens.

4. If you want to map column names from the data source to an existing dataset, check **Use a Dataset to suggest table and column names**, and then select a dataset from the **Dataset** list.

   The tables available in that dataset are listed in the **Dataset table** list, and the columns of the first table in the dataset are filled into the right-hand side of the **Column mappings** grid.

   > **Note** Choosing a dataset is a convenience that allows the **Mappings** dialog box to establish defaults for the dataset columns to map to. The dataset is not part of the information maintained for the mappings in the adapter. Moreover, you might use the same mappings with more than one dataset.

5. Under **Source table**, select the table in the data source to map. If there is only one table in the dataset, the default value is **Table**.
6. Under **Dataset table**, select the table in the dataset to map. If you selected a dataset to map to in Step 3, a name will already be filled in. (By default, the first table in the dataset is referred to as **Table**.)

   Based on your choices (or the defaults), values in the **Source Columns** and **Dataset Columns** lists are filled in. The grid indicates how mapping will be performed: each column in the left-hand column will be matched to the corresponding column in the right-hand column.

7. Adjust the mappings as needed. This can include:

   - Selecting different columns in either the source or dataset column lists to change which columns map to which.
   - Removing columns from the mapping. You might do this if the adapter is returning more columns from the data source than what you need.
   - Adding columns. You might do this if you know that columns will exist at run time that are not represented at design time, or if you deleted a column and now want it back, or if you refined the adapter's query and now need to adjust the mappings.

8. When you have finished adjusting the mappings, click **OK**.
9. If you want to specify how to handle possible error conditions, in the Properties window, select options for the **MissingMappingAction** and **MissingSchemaAction** properties. The following table lists some common settings and their effect.

MissingMappingAction	**MissingSchemaAction**	**Effect**
Passthrough	**Add** or **AddWithKey**	If a column exists in the source but not in the schema and there is no mapping for it, the column is created when the dataset is filled. This is the default setting, and is useful when you simply want to copy all the columns available in the source to the dataset.
Ignore	**Ignore**	If the column is not defined in the dataset schema, or if there is no mapping for it, information is not copied from the source to the dataset. This setting is useful if the adapter returns more information than you want in the dataset.
Error	**Error**	A run-time error is raised if the adapter cannot map a source column to a dataset column because the mapping is missing or because the dataset schema does not contain the column. This setting is useful when you are working with a stringently defined schema and want to be sure that any deviations from the schema are flagged.

See Also

Previewing Data-Adapter Results

After configuring a data adapter, you can test how the adapter will fill a dataset.

To preview data-adapter results

1. Create a data adapter. For details, see Creating Data Adapters.
2. In the form or component designer, select the data adapter and then open the Properties window.
3. At the bottom of the window, click **Preview**.

   The **Data Preview** window opens.

4. Select a data adapter from the **Data Adapter** list, or choose **All DataAdapters** to see the effect of completely populating the dataset.
5. If the command referenced by the adapter's **SelectCommand** property requires parameters, fill in test parameter values in the **Parameters** grid.
6. Click **Fill Dataset** to perform the command.

   The results of the **SelectCommand** statement or stored procedure are displayed under Data. If the adapter gets data from more than one table in the data source (as might happen if it references a stored procedure that returns multiple results sets), the list of tables cataloged under Data tables appears twice. You can choose the table to preview by selecting it in the list.

7. Repeat for other adapters or with other parameter values. By default, each result set generated by the preview is appended onto others already in the dataset preview. To start with an empty dataset, click **Clear Table**.

See Also

Creating Data Adapters

ADO.NET Datasets

Datasets are containers — caches — in which you can store data to use in your application. Datasets are a fundamental part of the ADO.NET architecture, providing both high-performance data access as well as scalability.

The following topics provide information about ADO.NET **DataSet** objects and how to work with them in Visual Studio.

In This Section

Introduction to Datasets
Provides an overview of what ADO.NET datasets are, how they can by typed or untyped, what they are used for, and how they are structured.
Visual Studio Tools for Creating Datasets
Describes what features Visual Studios provides to allow you to create datasets and when you would choose which tool.
Creating Typed Datasets with the Component Designer
Describes how to create datasets when working in a form or component.
Creating Dataset Columns Using Expressions
Describes how to define columns in a dataset table that are based on calculations or other expressions rather than based directly on data in a data source.
Adding Tables to an Existing Dataset
Describes how to create new data tables in a typed dataset created earlier.
Adding Existing Typed Datasets to a Form or Component
Describes how to create an instance of a dataset on a form or component.
Adding Untyped Datasets to a Form or Component
Describes how to create a dataset that has no associated schema.
Handling Null Values in a Dataset
Explains how to control the representation of columns that contain null values.
Walkthrough: Mapping Data Source Tables to Dataset Tables
Describes how to load data from a database into a dataset based on a different schema using data adapter table and column mappings.

Related Topics

DataSet Class
Provides reference information about the **DataSet** class and links to topics about each member of the class.
Introduction to XML Schemas
Provides details about the XML structure of dataset schemas.
Introduction to Data Adapters
Provides an overview of what data adapters are, what they are used for, and how they are structured.
Introduction to DataRelation Objects
Provides an overview of how datasets allow you to specify relationships between tables and how you can take advantage of these relationships.
Creating XML Schemas and Datasets
Lists topics that explain how you can use the XML Designer to create datasets.
Populating a DataSet from a DataAdapter
Describes how to call data-adapter methods to execute SQL commands that provide the data for a dataset.
Adding Data to a Table
Describes how to programmatically create rows in dataset tables.
Introduction to Dataset Updates
Provides an overview of the process and options for writing data to a dataset and from the dataset to a data source.
Windows Forms Data Architecture
Lists topics that provide information on data binding in Windows Forms.
Web Forms Data Binding
Lists topics that provide information on data binding in Web Forms Pages.
Data Walkthroughs
Provides links to step-by-step topics you can read and follow to learn more about using data in Visual Studio.
Creating and Filling Datasets Example Topics
Presents a series of specific example code topics for common tasks in creating and filling datasets with Visual Basic .NET.
Manipulating Data Example Topics
Presents a series of specific example code topics for common tasks in working with data within your application using Visual Basic .NET.

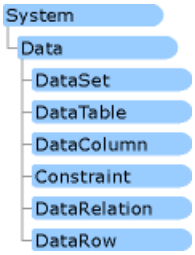
Introduction to Datasets

Datasets store data in a disconnected cache. The structure of a dataset is similar to that of a relational database; it exposes a hierarchical object model of tables, rows, and columns. In addition, it contains constraints and relationships defined for the dataset.

> **Note** You use datasets if you want to work with a set of tables and rows while disconnected from the data source. Using a dataset is not always an optimal solution for designing data access. For more information, see Recommendations for Data Access Strategies.

You can create and manipulate datasets using the following portions of the .NET Framework namespaces.

Dataset Namespace



The fundamental parts of a dataset are exposed to you through standard programming constructs such as properties and collections. For example:

- The DataSet class includes the Tables collection of data tables and the Relations collection of DataRelation objects.
- The DataTable class includes the Rows collection of table rows, the Columns collection of data columns, and the ChildRelations and ParentRelations collections of data relations.
- The DataRow class includes the RowState property, whose values indicate whether and how the row has been changed since the data table was first loaded from the database. Possible values for the RowState property include **Deleted**, **Modified**, **New**, and **Unchanged**.

Datasets, Schemas, and XML

An ADO.NET dataset is one view — a relational view — of data that can be represented in XML. In Visual Studio and the .NET Framework, XML is the format for storing and transmitting data of all kinds. As such, datasets have a close affinity with XML. This relationship between datasets and XML enables you to take advantage of the following features of datasets:

- The structure of a dataset — its tables, columns, relationships, and constraints — can be defined in an XML Schema. XML Schemas are a standards-based format of the W3C (World Wide Web Consortium) for defining the structure of XML data. Datasets can read and write schemas that store structured information using the ReadXmlSchema and WriteXmlSchema methods. If no schema is available, the dataset can infer one (via its InferXmlSchema method) from data in an XML document that is structured in a relational way. For more information about datasets and schemas, see Introduction to XML Schemas.
- You can generate a dataset class that incorporates schema information to define its data structures (such as tables and columns) as class members. (See "Typed versus Untyped Datasets" below).
- You can read an XML document or stream into a dataset using the dataset's ReadXML method and write a dataset out as XML using the dataset's WriteXML method. Because XML is a standard interchange format for data between different applications, this means that you can load a dataset with XML-formatted information sent by other applications. Similarly, a dataset can write out its data as an XML stream or document, to be shared with other applications or simply stored in a standard format.
- You can create an XML view (an XMLDataDocument object) of the contents of a dataset, and then view and manipulate the data using either relational methods (via the dataset) or XML methods. The two views are automatically synchronized as they are changed.

Typed versus Untyped Datasets

Datasets can be typed or untyped. A typed dataset is a dataset that is first derived from the base **DataSet** class and then uses information in an XML Schema file (an .xsd file) to generate a new class. Information from the schema (tables, columns, and so on) is generated and compiled into this new dataset class as a set of first-class objects and properties.

> **Note** For more information about dataset schemas, see XML Schemas and Data.

Because a typed **DataSet** class inherits from the base **DataSet** class, the typed class assumes all of the functionality of the **DataSet** class and can be used with methods that take an instance of a **DataSet** class as a parameter

An untyped dataset, in contrast, has no corresponding built-in schema. As in a typed dataset, an untyped dataset contains tables, columns, and so on — but those are exposed only as collections. (However, after manually creating the tables and other data elements in an untyped dataset, you can export the dataset's structure as a schema using the dataset's WriteXmlSchema method.)

You can use either type of dataset in your applications. However, Visual Studio has more tool support for typed datasets, and they make programming with the dataset easier and less error-prone.

Contrasting Data Access in Typed and Untyped Datasets

The class for a typed dataset has an object model in which its tables and columns become first-class objects in the object model. For example, if you are working with a typed dataset, you can reference a column using code such as the following:

```
' Visual Basic
' This accesses the CustomerID column in the first row of
' the Customers table.
Dim s As String
s = dsCustomersOrders1.Customers(0).CustomerID

// C#
// This accesses the CustomerID column in the first row of
// the Customers table.
string s;
s = dsCustomersOrders1.Customers[0].CustomerID;
```

In contrast, if you are working with an untyped dataset, the equivalent code is:

```
' Visual Basic
Dim s As String
s = CType(dsCustomersOrders1.Tables("Customers").Rows(0).Item("CustomerID"), String)

// C#
string s = (string) dsCustomersOrders1.Tables["Customers"].Rows[0]["CustomerID"];
```

Typed access is not only easier to read, but is fully supported by IntelliSense in the Visual Studio Code Editor. In addition to being easier to work with, the syntax for the typed dataset provides type checking at compile time, greatly reducing the possibility of errors in assigning values to dataset members. Access to tables and columns in a typed dataset is also slightly faster at run time because access is determined at compile time, not through collections at run time.

Even though typed datasets have many advantages, there are a variety of circumstances under which an untyped dataset is useful. The most obvious scenario is that no schema is available for the dataset. This might occur, for example, if your application is interacting with a component that returns a dataset, but you do not know in advance what its structure is. Similarly, there are times when you are working with data that does not have a static, predictable structure; in that case, it is impractical to use a typed dataset, because you would have to regenerate the typed dataset class with each change in the data structure.

More generally, there are many times when you might create a dataset dynamically without having a schema available. In that case, the dataset is simply a convenient structure in which you can keep information, as long as the data can be represented in a relational way. At the same time, you can take advantage of the dataset's capabilities, such as the ability to serialize the information to pass to another process, or to write out an XML file.

Dataset Case Sensitivity

Within a dataset, table and column names are by default case-insensitive — that is, a table in a dataset called "Customers" can also be referred to as "customers." This matches the naming conventions in many databases, including SQL Server, where the names of data elements cannot be distinguished only by case.

> **Note** Unlike datasets, XML documents are case-sensitive, so the names of data elements defined in schemas are case-sensitive. For example, schema protocol allows the schema to contain define a table called "Customers" and a different table called "customers." This can result in name collisions when a schema is used to generate a dataset class. For more information, see XML Elements, Attributes, and Types.

However, case sensitivity can be a factor in how data is interpreted within the dataset. For example, if you filter data in a dataset table, the search criteria might return different results depending on whether the comparison is case-sensitive or not. You can

control the case sensitivity of filtering, searching, and sorting by setting the dataset's CaseSensitive property. All the tables in the dataset inherit the value of this property by default. (You can override this property for each individual table.)

Populating Datasets

A dataset is a container; therefore, you need to fill it with data. When you populate a dataset, various events are raised, constraint checking applies, and so on. For more information about updating a dataset and the issues surrounding updates, see Dataset Updates in Visual Studio .NET.

You can populate a dataset in a variety of ways:

- Call the **Fill** method of a data adapter. This causes the adapter to execute an SQL statement or stored procedure and fill the results into a table in the dataset. If the dataset contains multiple tables, you probably have separate data adapters for each table, and must therefore call each adapter's **Fill** method separately.

  For more information about how datasets are filled, see Introduction to Data Adapters and Creating Data Adapters. For more information on how to use a data adapter to populate a dataset, see Populating a DataSet from a DataAdapter.

- Manually populate tables in the dataset by creating **DataRow** objects and adding them to the table's **Rows** collection. (You can only do this at run time; you cannot set the **Rows** collection at design time.) For more information, see Adding Data to a Table.
- Read an XML document or stream into the dataset. For more information, see the ReadXml method.
- Merge (copy) the contents of another dataset. This scenario can be useful if your application gets datasets from different sources (different XML Web services, for example), but needs to consolidate them into a single dataset. For more information, see the DataSet.Merge method.

Record Position and Navigation in Datasets

Because a dataset is a fully disconnected container for data, datasets (unlike ADO recordsets) do not need or support the concept of a current record. Instead, all records in the dataset are available.

Because there is no current record, there is no specific property that points to a current record and there are no methods or properties for moving from one record to another. (In contrast, ADO recordsets support an absolute record position and methods to move from one record to the next.) You can access individual tables in the dataset as objects; each table exposes a collection of rows. You can treat this like any collection, accessing rows via the collection's index or using collection-specific statements in your programming language.

> **Note** If you are binding controls in a Windows Forms to a dataset, you can use the form's binding architecture to simplify access to individual records. For more information, see Navigating Data in Windows Forms.

Related Tables and DataRelation Objects

If you have multiple tables in a dataset, the information in the tables might be related. A dataset has no inherent knowledge of such relationships; to work with data in related tables, therefore, you can create DataRelation objects that describe the relations between the tables in the dataset. **DataRelation** objects can be used to programmatically fetch related child records for a parent record, and a parent record from a child record.

For example, imagine customer and order data such as that in the Northwind database. The Customers table might contain records such as the following:

```
CustomerID   CompanyName               City
ALFKI        Alfreds Futterkiste       Berlin
ANTON        Antonio Moreno Taquerias  Mexico D.F.
AROUT        Around the Horn           London
```

The dataset might also contain another table with order information. The Orders table contains a customer ID as a foreign key column. Selecting only some of the columns in the Orders table, it might look like the following:

```
OrderId    CustomerID    OrderDate
10692      ALFKI         10/03/1997
10702      ALFKI         10/13/1997
10365      ANTON         11/27/1996
10507      ANTON         4/15/1997
```

Because each customer can have more than one order, there is a one-to-many relationship between customers and orders. For example, in the table above, the customer ALFKI has two orders.

You can use a **DataRelation** object to get related records from a child or parent table. For example, when working with the record describing the customer ANTON, you can get the collection of records describing the orders for that customer. Similarly, if you are working with the record describing order number 10507, you can use a **DataRelation** object to get the record describing the customer for that order (ANTON).

Constraints

As in most databases, datasets support constraints as a way to ensure the integrity of data. Constraints are rules that are applied when rows are inserted, updated, or deleted in a table. You can define two types of constraints:

- A unique constraint that checks that the new values in a column are unique in the table.
- A foreign-key constraint that defines rules for how related child records should be updated when a record in a master table is updated or deleted.

In a dataset, constraints are associated with individual tables (foreign-key constraints) or columns (a unique constraint, one that guarantees that column values are unique). Constraints are implemented as objects of type **UniqueConstraint** or **ForeignKeyConstraint**. They are then added to a table's Constraints collection. A unique constraint can alternatively be specified by simply setting a data column's Unique property to **true**.

The dataset itself supports a Boolean EnforceConstraints property that specifies whether constraints will be enforced or not. By default, this property is set to **true**. However, there are times when it is useful to temporarily turn constraints off. Most often, this is when you are changing a record in such a way that it will temporarily cause an invalid state. After completing the change (and thereby returning to a valid state), you can re-enable constraints.

In Visual Studio, you create constraints implicitly when you define a dataset. By adding a primary key to a dataset, you implicitly create a unique constraint for the primary-key column. You can specify a unique constraint for other columns by setting their Unique property to **true**.

You create foreign-key constraints by creating a **DataRelation** object in a dataset. In addition to allowing you to programmatically get information about related records, a **DataRelation** object allows you to define foreign-key constraint rules.

For more information about using **DataRelation** objects as foreign-key constraints, see Introduction to DataRelation Objects. For more information about creating constraints programmatically, see Adding Constraints to a Table.

Updating Datasets and Data Stores

When changes are made to records in the dataset, the changes have to be written back to the database. To write changes from the dataset to the database, you call the **Update** method of the data adapter that communicates between the dataset and its corresponding data source.

The **DataRow** class used to manipulate individual records includes the **RowState** property, whose values indicate whether and how the row has been changed since the data table was first loaded from the database. Possible values include **Deleted**, **Modified**, **New**, and **Unchanged**. The **Update** method examines the value of the **RowState** property to determine which records need to be written to the database and what specific database command (add, edit, delete) should be invoked.

For more information about updating data, see Dataset Updates in Visual Studio .NET.

See Also

Introduction to Data Access with ADO.NET | Introduction to Data Adapters | Visual Studio Tools for Creating Datasets | Creating Typed Datasets with the Component Designer | Creating XML Schemas and Datasets | Adding Existing Typed Datasets to a Form or Component | Adding Untyped Datasets to a Form or Component | Data Walkthroughs

Visual Studio Tools for Creating Datasets

Visual Studio provides the following ways to create datasets:

- The Component Designer, which allows you to drag data elements onto a form or component from the Toolbox or Server Explorer. Visual Studio can create the objects you need for your dataset, including a schema based on the structure of the original data source. When you are creating a form, the Component Designer appears as a tray at the bottom of the design surface. This method allows you to create either typed or untyped datasets.
- The XML Designer, which allows you to create a schema and generate a typed dataset based on it.

You can also create datasets dynamically in code; in that case, the dataset is untyped. (If you have created a typed dataset by generating it from a schema, you can change its structure dynamically in code as well.)

> **Note** When you deploy an application that includes Visual Studio data-access components, you must make sure that the user installing the application has version 2.7 or later of the Microsoft Data Access Components (MDAC). For more information, see Adding a Launch Condition for Microsoft Data Access Components.

Steps in Generating a Typed Dataset

When you generate a typed dataset, you actually perform a series of discrete steps:

1. Obtain an XML Schema (.xsd file). You might add a schema to your project in these ways:

   - Create a schema based on the data structure you are working with. If you use the Component Designer to add a data adapter to your form or component, the adapter can read the data-source structure and translate it into an .xsd file.
   - Create a schema manually using the XML Designer.
   - Create a reference to an XML Web service or other component that returns a dataset. As part of the reference, Visual Studio creates a local copy of the schema.

     For more information about dataset schemas, see XML Schemas and Data.

2. Generate a dataset class. In this step, Visual Studio derives a class from the base **DataSet** class in the **System.Data** namespace. It then uses the information in the schema to create strongly typed members representing the data tables and data columns for the dataset.

   Under most circumstances, Visual Studio generates this dataset class automatically when the schema is added to the project.

3. Create an instance of the derived dataset class from Step 2 to use on a form or component.

When you work with a dataset in your application, you are actually working with an instance of the dataset class (like that one created in Step 3).

Using the Component Designer

In general, you use the Component Designer when you want Visual Studio to generate a schema and a typed dataset class for you. In the Component Designer, you add one or more data adapters to your form or component. The adapters push data shape information into the dataset (via their FillSchema methods). The dataset can then write the resulting structure out as an XML Schema (.xsd file) in the project. At that point, you can have Visual Studio generate an instance of the typed dataset class to be added to your form or component.

The Component Designer is therefore an indirect way to create a dataset — you do not explicitly define the tables and columns in the dataset. Instead, you implicitly define them by configuring the data adapter. You can do this by establishing the SQL queries or stored procedures that the adapter uses to exchange data with the data source. You can also use the adapter's TableMappings property to specify which columns are included in the dataset and what their names are.

After configuring the data adapter, you generate the dataset from it. If you change the data adapter after generating the dataset — for example, by changing table mappings — you must then regenerate the dataset to reflect these changes.

The disadvantages of using the Component Designer are:

- You initially have no direct control over the schema; you can control the dataset definition only by configuring the data adapter from which Visual Studio generates the schema. After generating the schema, however, you can edit it using the XML Designer.

- You must manually regenerate the schema (and corresponding dataset class file) after changing a data adapter. For example, if you change the SQL statement used to fill a dataset, you must change the dataset schema as well.
- There are some dataset functions you cannot perform using the Component Designer alone, such as defining **DataRelation** objects. In that case, you must use the XML Designer.

Using the XML Designer

You use the XML Designer when you want to control more precisely how the schema is defined or if you cannot generate the schema from a data source. You can use the XML Designer to create a schema in two ways:

- Drag database elements (for example, column names) from Server Explorer onto the designer. This creates the schema that is initially generated from the database structure and which you can then modify as needed.
- Create a schema file yourself. This is useful when you are designing a schema without reference to any external data source.

In both cases, you can then generate an instance of a typed **DataSet** class based on the schema that you create and use it on a form or component.

The XML Designer allows you to add, remove, or modify elements. The XML Designer also allows you to perform functions (such as adding **DataRelation** objects) that cannot be accomplished using the Component Designer.

When you use the XML Designer, you are editing an .xsd file. Very frequently, your project will also contain a typed dataset class file based on the .xsd file. If you save the changes in the .xsd file, the corresponding class file is regenerated to reflect the updated schema. Visual Studio can keep the .xsd and corresponding class file synchronized by continuously regenerating the class file as you make changes in the .xsd file. You can turn off this functionality, which you might do if you are editing a schema file for which you do not need a class file.

The disadvantages of working in the XML Designer are:

- The XML Designer offers a slightly lower level of integration with other data tools in Visual Studio.
- The .xsd file is not validated against any external data source. For example, if you use the XML Designer to modify a dataset schema, you must also modify any data adapters that interact with the dataset.
- Some familiarity with XML Schemas and XML is assumed.

See Also

Introduction to Datasets | Introduction to Data Adapters | Introduction to DataRelation Objects | Introduction to XML Schemas | Creating Typed Datasets with the Component Designer | Creating XML Schemas and Datasets | Adding Untyped Datasets to a Form or Component | Data Walkthroughs

Creating Typed Datasets with the Component Designer

When you are using a dataset with a form or component, you can use the Component Designer to define the dataset schema and class file. You can then add an instance of the dataset to the form or component you are working with.

When you are editing a form, the Component Designer appears as a tray at the bottom of the designer window. The tray opens automatically as soon as you add a component or nonvisual control to the form. Instances of dataset classes are displayed there, where you can select them in order to set properties or write event handlers for them.

To create a dataset in the Component Designer

1. Add one or more data adapters to the form or component that specify what data you want to put in the dataset. For more information, see Creating Data Adapters.
2. In the Component Designer, select the adapter or adapters that will be used to transfer data between the data source and the dataset.

   Typically, each data adapter accesses data in a single table. Therefore, to create a dataset containing multiple data tables, you should select all the adapters for the tables you want to work with.

   > **Note** Adding multiple tables to the dataset does not automatically create a relationship between the tables; you must do that manually after creating the dataset. For more information, see Creating DataRelation Objects with the XML Designer.

3. From the **Data** menu, choose **Generate Dataset**.

   The **Generate DataSet** dialog box appears.

4. Select **New** and then specify a name for the new dataset. If you want to add the dataset to your form or component, click **Add an instance of this DataSet to the designer**.

   Visual Studio uses the information in the data adapters to generate an XML Schema file (.xsd), which is added to the project using the name you specified. Visual Studio then generates a new dataset class file based on the schema.

   > **Note** The new dataset class file is visible in **Solution Explorer** only if you select **Show All Files**.

   If you indicated that you wanted an instance of the dataset, the instance is added to the form or component using the name you specified with a numeric suffix. For example, if you named the dataset **dsAuthors**, the instance is named **dsAuthors1**.

   > **Note** If you accept the default dataset name of DataSet1, the instance in your form or component will be named DataSet11 because of the numeric suffix that is automatically added.

5. If you added a dataset to the form or component, add code to populate it. Most frequently, you do this by calling the **Fill** method of the data adapters. For more information, see Populating a DataSet from a DataAdapter.
6. If you want to update the dataset and database, add code to the form or component to do so. For more information about updating, see Dataset Updates in Visual Studio .NET.

See Also

Creating XML Schemas and Datasets | Introduction to DataRelation Objects | Adding Tables to an Existing Dataset | Creating Data Adapters | Adding Untyped Datasets to a Form or Component | Adding Existing Typed Datasets to a Form or Component | Walkthrough: Creating a Distributed Application | Data Walkthroughs

Creating Dataset Columns Using Expressions

As in some databases, a data table in a dataset can contain columns in which the value is calculated rather than read from a store. For example, in an orders detail record, it might make sense to define an ExtendedPrice column by multiplying the quantity and the price, rather than actually storing the total in the record. Calculated columns can also add or count values in child records and filter other records. You can define calculated columns using expressions.

The syntax of the expression consists of standard arithmetic, Boolean, and string operators and literal values. You can reference a data value using its column name (as you would in an SQL statement) and include aggregate functions such as Count.

For example, the expression for an ExtendedPrice column based on the UnitPrice and Quantity columns of the current record might simply be:

```
UnitPrice * Quantity
```

To refer to columns in child tables, you can use the reserved word "Child." followed by the name of the column. For example, the following expression returns the average value of the Price column in all related child records:

```
Avg(Child.Price)
```

Column Expressions in Typed Datasets

You can create the column and expression for typed datasets in the XML Designer. For more information about creating columns using the XML Designer, see Creating Tables in XML Schemas.

To create column expressions in typed datasets

1. If the schema is not already open in the XML Designer, double-click the XML Schema (.xsd) file in Solution Explorer.
2. In the grid representing the table you want to work with, add a new column by performing the following steps:
   a. In the first blank line of the table grid, select **Element** from the drop-down list.
   b. In the next box in the same line, enter a name for the column, using a valid column identifier.
   c. In the next box, select an appropriate data type for the results of the calculation.
3. Select the new column in the table grid, and then in the Properties window, enter an expression in the **Expression** property. For more information about column expression syntax, see the DataColumn.Expression property.

Column Expressions in Untyped Datasets

If you are working with an untyped dataset, you cannot use the XML Designer because there is no schema (unless you generate one from the dataset using WriteXmlSchema). However, you can create the column and expression in the form or Component Designer.

To create column expressions in untyped datasets

1. Select the untyped dataset in the form or component and open the Properties window.
2. Click the button in the **Tables** property to open the **DataTable Collection Editor** dialog box.
3. In the collection editor, click the button in the **Columns** property to open the **DataColumn Collection Editor** dialog box.
4. Click **Add** to create a new column.
5. Select the new column, and in the **Properties** grid, set the **ColumnName** property to provide a name by which you can refer to the column in code.

   Note You can also set the (Name) property to give the column a different name at design time.

6. Select an appropriate data type for the results of the calculation.
7. Enter an expression in the **Expression** property. For more information about column-expression syntax, see the DataColumn.Expression property.

See Also

Creating Tables in XML Schemas | DataColumn.Expression Property | Creating Expression Columns |
Code: Creating an Expression Column (Visual Basic):

Adding Tables to an Existing Dataset

Occasionally you might need to add a data table to a dataset that has already been generated. One way is to delete the dataset and regenerate it manually to include the new table. However, in that case you might lose additional dataset information (including constraints and **DataRelation** objects) that were added to the dataset after it was generated.

You can add tables to an existing dataset two ways:

- Use the Component Designer in a form or component. This is an easy solution that allows you to drag elements onto the form or component and then regenerate a modified dataset.
- Use the XML Designer to modify the dataset schema. This strategy is useful if you are comfortable using the XML Designer and do not want or need to open a form or component. Adding a table using the XML Designer is similar to creating the schema manually. For more information, see Creating Tables in XML Schemas.

To add a table to an existing dataset using the Component Designer

1. In the form or component you are working with, add a data adapter that references the table you want to add.
2. Select the data adapter, and then from the **Data** menu, choose **Generate Dataset**.

   The **Generate Dataset** dialog box is displayed.
3. Under **Choose a Dataset**, choose **Existing**, and then select the name of the dataset to add the table to.
4. Under **Choose which table(s) to add to the dataset**, select the names of all the tables that you want to add to the dataset. The list includes tables from all available data adapters.
5. If you want to create an instance of the modified dataset on your form or component, select **Add an instance of this dataset to the designer**.
6. Click **OK** to generate an updated dataset.

   Information about the added table is added to the dataset schema and to the derived dataset class file. Existing information (including tables, constraints, and relationships) is left intact.

See Also

Creating Typed Datasets with the Component Designer | Relationships in ADO.NET Datasets | Creating Tables in XML Schemas | Adding a DataTable to a DataSet | Code: Adding Two Related DataTables to a DataSet (Visual Basic)

Adding Existing Typed Datasets to a Form or Component

You will frequently need to add a typed dataset to a form or component so that you can manipulate data, bind controls to the dataset, and so on. When you add a dataset, you are actually creating an instance of the typed dataset class on your form or component. The typed dataset class must already exist and can come from several sources:

- You created it by generating it in another form or component in your project. For more information, see Creating Typed Datasets with the Component Designer.
- You created it manually in the XML Designer. For more information, see Creating XML Schemas and Datasets.
- You created a reference to an XML Web service or other component that returns a dataset. For more information, see Accessing an XML Web Service in Managed Code.

You can add an untyped dataset to a form or component as well, but you create it in the form or component rather than adding an instance of a typed dataset class. For more information, see Adding Untyped Datasets to a Form or Component.

To add an existing dataset to a form or component

1. Open the form or component you want to work with. If necessary, switch to **Design** view of the designer.
2. From the **Data** tab of the Toolbox, drag a **DataSet** object onto the designer.

   The **Choose a Dataset** dialog box appears.

3. Select **Typed Dataset**, and then from the drop-down list, select the dataset you want to use, and then click **OK**.

   The drop-down list is populated with a list of all typed dataset classes in your project.

   > **Tip** If you have added an XML Web service or component but cannot see the dataset's schema in Solution Explorer, open up the node for the reference. If you still do not see the schema, double-check that the XML Web service or component is returning a typed dataset.

   An instance of the typed dataset class is added to the form or component.

4. Add code to the form or component to fill the dataset.

   If the dataset represents data from a database, you generally call the **Fill** method of a data adapter to execute an SQL statement or stored procedure that fetches the data for the dataset. For more information, see Populating a DataSet from a DataAdapter. The following example shows how you might fill the table Customers in the dataset dsCustomersOrders (whose instance is called dsCustomersOrders1):

   ```
   ' Visual Basic
   oleDbDataAdapter1.Fill( dsCustomersOrders1, "Customers" )

   // C#
   oleDbDataAdapter1.Fill( dsCustomersOrders1, "Customers");
   ```

   If the dataset represents information returned by an XML Web service or a component, you need to load the dataset by invoking the method that returns data. You then typically need to merge the data returned by the XML Web service or component into the dataset on your form. The following example shows how to create an instance of an XML Web service, invoke its method called GetCustomersDataset, and load it into a local dataset called dsCustomers1:

   ```
   ' Visual Basic
   Dim ws1 As New MyServer.CService1()
   dsCustomers1.Merge(ws1.GetCustomersDataset())

   // C#
   MyServer.CService1 ws1 = new MyServer.CService1();
   dsCustomers1.Merge(ws1.GetCustomersDataset());
   ```

   You can also fill the dataset by creating data programmatically. For more information, see Adding Data to a Table.

5. If your application requires that you write information from the dataset to a database, add code to the form or component

to do this.

If the dataset is filled using a data adapter, you can call the data adapter's **Update** method to write changed data back the database. For more information, see Updating the Database with a DataAdapter and the DataSet.

See Also

Creating Typed Datasets with the Component Designer | Creating XML Schemas and Datasets |
Adding Untyped Datasets to a Form or Component | Populating a DataSet from a DataAdapter | Data Walkthroughs

Adding Untyped Datasets to a Form or Component

An untyped dataset is an instance of the DataSet class that you create in a form or component. Unlike a typed dataset, an untyped dataset is not an instance of a class generated using an XML Schema (.xsd) file. As a consequence, an untyped dataset has no inherent structure. You create the tables, columns, constraints, and **DataRelation** objects yourself at design time in the Properties window, at run time in code, or by allowing a data adapter's mappings to do so when you use the data adapter to fill the dataset.

> **Note** Although an untyped dataset is not based on a class that incorporates a schema, after building the untyped dataset, you can export its structure to a schema file. For more information, see the DataSet.WriteXmlSchema method.

Creating the Dataset

The instructions below describe how to create an untyped dataset using the Component Designer while editing a form or component. For more information about how to create an untyped dataset at run time in code, see Creating a DataSet and Adding a DataTable to a DataSet.

> **Note** When you deploy an application that includes Visual Studio data-access components, you must make sure that the user installing the application has version 2.7 or later of the Microsoft Data Access Components (MDAC). For more information, see Adding a Launch Condition for Microsoft Data Access Components.

To create an untyped dataset at design time

1. From the **Data** tab of the Toolbox, drag a **DataSet** element onto the form or component you are working with.

   The **Choose a Dataset** dialog box is displayed.

2. Select **Untyped Dataset** and click **OK**.

   A blank dataset is added to the form or component.

3. If you want to give the dataset a new name, select it, open the Properties window, and edit the following two properties.

   - **(Name)** This property sets the object name by which you refer to the dataset in code.
   - **DataSetName** If you export the dataset to an XML document, this property sets the name of the dataset in the document (that is, the name of the topmost element). Setting this property can help avoid problems with name duplication or case sensitivity.

Adding Tables and Columns

You can use the Properties window to create the structure of the untyped dataset. The dataset includes a Tables property that is a collection of the DataTable objects defined for the dataset. Each table in turn supports a Columns property that is a collection of DataColumn objects. To create tables and columns, therefore, you use the Collection Editor feature of the Properties window.

> **Note** You cannot create records in an untyped dataset at design time; you can populate a dataset only at run time. The **Rows** collection of a dataset is therefore not available in the Properties window.

To add tables and columns to the dataset

1. In the form or component, select the dataset and open the Properties window.
2. In the **Tables** property, click the button to open the **DataTable Collection Editor** dialog box.
3. Click **Add** to create a new table. If you want to rename the table, select it and in the right pane of the properties grid, change the value of the **(Name)** property.

   > **Note** The names of all tables in all datasets in the form or component share the same namespace. Therefore, you cannot use a table name in one dataset if that name is in use in another dataset.

4. To add columns to the table, do the following:
   a. Make sure the table you are working with is selected in the **Members** list, and then in the **Properties** grid on the right, click the button in the **Columns** property to display the **DataColumn Collection Editor** dialog box.
   b. Click **Add** to create a new column, and then set the new column's properties in the grid on the right. Some of the properties allow you to set column constraints, such as a unique constraint or a non-null constraint. Typical properties to set are listed in the following table.

Property	Description

AllowDBNull	A Boolean value indicating whether the column can be left blank in a given record.
ColumnName	The name of the column in the table's **Tables** collection, which is how the column can be referenced in code.
DataType	A .NET Framework data type to be enforced for the column. If you import or export an XML document to or from this dataset, data types are automatically converted to schema types (for example, `System.String` is converted to `xs:string` and vice versa). Select a data type from the drop-down list.
ReadOnly	A Boolean value indicating whether a column can be changed.
Unique	A Boolean value indicating a unique constraint on the column. You always set this to **true** for a primary key column, but it can be set for any column.
(Name)	An instance name for the column used only at design time.

   c. Repeat Step b for each column you want to add.
   d. When you are finished adding columns for the current table, click **OK** to close the **DataColumn Collection Editor** dialog box.
5. To set a primary key for the table, use the drop-down list in the **PrimaryKey** property.
   a. In the drop-down list, select the check box next to each part of the primary key.
   b. After selecting one or more columns for the primary key, you can expand the **PrimaryKey** property and set properties for each primary key column. The **Unique** property is automatically set for all the columns that are part of the primary key.
6. Repeat Steps 3 through 5 for each table and set of columns that you want to add to the dataset.

Adding Constraints

You can define two types of constraints in a dataset:

- A unique constraint that checks that the new values in a column are unique in the table.
- A foreign-key constraint that defines rules for how related child records should be updated when a record in a master table is updated or deleted.

To add constraints to an untyped dataset at design time

1. In the form or component, select the dataset and open the Properties window.
2. Click the button in the **Tables** property to open the **Tables Collection Editor**.
3. Select the table to add a constraint to, and then click the button in the **Constraints** property to open the **Constraints Collection Editor**.
4. Click **Add** and then select **Unique Constraint** or **Foreign Key Constraint** from the drop-down list.

   The **Unique Constraint** or **Foreign Key Constraint** dialog box id displayed. For help on filling in these dialog boxes, press F1.

Adding Relations

You can also create relationships (**DataRelation** objects) between tables in the dataset at design time. For more information about **DataRelation** objects, see Introduction to DataRelation Objects.

To add a DataRelation object to an untyped dataset at design time

1. In the form or component, select the dataset and open the Properties window.
2. In the **Relations** property, click the button to open the **DataRelation Collection Editor** dialog box.
3. Click **Add** to create a new data relation.

   The **Data Relation Editor** dialog box opens with default values for the new relationship.

4. In the dialog box, set values according to the following table.

Value	Description
Name	The name by which you can refer to this **DataRelation** object in code. A typical name includes the names of both tables being related (for example, `CustomersOrders`).

Parent table	The table containing the parent records. Select a table from the drop-down list.
Child table	The table containing child records. Select a table from the drop-down list.
Parent column	The names of the columns that make up the primary keys in the parent table. You can select column names from the drop-down list or type in a column name (for example, if you know the name of a column that will be defined at run time).
Child column	The names of the columns that make up the foreign keys in the child table.
Enforce constraints	A Boolean value indicating whether the **DataRelation** object will be used to cascade changes from the parent table to the child table and vice versa. **Note** If you choose to enforce constraints, a foreign key constraint is added to the parent table.
Update rule Delete rule Accept/reject rule	The nature of the cascaded change that should be made in the child table when changes occur in the parent table. For more information about the settings for these values, see Introduction to DataRelation Objects.

5. Click **OK** to save the changes to this relationship.
6. Repeat Steps 3 through 5 for each relationship you want to create.

See Also

Introduction to DataRelation Objects | Adding Constraints to a Table | DataSet.WriteXmlSchema Method

Handling Null Values in a Strongly Typed Dataset

In a typed dataset, you can control the representation of columns that contain **null** values. This behavior is only available when retrieving column values through their typed accessors. If you retrieve the value by means of the table's row index or column index, you will not be able to set the return value.

To set the value to return when a DataColumn's value is null

1. Open the dataset in the XML Designer.
2. Select an Element whose type is NOT an unnamed (or anonymous) type.
3. In the Properties window, select the **NullValue** property.
4. Do one of the following:

   - From the drop-down list select the appropriate value:

Value	Description
(Empty)	To have null values return as String.Empty **.**
(Null)	To have null values return as **null.**
(Throw)	To have null values throw an Exception.

   - - or -
   - Set the property to any custom value.

See Also

String.Empty | ADO.NET Datasets | DataSet Class | DataColumn Class

Walkthrough: Mapping Data Source Tables to Dataset Tables

The structure of tables in a dataset does not always match the structure of tables in a data source. There might be several reasons for this:

- The dataset was created from an existing schema that uses different names than those in the data source.
- You want to change data element names in the dataset for convenience, readability, translation to or from a foreign language, or other reason.
- You want to control the names of typed data members when you generate a dataset from the adapter.

To enable you to match up tables and columns in the data source with those in the dataset, you can set the **TableMappings** property of a data adapter. This property contains information that establishes a correspondence between data source table columns and dataset table columns. When the adapter reads data from the data source, **TableMappings** property is used to determine where in the dataset to write the data.

This walkthrough gives you brief overview of how to set up table mappings in a data adapter. The scenario for the walkthrough assumes that you are working with a schema provided to you by someone else. You will map the columns from a table in your database (in this case, the Customers table of the Northwind database) to the columns defined in the schema.

In order to complete this walkthrough, you will need:

- Access to a server with the Northwind SQL Server sample database.

The walkthrough is split into a number of smaller pieces:

- Creating a Windows Form. The form is used only to illustrate how to create the mapping; you could use any form or component.
- Creating a dataset with a pre-defined schema. The walkthrough contains a simple but complete schema that you can copy and paste into Visual Studio to simulate a schema you might have gotten from someone else.
- Creating and configuring the data adapter in which you will map the data source to the dataset.
- Adding the **DataGrid** control to show the effect of the mapping.

Creating the Project and Form

This walkthrough uses a Windows Form as the container on which you will create the data adapter and dataset. It is not necessary that you use a Windows Form — you could also use a Web Forms page or a component — but a Windows Form is easy to work with and allows you later to use a **DataGrid** control to see the effect of the mappings.

To create the project and form

1. From the **File** menu, point to **New**, and then choose **Project**.
2. In the **Project Types** pane, choose **Visual Basic Projects** or **Visual C# Projects**, and then in the **Templates** pane, choose **Windows Application**.
3. Assign a name to the project that will be unique and will conform to the naming conventions you use. For example, you might name this project **Walkthrough_Mappings**. After you have assigned a name, click **OK**.

   Visual Studio creates a new project and displays a new form in the Windows Form Designer.

Creating the Schema and Dataset

For the walkthrough, imagine that someone has provided you with a schema that defines a Customers table. The schema defines a Customers table that is slightly different from the one in your Northwind database.

In this section, you will add a schema in your project. The actual schema layout (the XML that defines the schema) is provided below. By default, the XML Designer generates a class file based on the schema and derived from the DataSet class. The resulting class file defines the typed dataset class that you use in your application.

For more information about dataset and how they are defined, see Introduction to Datasets.

To add the schema

1. In Solution Explorer, right-click the name of your project, choose **Add**, and then choose **Add New Item**.
2. In the **Templates** pane, choose **DataSet**, name the new item **dsCustomers**, and then click **Open**.

   The XML Designer opens with a new, blank dataset schema.

3. Click the **XML** tab at the bottom.
4. Copy the following schema text and then paste it into the XML view of the designer, completely overwriting the schema already in the file.

   Tip If the < and > symbols are replaced with
   <
   and
   >
   when you paste, remove the pasted text, and then re-paste the schema by right clicking the design surface and selecting **Paste as HTML** from the shortcut menu.

```
<?xml version="1.0" encoding="utf-8" ?>
<xs:schema id="dsCustomers" targetNamespace="http://www.tempuri.org/dsCustomers.xsd" xmln
s="http://www.tempuri.org/dsCustomers.xsd" xmlns:xs="http://www.w3.org/2001/XMLSchema" xm
lns:msdata="urn:schemas-microsoft-com:xml-msdata" elementFormDefault="qualified">
   <xs:element name="dsCustomers" msdata:IsDataSet="true">
      <xs:complexType>
         <xs:choice maxOccurs="unbounded">
            <xs:element name="Customers">
               <xs:complexType>
                  <xs:sequence>
                     <xs:element name="CustID" type="xs:string" />
                     <xs:element name="CompName" type="xs:string" />
                     <xs:element name="ContName" type="xs:string" minOccurs="0" />
                     <xs:element name="ContTitle" type="xs:string" minOccurs="0" />
                  </xs:sequence>
               </xs:complexType>
            </xs:element>
         </xs:choice>
      </xs:complexType>
         <xs:key name="Constraint1" msdata:PrimaryKey="true">
            <xs:selector xpath=".//Customers" />
            <xs:field xpath="CustID" />
         </xs:key>
      </xs:element>
   </xs:schema>
```

5. Save the schema and close it.
6. Check that you have a dataset class file based on the schema. In Solution Explorer, click the **Show All Files** button, and then expand the node for the new schema. You should see a class file (with extension of .vb or .cs) underneath the schema.

Now that you have a schema and related dataset class file, you can add an instance of the dataset to your form.

To add the dataset to the form

1. Click the **Form1** tab at the top to return to the Windows Form designer and your form.
2. From the **Data** tab of the Toolbox, drag the **DataSet** element onto the form.

   The **Add Dataset** dialog box is displayed.

3. Select **Typed Dataset** and then from the **Name** list, choose the dataset named **dsCustomers**. The dataset name is prefixed with your project name.

   An instance of the dataset is added to the form under the name dsCustomers1.

Creating a Data Adapter and Table Mapping

With the dataset defined, you can configure the other half of your data access story: the data adapter that will read data from the database and fill it into the dataset. The data adapter will contain the table mapping that matches the database columns to those in the dataset.

Adding the Data Adapter

The first step is to create the data adapter, which you can do using a wizard.

To add a data adapter

1. From the **Data** tab of the Toolbox, drag the **OleDbDataAdapter** element onto the form.

   The **Data Adapter Configuration Wizard** dialog box is launched.

2. In the wizard, do the following:
   a. In the second pane, create or choose a connection pointing to the SQL Server Northwind database.

      Security Note Storing connection-string details (such as the server name, user name, and password) can have implications for the security of your application. Using Windows Integrated Security is a more secure way to control access to a database. For more information, see Database Security.

   b. In the third pane, specify that you want to use an SQL statement to access the database.
   c. In the fourth pane, create the following SQL statement:

      ```
      SELECT CustomerID, CompanyName, ContactName, ContactTitle
      FROM Customers
      ```

      For assistance building the SQL statement, click **Query Builder** to launch the **Query Builder**.

      Note In this walkthrough, you will populate the dataset with all the rows from the Customers table. In production applications, you typically optimize data access by creating a query that returns only the columns and rows you need. For an example, see Walkthrough: Displaying Data in a Windows Form Using a Parameterized Query.

   When the wizard is complete, you will have a connection (**OleDbConnection1**) containing information about how to access your database. You will also have a data adapter (**OleDbDataAdapter1**) that contains a query defining what table and columns in the database you want to access.

Creating the Table Mapping

Now that you have a data adapter on the form, you can configure its table mappings.

To create the mappings

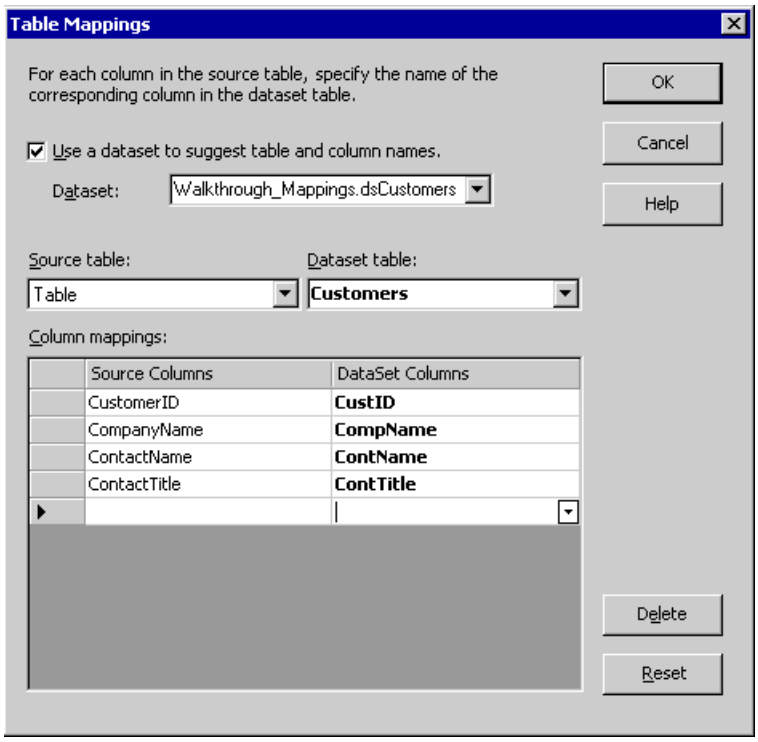
1. Select the data adapter at the bottom of the form.
2. In the Properties window, click the button in the **TableMappings** property.

   The Table Mapping dialog box is displayed.

3. Check **Use a dataset to suggest table and column names** and select **Walkthrough_Mappings.dsCustomers** from the drop-down list. (If you named your project something else, the prefix will be different in this list.)
4. Select **Customers** from the **Dataset table** list.
5. Change the names of columns in the **Dataset Columns** list individually by using the drop-down list that is available for each column. Create the following mappings:

Source Columns	Dataset Columns
CustomerID	CustID
CompanyName	CompName
ContactName	ContName
ContactTitle	ContTitle

   When you have finished, the **Mappings** dialog box will look like this:



6. Click OK to close the **Table Mappings** dialog box.

Using Controls to Illustrate Mapped Names

In the final step of the walkthrough, you add a **DataGrid** control that illustrates how the mappings are used in the dataset. When you bind the grid to the dataset table, you will see that the dataset table contains the mapped names.

To add and bind a DataGrid control

1. Add a **DataGrid** from the **Windows Forms** tab of the **Toolbox**.
2. Select the **DataGrid** and open the Properties window.
3. Set the following data-binding properties:

   - Set the **DataSource** property to **dsCustomers1**.
   - Set the **DataMember** property to **Customers**.

   The column headings will match those in the schema.

To be able to run the form and see the mapped data in use, you will need to fill the dataset.

To fill the dataset

- Double-click the form (not the control) to create an event handler for the form's **Load** event and add the following line to it:

```
'Visual Basic
OleDbDataAdapter1.Fill(dsCustomers1)

// C#
oleDbDataAdapter1.Fill(dsCustomers1);
```

Finally, you can test the form to see the mapping in action.

To test the form

1. Press **F5** to run the form.
2. When the form is displayed, confirm that the data from the Northwind Customers table is displayed in the grid using the mapped column names.

Next Steps

This walkthrough has provided a brief introduction to table mapping in data adapters. Mapping can be used not just to change column names, but in more sophisticated ways as well, including:

- Map columns from one table to an entirely different column in the dataset.
- Use mapping to limit the number of columns created in a dataset. This is useful if your data adapter reads all the columns in a database table but you only need a few of them in a dataset table.
- Map an expression column in a dataset to a database column. This allows you to calculate values in the dataset and write them to the database.
- Raise errors under various circumstances, including mismatches.

For more information, see Table Mapping in Data Adapters.

See Also

Introduction to Data Adapters | Introduction to Datasets | Introduction to XML Schemas | Creating XML Schemas and Datasets | Creating an XML Schema with the XML Designer

Filtering and Sorting in Datasets

After filling a dataset, you will often find it useful to work with different subsets of records in a table, or to view the data in a different order. You can do so by filtering and sorting the data in the dataset. To simplify this process, you can create data views, which provide objects that incorporate filter and sort criteria, and which you can use as the source for data binding.

The topics in this section describe options for filtering and sorting and instructions for performing these actions on data in a dataset.

In This Section

Introduction to Filtering and Sorting in Datasets
Provides an overview of how you can add filtering (selecting) and sorting in a dataset, both by calling data table methods and by using data view.
Adding Data Views to a Form or Component
Describes how to add and configure data views at design time.
Filtering and Sorting Data Using Data Views
Describes how to set data view properties to sort and filter data from a data table.
Filtering And Sorting Directly in Data Tables
Describes how to create lists of sorted or filtered records from a data table, without using a data view.
Working with Records in Data Views
Describes how to edit, insert, and delete records via a data view.
Creating and Working with Data View Managers
Describes how to create a data view manager, which provides centralized access to sorted and filtered views of all the tables in a dataset.

Related Topics

DataView Class
Provides reference information about the **DataView** class and links to topics about each member of the class.
Populating a DataSet from a DataAdapter
Describes how to call data-adapter methods to execute SQL commands that provide the data for a dataset.
Adding Data to a Table
Describes how to programmatically create rows in dataset tables.
Data Binding and Windows Forms
Lists topics that provide information on how to bind controls in Windows Forms to data, including to data views.
Data Access in Web Forms Pages
Lists topics that provide information on how to bind controls in Web Forms pages to data, including to data views.
Walkthrough: Creating a Master-Detail Windows Form
Shows how to use a data view filter to get child record from a designated parent record.

Introduction to Filtering and Sorting in Datasets

After a dataset has been populated, your application might need to view its data in various ways. For example, you might want to view the records in a specific order or see only a subset of the records (that is, filter them). Because the dataset is disconnected from the data source, it is often impractical and resource-intensive to re-execute SQL commands to perform these actions.

Instead, you can use built-in dataset features to filter and sort. You have two options:

- Data tables support a Select method that you can call to filter and sort. The method does not change the contents or order of records in a table; instead, it presents you with a list of records (an array) representing the criteria you specify.
- You can use a data view (DataView object). A data view is an object that acts as a layer on top of the data table, providing a filtered and sorted view of the table's contents. (You can also use a data view manager, which acts like a collection of data views.) A data view is similar to a view in a database, in that it is not a copy of the data. Instead, it is simply a different way of seeing the data in a table.

Both methods provide the same filtering and sorting capabilities. The primary difference is that you can call a data table's **Select** method only in code at run time. Using a data view, on the other hand, provides these advantages:

- You can create and configure the data view at design time, with the option to set its properties at run time as well.
- You can use it in data binding; that is, you can bind controls to it in a designer.
- You can create multiple data views to see the data in a table in different ways. For example, one data view might display data in an Orders table in date order, and another data view might show it in customer order.

> **Note** In some instances, you might not be using a dataset at all. Using a dataset (and filtering and sorting data in it) is an appropriate choice when you want to work with the same set of records while disconnected from the data source. For more information, see Recommendations for Data Access Strategies.

Default Data Views

If you choose to use data views, you can add one to a form or component in much the same way you do other data elements, namely, by dragging it onto a designer and setting its properties. However, even if you do not explicitly add a data view to your form or component, a data view — the default data view — is already available for each table in the dataset. You can access this default data view using a table's DefaultView property, which returns a **DataView** object. You can set the default data view's properties at run time.

For the most part, you can use the default data view as you would any data view. But adding custom data views to your form or component has these advantages:

- You can use multiple data views on a single set of data.
- You can set data view properties at design time. (The default data view is available only at run time.)
- You can assign names to your data views.

Filtering in a Data Table

Using the table's **Select** method or the RowFilter property of a data view, you can filter records in a data table to make available only records you want to work with. This is useful when you want to work with different subsets of the records in a dataset table. To specify filter criteria, you use the same expression syntax used to create column expressions. The filter expression is evaluated as a Boolean expression; if the expression returns **true**, the record is included. A filter expression might look like the following:

```
Price > 10.00
```

In the expression you can reference column values by name (as in the preceding example), property values, and literal values.

Filtering on Row State and Version

A dataset can maintain different versions of records in tables. When records are first filled in, the dataset contains the original version of the record. If you change a record, the dataset maintains a different version — the current version — that reflects the changes. A property on the record indicates whether the record is unmodified, updated, new, or deleted. For more information about row versions and row state, see Introduction to Dataset Updates.

A common use for filters is to specify only the current versions of records in the data table. If records have been changed, there are two versions of a record: The current version reflecting the changes, and the original version representing the record before

any changes were made. Records are also flagged according to their status: new, unchanged, deleted, or modified. A deleted record, for example, still exists in the data table, but its row-state flag has been set to deleted.

If you get all the records in a data table by simply looping through the table's **Rows** collection, you will include records that have been modified, new records, and deleted records. A common filter, therefore, is one that returns only specific versions of a record or records in a certain state.

If you are calling the **Select** method and passing it a filter expression, you can add a parameter indicating what row state or version you want to filter on. If you are using a data view, you can set the RowStateFilter to indicate what version or state of a row you want.

Sorting

Sorting is similar to filtering, in that you specify a sort expression. A typical sort expression is simply the name of the column to sort by. For example, to sort by the OrderDate column, you specify the sort expression `OrderDate`. However, you can sort by the value of any expression, including calculated values. If you call a table's **Select** method, you pass the sort expression as a parameter. If you are using data views, you specify the sort expression as the value of the view's Sort property.

Data View Manager

You can define individual data views that sort or filter the data in each dataset table. If the dataset contains multiple tables, an alternative is to create a data view manager (a DataViewManager object). The data view manager works something like a dataset-wide data view: It provides a single object that manages a collection of data views, each of which is associated with a specific table in a dataset.

A data view manager is particularly useful when you are working with related tables and want to sort or filter child records from a master table. As an example, imagine a dataset containing a Customers table, an Orders table, and a **DataRelation** object that links the two tables. You can use individual data views to sort the Customers table by CompanyName and the Orders table by OrderDate. However, when you use individual data views to sort this way — for example, when you use the relation object's GetChildRecords method to get Orders records — the records won't be sorted. In contrast, if you use a data view manager, they will be.

To use a data view manager, you must create and configure it in code; there is no design-time object that you can add to a form or component. This means that if you want to bind controls to the data view manager, you must also do so in code, as there is no design-time component to bind them to. For more information, see Creating and Working with Data View Managers.

See Also

Adding Data Views to a Form or Component | Filtering and Sorting Data Using Data Views |
Filtering And Sorting Directly in Data Tables | Working with Records in Data Views

Adding Data Views to a Form or Component

A data view is an object that acts as a layer on top of the data table, providing a filtered and sorted view of the table's contents. (You can also use a data view manager, which acts like a collection of data views.) For more information, see Introduction to Filtering and Sorting in Datasets. You can use design-time tools to add a data view to a form or component.

A common use for data views is to use them as data sources when data binding controls in a form. For more information, see Data Binding and Windows Forms and Data Access in Web Forms Pages.

To add a data view

1. Create a dataset. For more information see Creating Typed Datasets with the Component Designer.
2. Add code to populate the dataset. For more information, see Populating a DataSet from a DataAdapter.
3. From the **Data** tab of the Toolbox, drag a **DataView** item onto the form or component.

   A new data view with the default name DataView1 is added to form or component.

4. If you want to configure the data view at design time (rather than in code at run time), select the data view and use the Properties window to configure the data view. The following table lists commonly used properties of the data view.

   Note You can set all of these properties at run time except the name.

DataView Property	Description
(Name)	The name by which you can refer to the data view in code.
Table	A reference to the table (**DataTable** object) that this is a data view for. You can select a table name from the drop-down list. You must set this property or the view will not contain any records. (However, you can set it at run time.)
Sort	An expression indicating how to order the records represented by the view. (The data table itself is not sorted.) The most typical expression includes the name of a column in the table and, optionally, the qualifier ASC or DESC to indicate the sort direction. However, you can sort by the results of any expression, including calculations. For more information about syntax for sort expressions, see DataColumn.Expression Property.
RowFilter	An expression specifying which records to include in the view. The expression is evaluated as a Boolean expression; if the expression returns **true**, the record is included in the view. An example expression might look like the following: `Price > 10.00` For more information about syntax for filter expressions, see DataColumn.Expression Property.
RowStateFilter	The version of a record to include in the view. The default is **CurrentRows**, which gets a version of the row that includes any changes made to it. For more information about the versions of records in a data table, see Introduction to Dataset Updates
AllowEdit, AllowNew, AllowDelete	Specifies whether the view allows update operations (edit, new, or delete).

See Also

Introduction to Filtering and Sorting in Datasets | Filtering and Sorting Data Using Data Views | Working with Records in Data Views

Filtering and Sorting Data Using Data Views

Sorting with a data view allows you to set sort criteria at design time and provides an object you can use for data binding. For an overview, see Introduction to Filtering and Sorting in Datasets.

You can filter and sort using a data view that you have explicitly added to a form or component. Doing this allows you to set filter and sort options at design time. Alternatively, you can use the default data view — called **DefaultView** — that is automatically available for every table in a dataset. When you use the default view, you can specify filter and sort options only in code.

To filter and sort using a data view

1. If you want to set data view options at design time, add a data view to your form or component. For more information, see Adding Data Views to a Form or Component.
2. Set the data view's Sort property using a sort expression. The sort expression can include the names of data table columns or a calculation. If you set the sort expression at run time, the data view reflects the change immediately.

   For more information about syntax for sort expressions, see DataColumn.Expression Property..

3. Set the data view's RowFilter property using a filter expression. The filter expression should evaluate to **true** or **false**, as in this expression:

   ```
   CustomerStatus = 'Active'
   ```

   For more information about syntax for filter expressions, see DataColumn.Expression Property.

   > **Note** If a filter expression contains reserved characters, such as a single quotation mark, those characters must be specified using escape characters. For example, the following expression shows how to use an escape character to include an apostrophe in the expression:
   > CompanyName = 'Margie\'s Travel'
   > .

4. To filter based on a version or state of a record, set the **RowStateFilter** property to a value from the DataViewRowState Enumeration, such as the following:

   ```
   ' Visual Basic
   Dataview1.RowStateFilter = DataViewRowState.CurrentRows

   // C#
   Dataview1.RowStateFilter = DataViewRowState.CurrentRows;
   ```

The following example shows how to set the data view's filter and sort order at run time using the default data view:

```
' Visual Basic
Dataset1.Customers.DefaultView.Sort = "City"

// C#
Dataset1.Customers.DefaultView.Sort = "City";
```

The following example shows how to explicitly add a data view to a form. It checks the values of two radio buttons, and depending on which one is checked, sets a different sort order in the data view. The view filters only the current records. The results are displayed by binding an existing data grid to the data view.

```
' Visual Basic
If radioCity.Checked Then
    DataView1.Sort = "City"
ElseIf radioCompanyName.Checked Then
    DataView1.Sort = "CompanyName"
End If
Dataview1.RowFilter = "CustomerStatus = 'Active'"
Dataview1.RowStateFilter = DataViewRowState.CurrentRows
Me.DataGrid1.DataSource = DataView1
```

```
// C#
if (radioCity.Checked)
{
   DataView1.Sort = "City";
}
else if (radioCompanyName.Checked)
{
   DataView1.Sort = "CompanyName";
}

DataView1.RowFilter = "CustomerStatus = 'Active'";
DataView1.RowStateFilter = DataViewRowState.CurrentRows;
// The following calls a different method to display the
// data view contents.
this.dataGrid1.DataSource = DataView1;
```

See Also

Introduction to Filtering and Sorting in Datasets | Adding Data Views to a Form or Component | Filtering And Sorting Directly in Data Tables | Working with Records in Data Views | Data Binding and Windows Forms | Data Access in Web Forms Pages

Filtering and Sorting Directly in Data Tables

You can filter and sort the contents of a data table directly by calling a table's Select method. This strategy allows you to filter and sort only at run time. If you want to set up criteria for filtering and sorting at design time, and especially if you want to bind controls to the results of filtering or sorting, use a data view. For more information, see Filtering and Sorting Data Using Data Views.

When you sort directly in the data table, you do not reorder the contents of the table. Instead, you work with a result set representing the sorted records.

To filter and sort directly in a data table

- Call the data table's Select method, passing it up to three parameters:

  ```
  datatable.Select(filterExp, sortExp, rowstatevalue)
  ```

  The following table lists the parameter values.

Parameter	Description
filterExp	A filter expression that should evaluate to **true** or **false**.
sortExp	A sort expression. This is typically the name of a table column, but can be any calculated value. **Note** For more information about the syntax of filter and sort expressions, see DataColumn.Expression Property.
rowstatevalue	A value indicating what version or state you want to filter on. The acceptable values for this parameter are members of the DataViewRowState enumeration, such as **DataViewRowState.CurrentRows**. For more information about row state and row version, see Introduction to Dataset Updates.

  You can call the method with only the first or only the first and second parameters. To skip a parameter, pass an empty string as a placeholder.

  The **Select** method returns an array of data rows.

The following example shows how to filter and sort the Customers data table in a dataset called dataSet1. The filter expression selects customers whose status is active. The sort expression causes the results to be sorted by the City column, and the final parameter specifies that you want to see only the current (that is, post-change) versions of the records. The sorted list is displayed in a list box that is assumed to exist on the form.

```
' Visual Basic
Dim filterExp As String = "Status = 'Active'"
Dim sortExp As String = "City"
Dim drarray() As DataRow
Dim i As Integer
drarray = dataSet1.Customers.Select(filterExp, sortExp, DataViewRowState.CurrentRows)
For i = 0 To (drarray.Length - 1)
   listBox1.Items.Add( drarray(i)("City").ToString )
Next

// C#
string filterExp = "Status = 'Active'";
string sortExp = "City";
DataRow[] drarray;
drarray = dataSet1.Customers.Select(filterExp, sortExp, DataViewRowState.CurrentRows);
for (int i=0; i < drarray.Length; i++)
{
   listBox1.Items.Add(drarray[i]["City"].ToString());
}
```

The following example shows how to call the **Select** method to filter only on row state (skipping the filter and sort expressions), returning only deleted records:

```
' Visual Basic
drarray = dataSet1.Customers.Select(Nothing, Nothing, _
   DataViewRowState.Deleted)

// C#
drarray = dataSet1.Customers.Select(null, null,
   DataViewRowState.CurrentRows);
```

See Also

Introduction to Filtering and Sorting in Datasets | Filtering and Sorting Data Using Data Views | DataTable.Select Method

Working with Records in Data Views

When you are working with a data view, you can access filtered or sorted records by getting them from the data view instead of directly from the table they are in. You can also update, insert, and delete records via the data view, subject to some restrictions:

- The data view must include enough information about each record to be able to determine where in the data table the record goes. This information can include a primary key or other columns that together provide enough information to uniquely identify a record, such as a customer's name, address, and city.
- The data view's AllowEdit, AllowNew, and AllowDelete properties must be set to **true** for each respective operation.

Finding Records

To find records in a data view

1. Set the data view's **Sort** property to the column or columns you want to search.
2. Call the Find or FindRows method of a data view, passing it the value to look for in the sorted columns.

   - If you want to find a single record, call the **Find** method.

     - or -

   - If you want to find more than one record, use the **FindRows** method. For more information see, Searching a DataView.

The following example shows how to return the index of the found row based on the value in a text box control:

```
' Visual Basic
DataView1.Sort = "CustomerID"
Dim foundIndex as Integer = DataView1.Find(TextBox1.Text)

// C#
dataView1.Sort = "CustomerID";
int foundIndex = dataView1.Find(textBox1.Text);
```

Reading Records

To read records in a data view

- Use an index value to point to the record in the data view that you want to access.

You can access a column by referencing it by name within a data view, as in the following example, which gets the customer name from the first record in the view:

```
' Visual Basic
Dim DataView1 as New DataView(ds.Customers)
Dim cname As String = CType(DataView1(0)("CustomerName"), String)

// C#
DataView dataView1 = new DataView(ds.Customers);
string cname = dataView1[0]["CustomerName"].ToString();
```

The following example shows how to loop through all the records in a data view that points to a Customers table. The example gets the values of three fields (CustomerId, CompanyName, and City) from each record, concatenates all the data into a delimited string, and displays the string in a text box.

```
' Visual Basic
Dim drv As DataRowView
Dim s As String = ""
For Each drv In DataView1
   s &= drv("CustomerID").ToString & "    "
   s &= drv("CompanyName").ToString & "    "
   s &= drv("City").ToString & ControlChars.CrLf
Next
```

```
TextBox1.Text = s

// C#
string s = "";
foreach (DataRowView drv in dataView1)
{
   s += drv["CustomerID"].ToString() + "    ";
   s += drv["CompanyName"].ToString() + "    ";
   s += drv["City"].ToString() + "\n";
}
textBox1.Text = s;
```

Updating Records

To update records via a data view

- In code, use an index value to identify the record to update, and then set the column value by referencing its name.

  Note You cannot edit records via the data view if its **AllowEdit** property is set to **false**.

The following example shows how to identify and update a column.

```
' Visual Basic
DataView1(0)("CompanyName") = "Fabrikam, Inc."

// C#
dataView1[0]["CompanyName"] = "Fabrikam, Inc.";
```

Inserting Records

To insert records via a data view

1. Call the data view's AddNew method, which creates a new record and returns a DataRowView object:

   ```
   ' Visual Basic
   Dim drv As DataRowView
   drv = DataView1.AddNew()

   // C#
   DataRowView drv;
   drv = dataView1.AddNew();
   ```

2. Update the record as you would any data view record.

   Note You cannot insert records via the data view if its **AllowNew** property is set to **false**.

The following example shows how to add a new record to a view, updating three fields in it:

```
' Visual Basic
Dim drv As DataRowView
drv = DataView1.AddNew()
drv("CustomerID") = "AAA"
drv("CompanyName") = "AAFabrikam, Inc."
drv("City") = "Aurora"

// C#
DataRowView drv;
drv = dataView1.AddNew();
drv["CustomerID"] = "AAA";
drv["CompanyName"] = "AAFabrikam, Inc.";
drv["City"] = "Aurora";
```

Deleting Records

To delete records via a data view

- Call the data view's Delete method, passing it the index of the record to delete.

  Note You cannot delete records via the data view if its **AllowDelete** property is set to **false**.

The following example shows how to delete a record:

```
' Visual Basic
DataView1.Delete(0)

// C#
dataView1.Delete(0);
```

See Also

Introduction to Filtering and Sorting in Datasets | Adding Data Views to a Form or Component | Filtering and Sorting Data Using Data Views | Searching a DataView

Creating and Working with Data View Managers

A data view manager is an object that contains a collection of data views, one for each table in a dataset. The data view manager provides centralized access to sorted and filtered data. This is especially helpful when working with related tables. For an overview, see Introduction to Filtering and Sorting in Datasets.

> **Note** You must create and configure data view managers in code; there is no design-time object for a data view manager.

To create and configure a data view manager

1. Create a dataset. For more information see Creating Typed Datasets with the Component Designer.
2. Add code to populate the dataset. For more information, see Populating a DataSet from a DataAdapter.
3. Create an instance of the DataViewManager class.
4. Set the data view manager's DataSet property to the dataset you created in Step 1.
5. Set sort and filter expressions by accessing individual tables through the DataViewSettings collection.

   > **Note** Accessing a table using the **DataViewSettings** property causes the data view manager to dynamically create a data view for that table. (The data view does not exist until you access it.)

6. If you want to bind controls to the data view manager, set their data-binding properties after configuring the data view manager. For example, in a Windows Form, you can set a control's **DataSource** property to the data view manager and its **DataMember** property to the name of the table (and thus, data view) to bind to.

The following example shows how to create a data view manager and set sort and filter information. The example assumes that there is a dataset called dsCustomersOrders with a Customers and an Orders table. After creating the data view manager, the example binds a data grid to the Titles table available via the data view manager.

```
' Visual Basic
Dim dvm As New DataViewManager()
dvm.DataSet = dsCustomersOrders1
dvm.DataViewSettings("Customers").Sort = "CompanyName"
dvm.DataViewSettings("Orders").Sort = "OrderDate"
Datagrid1.DataSource = dvm
Datagrid1.DataMember = "titles"

// C#
DataViewManager dvm = new DataViewManager();
dvm.DataSet = dsCustomersOrders1;
dvm.DataViewSettings["Customers"].Sort = "CompanyName";
dvm.DataViewSettings["Orders"].Sort = "OrderDate";
dataGrid1.DataSource = dvm;
dataGrid1.DataMember = "titles";
```

See Also

Introduction to Filtering and Sorting in Datasets | Filtering and Sorting Data Using Data Views | Data Binding and Windows Forms | Data Access in Web Forms Pages

Relationships in ADO.NET Datasets

A dataset can contain multiple tables, including tables that have an implicit relationship between them — tables with key fields in common. To take advantage of this implicit relationship, you use an ADO.NET **DataRelation** object. The following topics provide information about ADO.NET **DataRelation** objects, how to create them, and how to use them to work with data in related tables.

In This Section

Introduction to DataRelation Objects
Provides an overview of how datasets allow you to specify relationships between tables and how you can take advantage of these relationships.

Creating DataRelation Objects with the XML Designer
Explains how to use the XML Designer to add a **DataRelation** object to a dataset.

Related Topics

DataRelation Class
Provides reference information about the **DataRelation** class and links to topics about each member of the class.

Introduction to Datasets
Provides an overview of what ADO.NET dataset are, how they can by typed or untyped, what they are used for, and how they are structured.

Introduction to Data Adapters
Provides an overview of what data adapters are and what they are used for.

Navigating a Relationship between Tables
Describes how to write code that uses a **DataRelation** object to find related records in a data table.

Introduction to DataRelation Objects

There are many times when your application needs to work with related tables. Although a dataset contains tables and columns as in a database, it does not inherently include a database's ability to relate tables. However, you can create DataRelation objects that establish a relationship between a parent (master) and a child (detail) table based on a common key.

For example, a dataset containing sales data might have a Customers table and an Orders table. Even if the tables contain a key in common (in this example, perhaps CustomerID), the dataset itself does not keep track of the records in one table that relate to those in another. But you can create a **DataRelation** object that references the parent and child tables (and their keys), and then use the object to work with the related tables.

The **DataRelation** object can perform two functions:

- It can make available the records related to a record you are working with. It provides child records if you are in a parent record, and a parent record if you are working with a child record.
- It can enforce constraints for referential integrity, such as deleting related child records when you delete a parent record.

It is important to understand the difference between a true join and the function of a **DataRelation** object. In a true join, records are taken from parent and child tables and put into a single, flat recordset. When you use a **DataRelation** object, no new recordset is created. Instead, the relation tracks the relationship between tables and keeps parent and child records in synch.

Accessing Related Records

If tables in a dataset have a logical relationship, a **DataRelation** object can make available to you the related records in another table. An example might be a dataset containing a customer database with the tables Customers (with the primary key CustomerID) and Orders (with a foreign key CustomerID). You can create a **DataRelation** object whose properties are set to reflect these keys.

You can then use the **DataRelation** object to get related records. The process is indirect, in that you do not join tables. Instead, you call the GetChildRows method of a data row in the parent table, passing to it the **DataRelation** object that defines the parent/child relationship. The method returns an array of related child records.

The following code illustrates a simple example of getting related records. In this example, the array `drarray` is set to the child records of the first row in the Customers table.

```
' Visual Basic
Dim RowCtr As Integer
' Dim an array of datarows to hold the child records.
Dim drarray() As DataRow
' GetChildRows gets related rows. It is a method on the datatable, and
' takes a DataRelation name as a string.
RowCtr = 0
drarray = dsCustomersOrders1.Customers(RowCtr).GetChildRows("CustomersOrders")

// C#
int rowCtr;
// Declare an array of datarows to hold the child records.
DataRow[] drarray;
// GetChildRows gets related rows. It is a method on the datatable, and
// takes either DataRelation object name as a string.
rowCtr = 0;
drarray = dsCustomersOrders1.Customers[rowCtr].GetChildRows("CustomersOrders");
```

Similarly, you can get the parent row of a given child record by calling the GetParentRow method of a data row in the child table. In that case, the method does not return an array, but a single data row. For more information, see Navigating a Relationship between Tables.

DataRelation Objects and Constraints

A **DataRelation** object is also used to create and enforce the following constraints:

- A unique constraint, which guarantees that a column in the table contains no duplicates.
- A foreign-key constraint, which can be used to maintain referential integrity between a parent and child table in a dataset.

Constraints that you specify in a **DataRelation** object are implemented by automatically creating appropriate objects or setting

properties. If you create foreign-key constraint using the **DataRelation** object, instances of the ForeignKeyConstraint class are added to the data relation's ChildKeyConstraint property.

A unique constraint is implemented either by simply setting the Unique property of a data column to **true** or by adding an instance of the UniqueConstraint class to the **DataRelation** object's ParentKeyConstraint.

As part of the foreign-key constraint, you can specify referential integrity rules that are applied at three points:

- when a parent record is updated
- when a parent record is deleted
- when a change is accepted or rejected

The rules you can make are specified in the Rule enumeration and are listed in the following table.

Foreign-key constraint rule	**Action**
Cascade	The change (update or delete) made to the parent record is made in related records in the child table as well.
SetNull	Child records are not deleted, but the foreign key in the child records is set to **DBNull**. With this setting, child records can be left as "orphans" — that is, they have no relationship to parent records. **Note** Using this rule can result in invalid data in the child table.
SetDefault	The foreign key in the related child records is set to its default value (as established by the column's DefaultValue property).
None	No change is made to related child records. With this setting, child records can end up containing references to invalid parent records.
Default	The default value is **Cascade**.

For more information about updates in dataset tables, see Dataset Updates in Visual Studio .NET.

Constraint-only Relations

When creating a **DataRelation** object, you have the option of specifying that the relation will be used only to enforce constraints — that is, it will not also be used to access related records. This option allows you to generate a dataset that is slightly more efficient and that contains fewer methods than one with the related-records capability. However, you will not be able to access related records.

See Also

Creating DataRelation Objects with the XML Designer | Creating Typed Datasets with the Component Designer | Creating XML Schemas and Datasets

Creating DataRelation Objects with the XML Designer

DataRelation objects are part of a dataset's structure, along with the tables and columns defined in the dataset. For typed datasets, therefore, **DataRelation** objects are defined in the dataset schema. You create **DataRelation** objects in a dataset using the XML Designer. You do not have to know XML or the schema format in order to do so.

To create a DataRelation object in a dataset

1. If it does not already exist, create the schema for the dataset. For more information, see Visual Studio Tools for Creating Datasets.
2. In Solution Explorer, double-click the XML Schema (.xsd) file to open it in the XML Designer.

   The schema is opened in **Schema** view, with each table displayed in a gridlike control.

3. From the **Data** tab of the Toolbox, drag a **Relation** object onto the child table.

   The **Relation (KeyRef) Editor** dialog box opens.

4. If you want to assign a new name to the **DataRelation** object, overwrite the value in the **Name** box.
5. Make sure that the **Parent Element** selector is set to the name of the parent table and the **Child Element** selector to the name of the child table for which you want to create the **DataRelation** object.
6. From the **Key** list, choose the key in the parent table that you want to create a **DataRelation** object for.

   > **Note** The items presented in the **Key** list are not the names of the primary key columns. Instead, they are schema elements that can reference any column with a unique constraint (including, but not limited to, the primary key). There can be multiple key elements for a table, even for a table with a single-part primary key. For more information, see Introduction to Data Access with ADO.NET.

7. In the **Fields** grid, make sure that the columns for the parent table primary key (**PK Field**) and the child table foreign key (**FK Field**) column are set appropriately. If necessary, you can overwrite the names of the columns in the grid, or you can type in the names of additional columns used to define the parent/child relationship.
8. If you want to create a relation used only to enforce constraints (not used to get related records from a child or parent table), check **Constraint only**.

   Creating a constraint-only relation results in a slightly more efficient dataset with fewer methods in it. However, you will not be able to access related records.

9. If you want the dataset to enforce referential integrity during updates and deletes, check the **Enforce Constraints** box and then select an enforcement rule from the lists under **Update Rule** and **Delete Rule**. For more information about constraints and what each setting means, see Introduction to DataRelation Objects.

   > **Note** If you do not check this box, the **DataRelation** object allows you to get related records from each table, but does not perform update operations on the related table in order to keep the tables synchronized.

10. Click **OK** to create the **DataRelation** object.

    The XML Designer draws a relationship line between the parent and child table, with the relation object represented as a gray square.

To edit a relation object

- In **Design** view of the XML Designer, right-click the relation object (the gray square) and choose **Edit Relation**.

  The **Relation (KeyRef) Editor** dialog box opens, displaying information about the current relation object.

To remove a relation object

- In **Design** view of the XML Designer, click the relation object (the gray square) to select it, and then from the **Edit** menu, choose **Delete**.

See Also

Introduction to DataRelation Objects | Introduction to Datasets | Visual Studio Tools for Creating Datasets | DataRelation Class

Performing Database Operations Directly

In many situations, you will design your data access strategy around using a dataset (an in-memory cache of data) and data adapters to fill the dataset and perform updates. In other situations, however, you might find it useful to bypass datasets and communicate directly with the data source. These situations include:

- Performing queries on data intended to be read-only in your application. This might include executing a command that performs a database lookup.
- Designing data access in ASP.NET Web applications, which under various circumstances might not benefit from using a dataset.
- Executing a query that returns only a single value, such as a calculation or the result of an aggregate function.
- Creating and modifying database structures, such as tables and stored procedures.

When you are creating tables and stored procedures or otherwise executing logic that does not return a result set, you cannot use a dataset; instead, you execute data commands directly. In the other situations, using data commands is an alternative to using data adapters and a dataset, and you can choose between the alternating ways to access data.

The following topics provide information on what data commands are and when and how to use them.

In This Section

Introduction to DataCommand Objects in Visual Studio
Provides an overview of data command objects and how to work with them.
Recommendations for Data Access Strategies
Provides information on choosing between datasets and data commands for data access.
Working with Data Commands: High-Level Process
Explains, at a high level, the steps involved in creating data commands and executing them.
Adding Data Commands to a Form or Component
Explains how to add a data command object to a form or component in Visual Studio.
Setting and Getting Data Command Parameters
Explains how to use the **Parameters** collection to pass data to commands and read return values after the command has been executed.
Executing a Data Command that Returns a Result Set
Provides instructions on how to execute a data command using a data reader that returns one or more records.
Executing Updates or Database Commands using a Data Command
Describes how to execute Update, Insert, and Delete statements directly against the database, and how to issue database definition language (DDL) commands.
Executing a Data Command that Returns a Single Value
Provides instructions on how to execute a command that returns a single value, such as a command that performs a lookup or calculates a value.

Related Topics

Web Data Access Strategy Recommendations
Provides guidelines for choosing whether to use direct access to a data source or a disconnected cache (dataset); and if you use a dataset, whether to store it between round trips.
Introduction to Datasets
Provides an overview of datasets and how they are used to cache data.
Introduction to Dataset Updates
Provides information on how data updates are made using datasets and data adapters.
Connecting to Data Sources with ADO.NET
Provides links to topics about creating and working with connection objects in Visual Studio.
Concurrency Control in ADO.NET
Provides information on how to work with concurrency control in ADO.NET.
Executing a Command
Provides information from the .NET Framework Developer's Guide about using data commands programmatically.
Retrieving Data Using the DataReader
Provides information from the .NET Framework Developer's Guide on using the data reader to work with the results of a command programmatically, including how to work with the results of multiple statements or procedures.
Obtaining Data as XML from SQL Server
Provides information from the .NET Framework Developer's Guide on how to get data in XML format from a SQL Server 7.0 or

later database using a data command and an **XMLReader** object.

Modifying Data in a Database

Provides information from the .NET Framework Developer's Guide about executing update commands against a database.

Performing Catalog Operations

Provides more code examples of how to execute database definition (DDL) commands.

Obtaining a Single Value from a Database

Provides information from the .NET Framework Developer's Guide on executing commands that return a scalar value.

Introduction to Data Access with ADO.NET

Explains the ADO.NET data model and provides an architectural diagram that shows where data commands fit in.

Adding a Launch Condition for Microsoft Data Access Components

Describes how to check that the correct version of the ADO.NET data access components is available when you install your application.

Recommendations for Data Access Strategies

ADO.NET assumes a model for data access in which you open a connection, get data or perform an operation, and then close the connection. ADO.NET provides two basic strategies for how you work with this model. One model is store data in a dataset, which is an in-memory cache of records you can work with while disconnected from the data source. To use a dataset, you create an instance of it and then use a data adapter to fill it from the data source. You then work with the data in the dataset — for example, by binding controls to dataset members. For more information, see Introduction to Datasets.

An alternative strategy is to perform operations directly against the database. In this model, you use a data command object that includes an SQL statement or a reference to a stored procedure. You can then open a connection, execute the command to perform the operation, and then close the connection. If the command returns a result set — that is, if the command performs a Select statement — you can fetch the data using a data reader, which acts like a highly efficient read-only cursor. The data reader can then act as a source for data binding. For more information, see Introduction to DataCommand Objects in Visual Studio.

Each strategy has specific advantages, as detailed in the sections below. You should choose a strategy based on what your data-access requirements are.

> **Note** When you deploy an application that includes Visual Studio data-access components, you must make sure that the user installing the application has version 2.7 or later of the Microsoft Data Access Components (MDAC). For more information, see Adding a Launch Condition for Microsoft Data Access Components.

Storing Data in Datasets

A common model for data access in Visual Studio .NET applications is to store data in datasets and use a data adapter to read and write data in the database. (A .NET application is one that uses the .NET Framework: the common language runtime and the managed classes.) The advantages of the dataset model are:

- **Working with multiple tables** A dataset can contain multiple tables of results, which it maintains as discrete objects. You can work with the tables individually or navigate between them as parent-child tables.
- **Manipulating data from multiple sources** The tables in a dataset can represent data from many different sources (for example, from different databases, from XML files, spreadsheets, and so on, all in the same dataset). Once the data is in the dataset, you can manipulate it and relate it in a homogeneous format as if it had come from a single source.
- **Moving data between tiers in a distributed application** By keeping data in a dataset, you can easily move it between the presentation tier, business tier, and data tier of your applications.
- **Data exchange with other applications** A dataset provides a powerful way to exchange with other components of your application and with other applications. Datasets include extensive support for features such as serializing data as XML and reading and writing XML Schemas.
- **Data binding** If you are working with forms, it is usually easier to bind controls to data in a dataset than it is to programmatically load data values into the control after executing a command.
- **Maintaining records for reuse** A dataset allows you to work with the same records repeatedly without requerying the database. Using dataset facilities, you can filter and sort records, and you can use the dataset as a source of data if you are paging.
- **Ease of programming** When you work with a dataset, you can generate a class file that represents its structure as objects (for example, a Customers table in the dataset can be accessed as the *dataset*.Customers object). This makes it easier, clearer, and less error-prone to program with, and is supported by Visual Studio tools such as IntelliSense, the Data Adapter Configuration wizard, and so on.

Performing Database Operations Directly

Alternatively, you can interact with the database directly. In this model, you use a data command object that includes an SQL statement or a reference to a stored procedure. You can then execute the command to perform the operation. If the command returns a result set — that is, if the command performs a Select statement — you can fetch the data using a data reader, which acts like a highly efficient read-only cursor.

> **Security Note** When using data commands with a CommandType property set to **Text**, carefully check information that is sent from a client before passing it to your database. Malicious users might try to send (inject) modified or additional SQL statements in an effort to gain unauthorized access or damage the database. Before you transfer user input to a database, you should always verify that the information is valid. A best practice is to always use parameterized queries or stored procedures when possible.

Performing database operations directly has specific advantages, which include:

- **Extra functionality** As noted, there are some operations, such as executing DDL commands, that you can only accomplish by executing data commands.
- **More control over execution** By using commands (and a data reader, if you are reading data), you get more direct control over how and when an SQL statement or stored procedure is executed and what becomes of the results or return values.
- **Less overhead** By reading and writing directly to the database, you can bypass storing data in a dataset. Because the dataset requires memory, you can reduce some overhead in your application. This is especially true in situations where you intend to use the data only once, such as displaying search results in a Web page. In that case, creating and filling a dataset might be an unnecessary step in displaying the data.
- **Less programming in some instances** In a few instances, particularly Web applications, there is some extra programming required to save the state of a dataset. For example, in Web Forms pages, the page is recreated with each round trip; unless you add programming to save and restore a dataset, it, too, is discarded and recreated with each round trip. If you use a data reader to read directly from the database, you avoid the extra steps required to manage the dataset.

Because of the stateless nature of Web applications and the corresponding issues associated with storing datasets, it is sometimes more practical in Web applications to work directly against the database. For more information, see Introduction to Data Access in Web Forms Pages.

Recommendations for Accessing Data

The following sections provide recommendations for which data-access strategy to use with specific types of applications.

Web Forms

In general, use data commands; to fetch data, use a data reader. Because Web Forms pages and their controls and components are recreated each time the page makes a round trip, it often is not efficient to create and fill a dataset each time, unless you also intend to cache it between round trips.

Use dataset under the following circumstances:

- You want to work with multiple separate tables or tables from different data sources.
- You are exchanging data with another application or a component such as an XML Web service.
- You need to perform extensive processing with each record you get from the database. If you use a data command and data reader, processing each record as you read it can result in the connection being held open for a long period, which in turn can affect the performance and scalability of your application.
- If your data processing involves interdependent records (for example, looking up information in related records).
- If you want to perform XML operations such as XSLT transformations on the data.
- If you prefer the ease of programming provided by datasets.

For more information, see Web Data Access Strategy Recommendations.

XML Web Services

XML Web services are ASP.NET Web applications, and therefore use the same model as Web Forms pages: the XML Web service is created and discarded each time a call is made to it. This suggests that the data-access model for an XML Web service is largely the same as it is for Web Forms. However, XML Web services are often middle-tier objects, and an important part of their purpose is often to exchange data with other applications across the Web.

Use a dataset if:

- Your XML Web service sends and receives data — for example, sending it as the return value of a method and receiving it as a method argument. This is a fundamental choice in XML Web services; even if there are other reasons you might consider using a data command, data exchange with other components almost always means that you should use a dataset.
- For any of the reasons listed above for Web Forms.

Use a data command (and if appropriate, a data reader) under the following circumstances:

- The XML Web service is retrieving a scalar value.
- The XML Web service is performing a non-query operation, such as a DDL command.
- The XML Web service is calling a stored procedure to execute logic within the database.

Windows Forms

In general, in a Windows Form, use a dataset. Windows Forms are typically used on rich clients where the form is not created and discarded (along with its data) with each user operation, as with Web Forms. Windows Forms applications also traditionally offer data-access scenarios that benefit from maintaining a cache of records, such as displaying records one by one in the form.

Specifically, use dataset under the following circumstances:

- If you are working with the same records repeatedly, such as allowing a user to navigate between records.
- If you are using the Windows Forms data-binding architecture, which is specifically designed to work with datasets.
- For any of the other reasons listed under Web Forms above.

Use a data command (and if appropriate, a data reader) under the following circumstances:

- If you are getting a scalar value from the database
- If you are performing a non-query operation, such as a DDL command.
- If you are getting read-only data to display in a form — for example, creating a report. Stated differently, if there is no need to keep the data available after accessing it, use a data command.

See Also

Introduction to DataCommand Objects in Visual Studio | Introduction to Datasets |
Working with Data Commands: High-Level Process | Executing a Command | Introduction to Dataset Updates

Working with Data Commands: High-Level Process

ADO.NET data commands give you the ability to execute commands directly against a database or other data source, without needing a dataset or data adapter. For more information, see Introduction to DataCommand Objects in Visual Studio.

> **Security Note** When using data commands with a CommandType property set to **Text**, carefully check information that is sent from a client before passing it to your database. Malicious users might try to send (inject) modified or additional SQL statements in an effort to gain unauthorized access or damage the database. Before you transfer user input to a database, you should always verify that the information is valid; best practice is to always use parameterized queries or stored procedures when possible.

To use a data command in Visual Studio, you perform a set of steps, some at design time and others at run time, typically using code.

To work with data commands

1. At design time, add a data command to your form or component and set:

   - Its SQL command or stored procedure.
   - Any parameters that the command requires.

   For more information, see Adding Data Commands to a Form or Component.

   > **Note** As with all data components, you can also create and configure a data command programmatically at run time. For more information, see Executing a Command.

2. At run time, set any parameter values that you want to pass to the data command. You often get parameter values from information that the user enters into a form or that is passed from another component. For more information, see Setting and Getting Data Command Parameters.
3. Execute the command. This runs the SQL command or stored procedure associated with the command. The exact command you use to execute the data command depends on what you expect the command to return: a result set, a scalar value, a count of affected value, or XML data.
4. Read the return value or values from the data command. As with executing the command, the way you read results vary depending on what type of command you executed and what it returns.

   For more information about different ways to execute commands and get their results, see the following topics:

   - Executing a Data Command that Returns a Result Set
   - Executing Updates or Database Commands using a Data Command
   - Executing a Data Command that Returns a Single Value
   - Obtaining Data as XML from SQL Server

See Also

Introduction to DataCommand Objects in Visual Studio | Retrieving Data Using the DataReader | Executing a Command | Obtaining Data as XML from SQL Server | Modifying Data in a Database | Performing Catalog Operations | Obtaining a Single Value from a Database

Adding Data Commands to a Form or Component

ADO.NET data commands give you the ability to execute commands directly against a database or other data source, without needing a dataset or data adapter. For more information, see Introduction to DataCommand Objects in Visual Studio.

Before executing a data command, you add an instance of a command object to your form or component and configure it. Visual Studio includes tools to help you do this at design time.

> **Security Note** When using data commands with a CommandType property set to **Text**, carefully check information that is sent from a client before passing it to your database. Malicious users might try to send (inject) modified or additional SQL statements in an effort to gain unauthorized access or damage the database. Before you transfer user input to a database, you should always verify that the information is valid; best practice is to always use parameterized queries or stored procedures when possible. For more information, see Security Considerations regarding SQL Statements.

To add a data command to a form or component

1. If you do not already have a connection object available on the form or component, add one. For more information, see Creating ADO.NET Connection Objects.
2. From the **Data** tab of the Toolbox, drag an OleDbCommand, SqlCommand, OdbcCommand, or OracleCommand object onto your form or component.

   > **Note** Data adapters, data connections, data commands, and data readers are the components that make up a .NET Framework data provider. Microsoft and third-party providers can make available other .NET Framework data providers that can be integrated into Visual Studio. For more information, see .NET Framework Data Providers.

3. Set the following properties for the data command.

Property	Description
(Name)	The name by which you want to refer to the command in code.
Connection	A reference to a connection object that the command will use to communicate with the database. You can select an existing connection from the drop-down list.
CommandType	A value specified by the CommandType enumeration indicating what type of command you want to execute: • **Text** An SQL statement. • **StoredProcedure** A stored procedure. • **TableDirect** A way of fetching the entire contents of a table. (**TableDirect** is only supported by the .NET Framework Data Provider for OLE DB.)
CommandText	The command to execute. The exact text you specify depends on the value of the **CommandType** property: • **Text** Enter the actual SQL statement to execute. • **StoredProcedure** Enter the name of the stored procedure. • **TableDirect** Enter the name of the table to fetch. (**TableDirect** is only supported by the .NET Framework Data Provider for OLE DB.)
Parameters	A collection of parameter objects of type (OleDbParameter, SqlParameter, OdbcParameter, or OracleParameter). You can pass data to the command by setting properties of individual parameters. For more information on configuring parameters, see Configuring Parameters for Data Adapters. For more information about passing and receiving parameter values, see Setting and Getting Data Command Parameters.

Next Steps

After you have added a data command, you need to execute it.

If you want to return a ...	See
Result set	Executing a Data Command that Returns a Result Set
Count of affected records	Executing Updates or Database Commands Using a Data Command
Scalar value	Executing a Data Command that Returns a Single Value

Result set as XML	Obtaining Data as XML from SQL Server

See Also

Introduction to DataCommand Objects in Visual Studio | Working with Data Commands: High-Level Process | Parameters in Data-Adapter Commands | Configuring Parameters for Data Adapters | Setting and Getting Data Command Parameters

Setting and Getting Data Command Parameters

If you are using data commands directly to execute commands against a database, the SQL statements or stored procedures you are executing often require parameters. For example, a simple Update statement that requires parameters might look like the following:

```
UPDATE Employees
SET LastName = @LastName, FirstName = @FirstName, BirthDate = @BirthDate
WHERE (EmployeeID = @Emp_id)
```

When this statement is executed, you must provide values for all the parameters (@LastName, @FirstName, @BirthDate, and @Emp_id). To do so, you use parameter objects. Data commands support a **Parameters** collection that contains a set of objects of type OleDbParameter , SqlParameter, OdbcParameter or OracleParameter class. There is one parameter object in the collection for each parameter you need to pass. In addition, if you are calling a stored procedure, you might need an additional parameter to accept the procedure's return value.

> **Note** For Oracle, when using named parameters in an SQL statement or stored procedure, you must precede the parameter name with a colon (:). However, when referring to a named parameter elsewhere in your code (for example, when calling the **Add** method), do not precede the named parameter with a colon (:). The data provider supplies the colon automatically.

> **Note** If you are using a dataset, you typically do not execute data commands directly. Instead, you use adapter-specific ways to set parameters. For more information, see Using Parameters with a DataAdapter.

You can configure a command's **Parameters** collection using the Properties window or in code. For more information about using the Properties window, see Configuring Parameters for Data Adapters. For an example of how to create and configure parameters programmatically, see Using Stored Procedures with a Command.

> **Tip** It is a good idea to give names to your parameter objects so that you can reference them in code easily. Although you can reference a parameter object using its index value in the **Parameters** collection, it is clearer and less error-prone to refer to it by name.

Before you execute a command, you must set a value for every parameter in the command.

To set a parameter value

- For each parameter in the command's **Parameters** collection, set its **Value** property to the value to pass.

  The following example shows how to set parameters before executing a command that references a stored procedure. The sample assumes that you have already configured the **Parameters** collection with three parameters named au_id, au_lname, and au_fname. The individual parameters are set by name to make it clear which parameter is being set.

```
' Visual Basic
' The CommandText and CommandType properties can be set
' in the Properties window but are shown here for completeness.
With OleDbCommand1
   .CommandText = "UpdateAuthor"
   .CommandType = System.Data.CommandType.StoredProcedure
   .Parameters("au_id").Value = listAuthorID.Text
   .Parameters("au_lname").Value = txtAuthorLName.Text
   .Parameters("au_fname").Value = txtAuthorFName.Text
End With
OleDbConnection1.Open()
OleDbCommand1.ExecuteNonQuery()
OleDbConnection1.Close()

// C#
// The CommandText and CommandType properties can be set
// in the Properties window but are shown here for completeness.
OleDbCommand1.CommandText = "UpdateAuthor";
OleDbCommand1.CommandType = System.Data.CommandType.StoredProcedure;
```

```
OleDbCommand1.Parameters["au_id"].Value = listAuthorID.Text;
OleDbCommand1.Parameters["au_lname"].Value = txtAuthorLName.Text;
OleDbCommand1.Parameters["au_fname"].Value = txtAuthorFName.Text;
OleDbConnection1.Open();
OleDbCommand1.ExecuteNonQuery();
OleDbConnection1.Close();
```

Getting Return Values

Stored procedures often pass values back to the application that called them. They can do so by passing the value using a parameter or by defining and passing a return value.

To get values returned by procedures

1. Create parameters whose **Direction** property is set to **Output** or **InputOutput** (if the parameter is used in the procedure to both receive and send values). Make sure the data type of the parameter matches the expected return value.
2. After executing the procedure, read the **Value** property of the parameter being passed back.

To get a procedure's return value

1. Create a parameter and set its **Direction** property is set to **ReturnValue**.

   Note The parameter object for the return value must be the first item in the **Parameters** collection.

2. Make sure the parameter's data type matches the expected return value.

   Note Update, Insert, and Delete SQL statements return an integer value indicating the number of records affected by the statement. You can get this value as the return value of the **ExecuteNonQuery** method. For more information, see Executing Updates or Database Commands using a Data Command.

The following example shows how to get the return value of a stored procedure called CountAuthors. In this case, it is assumed that the first parameter in the command's **Parameters** collection is named "retvalue" and that is configured with a direction of **ReturnValue**.

```
' Visual Basic
Dim cntAffectedRecords As Integer
' The CommandText and CommandType properties can be set
' in the Properties window but are shown here for completeness.
OleDbcommand1.CommandText = "CountAuthors"
OleDbCommand1.CommandType = CommandType.StoredProcedure
OleDbConnection1.Open()
OleDbCommand1.ExecuteNonQuery()
OleDbConnection1.Close()
cntAffectedRecords = CType(OleDbCommand1.Parameters("retvalue").Value, Integer)
MessageBox.Show("Affected records = " & cntAffectedRecords.ToString)

// C#
int cntAffectedRecords;
// The CommandText and CommandType properties can be set
// in the Properties window but are shown here for completeness.
oleDbcommand1.CommandText = "CountAuthors";
oleDbCommand1.CommandType = CommandType.StoredProcedure;
oleDbConnection1.Open();
oleDbCommand1.ExecuteNonQuery();
oleDbConnection1.Close();
cntAffectedRecords = (int)(OleDbCommand1.Parameters["retvalue"].Value);
MessageBox.Show("Affected records = " + cntAffectedRecords.ToString());
```

See Also

Introduction to DataCommand Objects in Visual Studio | Working with Data Commands: High-Level Process | Parameters in Data-Adapter Commands | Configuring Parameters for Data Adapters | Executing a Data Command that Returns a Result Set | Executing Updates or Database Commands using a Data Command | Executing a Data Command that Returns a Single Value

Executing a Data Command that Returns a Result Set

You can use a data command to execute a database procedure that returns a read-only result set — that is, you can execute a SQL Select statement or a stored procedure that contains a Select statement. This is similar to what happens when you fill a dataset using a data adapter, except that the result set is returned directly to you.

If you don't have other reasons to use a dataset (for example, you want to share the data with another component or application), it can be faster to use a data command to get a result set directly rather than load it into the dataset. For more information on the relative advantages of datasets versus data commands, see Recommendations for Data Access Strategies.

You can return zero or more records using a data command. You might therefore use a data command for the following:

- To return a set of records that you fill into a control on a form. This is especially useful in Web Forms pages, where you often creating read-only listings (such as search results or inventory lists).
- To return a single record, which you pinpoint by passing parameters to the command that identify the record by ID or other unique value. You might pass an ID to a command to get back a corresponding record, or you might be filling a single record into a data view so you can bind a set of form controls to the data view.
- To execute a command that returns catalog information from a database.

A result set is returned to you in a data reader (an object of type OleDbDataReader, SqlDataReader, OdbcDataReader, or OracleDataReader). You can then loop through this object, extracting individual records from the result set. Because the data reader provides forward-only, read-only access to the data, it is very fast.

To execute a command that returns a result set

1. Add a data command to your form or component. For more information, see Adding Data Commands to a Form or Component.
2. Set the data command object's **CommandText** property to an SQL statement or the name of a stored procedure that returns a result set.
3. Set the **CommandType** property to **CommandType.Text** (for a statement) or **CommandType.StoredProcedure** (for a stored procedure). These values are defined in the CommandType enumeration.
4. If the command takes parameters, set them. For more information, see Setting and Getting Data Command Parameters.
5. Create a data reader object as an instance of either the OleDbDataReader, SqlDataReader, OdbcDataReader, or OracleDataReader class.
6. Open the connection associated with the data command.
7. Call the command's **ExecuteReader** method, setting the result to the data reader you created in Step 5.

   Tip The **ExecuteReader** method allows you to pass a parameter indicating command behavior, including options to close the connection immediately, return only a single record, and return keys only. For more information, see OleDbCommand.ExecuteReader Method (CommandBehavior) SqlCommand.ExecuteReader Method (CommandBehavior), OdbcCommand.ExecuteReader Method (CommandBehavior), or OracleCommand.ExecuteReader Method (CommandBehavior).

8. Loop through the data reader using its **Read** method until the method returns false.
9. Close the reader.
10. If you did not do so in executing the **ExecuteReader** method, close the connection.

The following example shows how to read the records in the Authors table of the SQL Server pubs database. The example uses the command type of **TableDirect** and specifies the name of the Authors table as the command text. (Both of these settings can be made in the Properties window, but are shown here for completeness.) When the ExecuteReader method has been called, the example loops through the data reader and builds up a delimited string of author information that it displays in a text box.

Note In the example, column information is extracted by referring to its column name in the data reader. You can optimize access by using more efficient ways to get the column information. For more information, see Retrieving Data Using the DataReader.

```
' Visual Basic
Dim dreader As System.Data.OleDb.OleDbDataReader
OleDbCommand1.CommandText = "authors"
OleDbCommand1.CommandType = CommandType.TableDirect
```

```
OleDbConnection1.Open()
dreader = OleDbCommand1.ExecuteReader(CommandBehavior.CloseConnection)
Dim s As String = ""
While dreader.Read()
   s &= dreader("au_id").ToString() _
     & vbTab & dreader("au_fname").ToString() _
     & " " & dreader("au_lname").ToString & ControlChars.CrLf
End While
TextBox1.Text = s
dreader.Close()

// C#
System.Data.OleDb.OleDbDataReader dreader;
OleDbCommand1.CommandText = "authors";
OleDbCommand1.CommandType = CommandType.TableDirect;
OleDbConnection1.Open();
dreader = OleDbCommand1.ExecuteReader(CommandBehavior.CloseConnection);
string s = "";
while (dreader.Read())
{
   s += dreader["au_id"].ToString() + "\t"
     + dreader["au_fname"].ToString() + " "
     + dreader["au_lname"].ToString() + "\n";
}
TextBox1.Text = s;
dreader.Close();
```

The following example shows two ways to use a data reader to get data. The first part is similar to the previous example, but it uses a data reader to load a **ListBox** control in a Windows Form with just the **author_id** column from the authors table.

The second part is a handler for the list box control's **SelectedIndexChanged** event. The method uses the au_id value that the user has selected as the parameter for a SQL Select statement written to return only one record (by selecting on the primary key). Because the data reader is expected to return only one record, it is not in a loop; instead, its **Read** method is called only one. (The example does check that there is at least one record by calling the **Read** method in an if statement.)

The two event handlers use different data commands; the first one takes no parameters, and the second one takes a single parameter for the author id.

```
' Visual Basic
Private Sub btnLoadListBox_Click(ByVal sender As System.Object, _
ByVal e As System.EventArgs) Handles btnLoadListBox.Click
   ' This method fills a ListBox control with author ids.
   Dim dreader As System.Data.OleDb.OleDbDataReader
   OleDbCommand1.CommandText = "authors"
   OleDbCommand1.CommandType = CommandType.TableDirect
   OleDbConnection1.Open()
   dreader = OleDbCommand1.ExecuteReader()
   While dreader.Read()
      ListBox1.Items.Add(dreader("au_id"))
   End While
   dreader.Close()
   OleDbConnection1.Close()
End Sub

Private Sub ListBox1_SelectedIndexChanged(ByVal sender As System.Object, _
ByVal e As System.EventArgs) Handles ListBox1.SelectedIndexChanged
   Dim dreader As System.Data.OleDb.OleDbDataReader
   ' The following two properties can be set in the Properties window
   ' but are shown here for completeness.
   OleDbCommand2.CommandText = _
      "select au_id, au_lname, au_fname from authors where au_id = ?"
   OleDbCommand2.CommandType = CommandType.Text
   OleDbCommand2.Parameters("au_id").Value = ListBox1.Text
   OleDbConnection1.Open()
   dreader = OleDbCommand2.ExecuteReader()
   If dreader.Read() Then
      txtAuthorID.Text = dreader("au_id").ToString()
```

```
        txtAuthorLastName.Text = dreader("au_lname").ToString()
        txtAuthorFirstName.Text = dreader("au_fname").ToString()
    End If
    dreader.Close()
    OleDbConnection1.Close()
End Sub

// C#
private void btnLoadListBox_Click(object sender, System.EventArgs e)
{
    // This method fills a ListBox control with author ids.
    System.Data.OleDb.OleDbDataReader dreader;
    OleDbCommand1.CommandText = "authors";
    OleDbCommand1.CommandType = CommandType.TableDirect;
    OleDbConnection1.Open();
    dreader = OleDbCommand1.ExecuteReader();
    while( dreader.Read())
    {
        ListBox1.Items.Add(dreader["au_id"]);
    }

    dreader.Close();
    OleDbConnection1.Close();
    Button6.Enabled = true;
}

private void ListBox1_SelectedIndexChanged(object sender,
System.EventArgs e)
{
    System.Data.OleDb.OleDbDataReader dreader;
    // The following two properties can be set in the Properties window
    // but are shown here for completeness.
    OleDbCommand2.CommandText =
        "select au_id, au_lname, au_fname from authors where au_id = ?";
    OleDbCommand2.CommandType = CommandType.Text;
    OleDbCommand2.Parameters["au_id"].Value = ListBox1.Text;
    OleDbConnection1.Open();
    dreader = OleDbCommand2.ExecuteReader();
    if (dreader.Read())
    {
        txtAuthorID.Text = dreader["au_id"].ToString();
        txtAuthorLastName.Text = dreader["au_lname"].ToString();
        txtAuthorFirstName.Text = dreader["au_fname"].ToString();
    }
    dreader.Close();
    OleDbConnection1.Close();
}
```

See Also

Introduction to DataCommand Objects in Visual Studio | Working with Data Commands: High-Level Process | Setting and Getting Data Command Parameters | Executing Updates or Database Commands using a Data Command | Executing a Data Command that Returns a Single Value | Obtaining Data As XML From SQL Server

Executing Updates or Database Commands using a Data Command

Some types of data commands that you can execute against a database do not return any values except a value indicating the success of the command. These types of commands include:

- Database definition (DDL) commands that you use to create and manage database structures such as tables and stored procedures.
- Update commands (Update, Insert, and Delete statements).

You must use database commands to execute DDL operations; there is no other way to perform these tasks in ADO.NET.

If you are using a dataset, you do not need to use a separate data command to execute database updates. Instead, you use the data adapter to update the database. For more information, see Dataset Updates in Visual Studio .NET. However, if you are not using a dataset, you can send update commands directly to the database. For more information on the relative advantages of datasets versus data commands, see Recommendations for Data Access Strategies.

With both types of commands, the command returns an integer indicating the success of the operation. The return value differs according to whether you are updating records or issuing a DDL command:

- If you are creating or modifying database structures, the return value is -1 if the operation was successful.
- If you are updating records, the return value indicates the number of records affected by the operation.

In either case, if the operation fails, the command returns zero.

To execute a command to update a database

1. Add a data command to your form or component and configure it with the SQL statement or stored procedure to execute. For more information, see Adding Data Commands to a Form or Component.
2. If the command takes parameters, configure them. For more information, see Configuring Parameters for Data Adapters.
3. If there are parameters, add code to set their values. For more information, see Setting and Getting Data Command Parameters.
4. Add code to open the connection associated with the data command.
5. Call the command's ExecuteNonQuery Method, setting the result to an integer.

   Note Although the method returns only a single integer value, a stored procedure you call might return values using the command's Parameters collection. For more information about working with parameters, see Setting and Getting Data Command Parameters.

6. Close the connection.

The following example shows how to create a new table in a database. The example assumes that you have already added a data command and a connection to the form or component and configured the command to use that connection. The code sets the CommandText Property to a SQL Server Create Table statement that creates a two-column table of lookup codes. The method returns -1 if the table is created successfully.

```
' Visual Basic
Dim newtablecmd As String
Dim cmdresults As Integer
newtablecmd = "CREATE TABLE LookupCodes (code_id  smallint IDENTITY(1,1) PRIMARY KEY CLUSTERE
D, code_desc  varchar(50) NOT NULL)"

' The following two property settings can also be set
' in the Properties window, but are shown here for completeness.
OleDbCommand1.CommandType = CommandType.Text
OleDbCommand1.CommandText = newtablecmd

' The connection must be open before you can execute a command.
OleDbConnection1.Open()
cmdresults = OleDbCommand1.ExecuteNonQuery()
OleDbConnection1.Close()
MessageBox.Show("After creating the table, results = " & cmdresults.ToString)
```

```
// C#
string newtablecmd;
int cmdresults;
newtablecmd = "CREATE TABLE LookupCodes (code_id  smallint IDENTITY(1,1) PRIMARY KEY CLUSTERE
D, code_desc  varchar(50) NOT NULL)";

// The following two property settings can also be done
// in the Properties window, but are shown here for completeness.
OleDbCommand1.CommandType = CommandType.Text;
OleDbCommand1.CommandText = newtablecmd;

// The connection must be open before you can execute a command.
OleDbConnection1.Open();
cmdresults = OleDbCommand1.ExecuteNonQuery();
OleDbConnection1.Close();
MessageBox.Show("After creating the table, results = " + cmdresults.ToString());
```

The following example shows how to use a data command to update a database by inserting a new record into the Authors table of the SQL Server Pubs database. In this instance, the command calls a stored procedure named "NewAuthor" that is assumed to contain an Insert Into statement with nine values for the new author record. The data command OleDbCommand2 has already been configured to have a **Parameters** collection with nine parameters in it, one each for the parameters being passed to the stored procedure. The code sets the parameters values based on text boxes in a form, opens the connection, calls the **ExecuteNonQuery** method, and then closes the connection.

```
' Visual Basic
Dim cmdresults As Integer
' The following two property settings can also be done
' in the Properties window, but are shown here for completeness.
OleDbcommand2.CommandText = "NewAuthor"
OleDbCommand2.CommandType = CommandType.StoredProcedure

' Set parameter values. In this case, all parameter values
' are strings.
OleDbCommand2.Parameters("au_id").Value = TextBox1.Text
OleDbCommand2.Parameters("au_lname").Value = TextBox2.Text
OleDbCommand2.Parameters("au_fname").Value = TextBox3.Text
OleDbCommand2.Parameters("phone").Value = TextBox4.Text
OleDbCommand2.Parameters("address").Value = TextBox5.Text
OleDbCommand2.Parameters("city").Value = TextBox6.Text
OleDbCommand2.Parameters("st").Value = TextBox7.Text
OleDbCommand2.Parameters("zip").Value = TextBox8.Text
OleDbCommand2.Parameters("contract").Value = CheckBox1.Checked

OleDbConnection2.Open()
Try
   cmdresults = OleDbcommand2.ExecuteNonQuery()
Catch ex as Exception
   MessageBox.Show("Failed to execute command")
End Try
OleDbConnection2.Close()
MessageBox.Show("Number of records inserted = " & cmdresults.ToString)

// C#
int cmdresults = 0;
// The following two property settings can also be done
// in the Properties window, but are shown here for completeness.
OleDbCommand2.CommandText = "NewAuthor";
OleDbCommand2.CommandType = CommandType.StoredProcedure;

// Set parameter values. In this case, all parameter values
// are strings.
OleDbCommand2.Parameters["au_id"].Value = TextBox1.Text;
OleDbCommand2.Parameters["au_lname"].Value = TextBox2.Text;
OleDbCommand2.Parameters["au_fname"].Value = TextBox3.Text;
OleDbCommand2.Parameters["phone"].Value = TextBox4.Text;
OleDbCommand2.Parameters["address"].Value = TextBox5.Text;
OleDbCommand2.Parameters["city"].Value = TextBox6.Text;
```

```
OleDbCommand2.Parameters["st"].Value = TextBox7.Text;
OleDbCommand2.Parameters["zip"].Value = TextBox8.Text;
OleDbCommand2.Parameters["contract"].Value = CheckBox1.Checked;

OleDbConnection2.Open();
try
{
   cmdresults = OleDbCommand2.ExecuteNonQuery();
}
catch (Exception ex)
{
   MessageBox.Show("Failed to execute command");
}
OleDbConnection2.Close();
MessageBox.Show("Number of records inserted = " + cmdresults.ToString());
```

See Also

Introduction to DataCommand Objects in Visual Studio | Working with Data Commands: High-Level Process | Setting and Getting Data Command Parameters | Executing a Data Command that Returns a Result Set | Executing a Data Command that Returns a Single Value

Executing a Data Command that Returns a Single Value

At times, you might want to execute a database command or function that returns a single value — that is, a *scalar value*. Because you are returning only one value, this type of command is typically not performed with datasets. A typical example is an SQL statement that returns an aggregate value (SUM, COUNT, and so on). Other examples are a stored procedure that takes a credit card number and returns an authorization code, or a procedure that takes a product code and returns a product name.

> **Note** If you are using a dataset, you can aggregate values of data table columns using column expressions. For more information, see Creating Dataset Columns Using Expressions.

To execute a command that returns a scalar value

1. Add a data command to your form or component. For more information, see Adding Data Commands to a Form or Component.
2. Set the data command object's CommandText property to an SQL statement or the name of a stored procedure.
3. Set the **CommandType** property to **CommandType.Text** (for a statement) or **CommandType.StoredProcedure** (for a stored procedure). These values are defined in the CommandType enumeration.
4. If the command takes parameters, set them. For more information, see Setting and Getting Data Command Parameters.
5. Open the connection associated with the data command.
6. Call the command's ExecuteScalar method, setting the result to a variable of the appropriate data type.
7. Close the connection.

The following example shows how to use a data command to execute an SQL statement that returns a scalar value. The statement queries a products table. Its takes a product ID code passed as a parameter, and returns an integer value indicating the quantity in stock for that product. In this example, there is no aggregate function (for example, SUM), because the quantity in stock is stored as a column value in the Products table.

```
' Visual Basic
Dim scalarcmd As String
Dim qtyinstock As Integer
scalarcmd = "SELECT UnitsInStock FROM Products WHERE (ProductID = ?)"
OleDbCommand3.CommandType = CommandType.Text
OleDbCommand3.CommandText = scalarcmd
OleDbCommand3.Parameters("productid").Value = txtProductID.Text
OleDbConnection1.Open()
qtyinstock = CType(OleDbCommand3.ExecuteScalar(), Integer)
OleDbConnection1.Close()
MessageBox.Show("QtyinStock = " & qtyinstock.ToString)

// C#
string scalarcmd;
int qtyinstock;
scalarcmd = "SELECT UnitsInStock FROM Products WHERE (ProductID = ?)";
OleDbCommand3.CommandType = CommandType.Text;
OleDbCommand3.CommandText = scalarcmd;
OleDbCommand3.Parameters["productid"].Value = txtProductID.Text;
OleDbConnection1.Open();
qtyinstock = (int)(OleDbCommand3.ExecuteScalar());
OleDbConnection1.Close();
MessageBox.Show("QtyinStock = " + qtyinstock.ToString());
```

See Also

Introduction to DataCommand Objects in Visual Studio | Working with Data Commands: High-Level Process | Setting and Getting Data Command Parameters | Executing a Data Command that Returns a Result Set | Executing Updates or Database Commands using a Data Command

Dataset Updates in Visual Studio .NET

When you work with datasets — disconnected stores of records residing in memory — there are a number of issues associated with updating. When data is modified in a dataset, the dataset tracks those changes. You can then send dataset changes to other processes, such as the original data source or some other component. Throughout the update process there are several raised events where data validation, concurrency evaluation, and application logic can be applied.

> **Note** In Windows Forms, the data-binding architecture takes care of sending changes from data-bound controls to the dataset, and you do not have to explicitly update the dataset with your own code. For more information, see Windows Forms Data Architecture.

The topics in this section describe how dataset updates work and how to update, insert, and delete records in a dataset as well as the underlying data source.

> **Note** When you deploy an application that includes Visual Studio data-access components, you must make sure that the user installing the application has version 2.7 or later of the Microsoft Data Access Components (MDAC). For more information, see Adding a Launch Condition for Microsoft Data Access Components.

In This Section

Introduction to Dataset Updates
: Provides an overview of how changes are made in a dataset and how the dataset tracks information about changes in order to send change information to other processes.

Updating, Inserting, and Deleting Records in a Dataset
: Provides links to procedural topics detailing how to update data in datasets.

Writing Dataset Changes to a Data Source
: Provides links to topics that describe how to update a database with the contents of a dataset.

Concurrency Control in ADO.NET
: Provides links to topics describing what concurrency is and how it is handled in ADO.NET.

Related Sections

ADO.NET Datasets
: Provides topics about what datasets are and how to use them.

Adding New Data Connections in Server Explorer
: Describes how to create a design-time connection.

Creating and Filling Datasets Example Topics
: Presents a series of specific example code topics for common tasks in creating and filling datasets with Visual Basic .NET.

Manipulating Data Example Topics
: Presents a series of specific example code topics for common tasks in working with data within your application using Visual Basic .NET.

Configuring Applications Using Dynamic Properties
: Provides conceptual information about storing property values in a configuration file that can be changed at run time, which is particularly useful for connection information such as server name and user name.

Accessing Data
: Provides links to information about working with ADO.NET in Visual Basic and Visual C#.

System.Data Namespace
: Describes the System.Data namespace and provides a table of relevant classes.

DataSet Class
: Describes the **DataSet** class and provides information and links about its members.

Concurrency Control in ADO.NET
: Provides links to topics that explain what concurrency is, as well as strategies for handling concurrency issues.

Adding a Launch Condition for Microsoft Data Access Components
: Describes how to check that the correct version of the ADO.NET data access components is available when you install your application.

Introduction to Dataset Updates

Because a dataset is effectively a cache — an in-memory copy — of data, the process of writing information to the original data source is separate from the process of modifying the data in the dataset.

> **Note** In some instances, you might not be using a dataset at all. It is possible to send updates directly to a data source (generally a database) by calling an SQL statement or stored procedure and passing it update information. For more information about communicating directly with a data source, see Introduction to DataCommand Objects in Visual Studio.

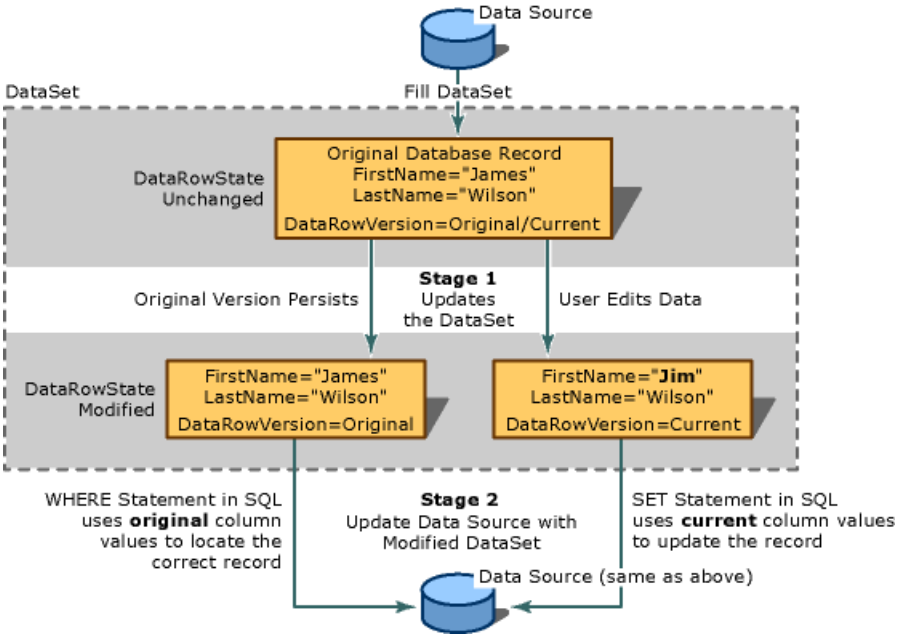
Two-Stage Updates

Updating a data source via a dataset is a two-step process. The first step is to update the dataset with new information — new records, changed records, or deleted records. If your application is concerned only with the dataset — for example, after updating the dataset, you send it to another application that will perform further processing of the dataset — then you are finished with the update.

> **Note** In Windows Forms, the data-binding architecture takes care of sending changes from data-bound controls to the dataset, and you do not have to explicitly update the dataset with your own code. For more information, see Windows Forms Data Architecture.

If you are updating a data source (such as a database), the second step is to send the changes from the dataset to the original data source. That is, the process of updating the dataset does not also write the changes through to an underlying data source; you must explicitly perform this second step. You typically accomplish this by calling the Update method of the same data adapter that you used to populate the dataset, although you can also use different adapters, for example, to move data from one data source to another or to update multiple data sources.

Two-stage update process and the role of the DataRowVersion in a successful update



For more information about transferring changes from the dataset to the database, see Database Updates from Datasets.

Structurally, a dataset makes data available as sets of collections. Datasets contain collections of tables. Tables contain collections of rows. Tables are exposed as a collection of the DataSet object and records are available in the Rows collection of DataTable objects. Making changes to data in a dataset by simply manipulating these collections using base collection methods is possible, but if you intend to update an underlying data source you must use the methods specifically designed for dataset modification.

For example, to remove a record from a data table, you could call the RemoveAt Method of the table's **Rows** collection, which physically deletes the record from the dataset. If you are using the dataset only as a structured store for data and are not concerned about transmitting change information to another application, manipulating collections this way is an acceptable way of updating a dataset.

However, if you intend to send changes to a data source or another application, you need to maintain change information (that is, metadata) about each update. Later, when you send changes to the data source or application, the process will have the information it needs to locate and update the proper records. For example, if you delete a record in the dataset, information about

the deleted record has to be maintained in the dataset so when the **DeleteCommand** of the data adapter is invoked there will be enough historic information to locate the original record in the data source so it can be deleted. For more information, see "Maintaining Information About Changes" below. For more information, see Updating, Inserting, and Deleting Records in a Dataset .

Merging Datasets

You can update the contents of a dataset by merging — that is, by copying the contents of one dataset (referred to as the *source* dataset) into the calling dataset (referred to as the *target* dataset). When you merge datasets, new records in the source dataset are added to the target dataset. Additionally, extra columns in the source dataset are added to the target dataset. Merging datasets is particularly useful when you have a local dataset and get a second dataset from another application or from a component such as an XML Web service, or just need to integrate data from multiple datasets.

When merging datasets, you can also pass an optional **Boolean** argument (*preserveChanges*) that tells the **Merge** method whether to retain existing modifications in the target dataset. Because datasets maintain multiple versions of records, it is important to keep in mind that more than one version of the records is being merged. The following table illustrates a record in two datasets that will be merged:

DataRowVersion	Target Dataset	Source Dataset
Original	James Wilson	James C. Wilson
Current	Jim Wilson	James C. Wilson

Calling the **Merge** method on the above table with `preserveChanges=false targetDataset.Merge(sourceDataset)` results in the following:

DataRowVersion	Target Dataset	Source Dataset
Original	James C. Wilson	James C. Wilson
Current	James C. Wilson	James C. Wilson

Calling the **Merge** method with `preserveChanges = true targetDataset.Merge(sourceDataset, true)` results in the following:

DataRowVersion	Target Dataset	Source Dataset
Original	James C. Wilson	James C. Wilson
Current	Jim Wilson	James C. Wilson

> **Caution** In the preserveChanges=**true** scenario, if the **RejectChanges** method is called on a record in the target dataset, it will revert to the original data from the source dataset. This means that if you went to update the original data source with the target dataset, it may not be able to find the original row to update. However, in the case of a concurrency violation (a situation where another user modified a record in the data source after the dataset was filled), filling another dataset with the updated records from the data source and then performing a merge will prevent a concurrency violation. For more information, see Concurrency Control in ADO.NET and Merging Datasets.

Update Constraints

To make changes to an existing data row, you add or update data in the individual columns. If the dataset contains constraints (such as foreign keys, or nonnullable constraints), it is possible that as you update a record — after you've finished updating one column but before you get to the next one — the record can temporarily be in an error state.

To prevent premature constraint violations you can temporarily suspend update constraints. This serves two purposes:

- It prevents an error from being thrown when you update one column before you get to another column.
- It suspends certain update events from being raised (events that are often used for validation).

After you have completed an update, you can re-enable constraint checking, which also re-enables update events and raises them.

> **Note** In Windows Forms, the data-binding architecture built into the datagrid suspends constraint checking until focus moves out of a row, and you do not have to explicitly call the **BeginEdit**, **EndEdit** or **CancelEdit** methods.

Constraints are automatically disabled when the **Merge** method is invoked on a dataset. When the merge is complete if there are any constraints on the dataset that cannot be enabled a **ConstraintException** is thrown. In this situation, the **EnforceConstraints** property is set to **false** and all constraint violations must be resolved before resetting the **EnforceConstraints** property to **true**.

After you have completed an update, you can re-enable constraint checking, which also re-enables update events and raises them.

For more information about suspending events, see Suspending Update Constraints. For more information about events, see

Data Update Events.

Dataset Update Errors

When you update a record in a dataset, there is the possibility of an error. For example, you might inadvertently write data to a column that is of the wrong data type, or too long, or that has some other integrity problem. In addition, you may have application-specific validation checks that can raise custom errors during any stage of an update event. For more information, see Data Validation in Datasets.

Maintaining Information About Changes

Information about the changes in a dataset is maintained in two ways: by flagging the row that indicates whether it has been changed (**RowState**), and by keeping multiple copies of a record (**DataRowVersion**). By using this information, processes can determine what has changed in the dataset and can send appropriate updates to the data source.

RowState Property

The RowState property of a **DataRow** object is a value that provides information about the status of a particular row of data.

The following table details the possible values of the DataRowState enumeration:

DataRowState Value	Description
Added	The row has been added as an item to a DataRowCollection. (A row in this state does not have a corresponding original version since it did not exist at the time the last **AcceptChanges** method was called).
Deleted	The row was deleted using the DataRow.Delete Method of a DataRow object.
Detached	The row has been created but is not part of any **DataRowCollection**. A **DataRow** object is in this state immediately after it has been created and before it is added to a collection, or if it has been removed from a collection.
Modified	A column value in the row has been changed in some way.
Unchanged	The row has not changed since **AcceptChanges** was last called.

DataRowVersion Enumeration

Datasets maintain multiple versions of records. The DataRowVersion enumeration of a **DataRow** object is a value that can be used to return a specific version of a **DataRow** object.

The following table details the possible values of the **DataRowVersion** enumeration:

DataRowVersion Value	Description
Current	The current version of a record contains all modifications performed on the record since the last time **AcceptChanges** has been called. If the row has been deleted there is no current version.
Default	The default value of a record, as defined by the dataset schema or data source.
Original	The original version of a record is a copy of the record as it was the last time changes were committed in the dataset. In practical terms, this is typically the version of a record as read from a data source.
Proposed	The proposed version of a record that is available temporarily, while you are in the middle of an update — that is, between the time you called the **BeginEdit** method and the **EndEdit** method. You typically access the proposed version of a record in a handler for an event such as **RowChanging**. Invoking the **CancelEdit** method reverses the changes and deletes the proposed version of the data row.

The original and current versions are useful when update information is transmitted to a data source. Typically, when an update is sent to the data source, the new information for the database is in the current version of a record, and information from the original version is used to locate the record to update. For example, in a case where the primary key of a record is changed, you must have a way to locate the proper record in the data source, in order to update the changes. If no original version existed, the record would most likely be appended to the data source resulting not only in an extra unwanted record, but in one record that is inaccurate and out of date. The two versions are also used in concurrency control; you can compare the original version against a record in the data source to determine if the record has changed since it was loaded into the dataset. For more information, see Concurrency Control in ADO.NET.

The proposed version is useful when you need to perform validation before actually committing the changes to the dataset.

Even if records have changed, there are not always original or current versions of that row. When you insert a new row into the table, there is no original version, only a current version. Similarly, if you delete a row by calling the table's **Delete** method, there is an original version, but no current version.

You can test to see if a specific version of a record exists by querying a data row's HasVersion property. You can access either version of a record by passing a DataRowVersion enumeration value as an optional argument when you request the value of a column.

Getting Changed Records

It is common that you do not update every record in a dataset. For example, a user might be working with a Windows Forms DataGrid control that displays many records. However, the user might update only a few records, delete one, and insert a new one. Datasets and data tables provide a method (**GetChanges**) for returning only the rows that have been modified.

You can create subsets of changed records using the **GetChanges** method of either the data table (DataTable.GetChanges) or of the dataset (DataSet.GetChanges) itself. If you call the method for the data table, it returns a copy of the table with only the changed records. Similarly, if you call the method on the dataset, you get a new dataset with only changed records in it. **GetChanges** by itself will return all changed records, whereas by passing the desired **DataRowState** as a parameter to the **GetChanges** method you can specify what subset of changed records you want: newly added records, records marked for deletion, detached records, and modified records.

Getting a subset of changed records is particularly useful when you want to send records to another component for processing. Instead of sending the entire dataset, you can reduce the overhead of communicating with the other component by getting only the records that the component needs. For more information, see Retrieving Changed Rows.

Committing Changes in the Dataset

As changes are made in the dataset, the **RowState** property of changed rows is set. The original and current versions of records are established and maintained and made available to you via the **RowVersion** property. The metadata stored in these properties representing the changes is needed to send the proper updates to the data source.

If the changes reflect the current state of the data source, you no longer need to maintain this information. Typically, there are two times when the dataset and its source are in synch:

- Immediately after you have loaded information into the dataset, such as when you read data from the source.
- After sending changes from the dataset to the data source, (but not before, because you would lose the change information required to send changes to the database).

You can commit the pending changes to the dataset by calling the **AcceptChanges** method. Typically, **AcceptChanges** would be called at the following times in your application.

- After you have loaded the dataset. If you load a dataset by calling a data adapter's **Fill** method, the data adapter automatically commits changes for you. However, if you load a dataset by merging another dataset into it, you have to commit the changes manually.

  > **Note** You can prevent the data adapter from automatically committing the changes when calling the **Fill** method by setting the **AcceptChangesDuringFill** property of the adapter to false. If set to false, the **RowState** of each row inserted during the fill is set to **Added**.

- After you have sent dataset changes to another process, such as an XML Web service.

  > **Caution** Committing the change this way erases any change information. Do not commit changes until after you have performed any operations in which your application depends on knowing what changes have been made in the dataset.

This method accomplishes the following:

- Writes the **Current** version of a record into its **Original** version, overwriting the original version.
- Removes any row whose **RowState** property is set to **Deleted**.
- Sets the **RowState** property of a record to **Unchanged**.

The **AcceptChanges** method is available at three levels. You can call it on a **DataRow** object, which commits changes for just that row. You can also call it on a **DataTable** object to commit all rows in a table, or on the **DataSet** object to commit all pending changes in all records of all tables of the dataset.

The following table describes which changes are committed based on what object the method is called on.

Method	Result
DataRow.AcceptChanges	Changes are committed only on the specific row
DataTable.AcceptChanges	Changes are committed on all rows in the specific table
DataSet.AcceptChanges	Changes are committed on all rows in all tables of the dataset

> **Note** If you load a dataset by calling a data adapter's **Fill** method, you do not have to explicitly accept changes; by default the **Fill** method calls the **AcceptChanges** method when it has finished populating the data table.

A related method, **RejectChanges**, undoes the effect of changes by copying the **Original** version back into the **Current** version of records and setting the **RowState** of each record back to **Unchanged**.

Data Validation

In order to verify that the data in your application meets the requirements of the processes that it is passed to, you often have to add validation. This might involve checking that a user's entry into a form is correct, validating data sent to your application by another application, or even checking that information calculated within your component falls within the constraints of your data source and application requirements.

You can validate data in several ways:

- In the presentation layer, by adding validation to forms. If you are working with Web Forms pages, you can use validation controls for this purpose. For more information, see Introduction to Validating User Input in Web Forms. If you are working with Windows Forms, you typically associate validation with TextBox or RichTextBox controls. Another way to control valid input from a user on a Windows Form is the Masked Edit Control.
- In the data back end, by sending data to the data source — for example, the database — and allowing it to accept or reject the data. If you are working with a database that has sophisticated facilities for validating data and providing error information, this can be a practical approach because you can validate the data no matter where it comes from. However, it might not accommodate application-specific validation requirements. Additionally, having the data source validate data can result in numerous round trips to the data source depending on how your application facilitates resolving validation errors raised by the back end.

  > **Security Note** When using data commands with a CommandType property set to **Text**, carefully check information that is sent from a client before passing it to your database. Malicious users might try to send (inject) modified or additional SQL statements in an effort to gain unauthorized access or damage the database. Before you transfer user input to a database, you should always verify that the information is valid; it is a best practice to always use parameterized queries or stored procedures when possible. For more information, see Scripting Exploits.

- In the business layer, by adding code to your application to validate data. A good place to perform this type of validation is in the dataset. The dataset provides some of the advantages of back-end validation — such as the ability to validate changes to the data no matter how they were made — with the ability to provide application-specific validation. For more information, see Data Validation in Datasets.

See Also

Dataset Updates in Visual Studio .NET | Introduction to DataCommand Objects in Visual Studio | Windows Forms Data Architecture | Database Updates from Datasets | Updating, Inserting, and Deleting Records in a Dataset | Merging Datasets | Concurrency Control in ADO.NET | Suspending Update Constraints | Retrieving Changed Rows

Updating, Inserting, and Deleting Records in a Dataset

After your dataset is populated with data, you will typically perform some kind of manipulation of the data before sending it back to the data source or to another process or application. Since each record in a dataset is represented by a DataRow object, changes to a dataset are accomplished by updating and deleting individual rows. Additionally, new records can be inserted into the dataset by adding new **DataRow** objects to the **DataTable** objects' Rows collection.

You can manipulate the collection directly to delete records. However, if you want to send changes to the database or to another process, the dataset has to maintain change information for deleted rows. Data tables therefore provide special methods for deleting records so this information can be retained.

> **Note** In Windows Forms, the data-binding architecture takes care of sending changes from data-bound controls to the dataset, and you do not have to explicitly update the dataset with your own code. For more information, see Windows Forms Data Architecture.

If the dataset includes constraints, you can temporarily turn constraints off before making any updates, and then turn them back on afterward. For more information, see Suspending Update Constraints.

In This Section

Updating Existing Records in a Dataset
: Describes how to access and change values to individual records.

Inserting New Records into a Dataset
: Describes how to create a new **DataRow** object and assign values to the individual columns it contains.

Deleting Records in a Dataset
: Describes the proper way to remove records from a dataset while maintaining information needed to locate and remove the record fom the data source.

Data Update Events
: Describes the events that are raised in a dataset when you make changes to records.

Suspending Update Constraints
: Describes how to temporarily turn off constraint checking while making changes to individual columns in a record.

Merging Datasets
: Describes how to copy records from one dataset to another.

Committing Changes in a Dataset
: Describes how to reset change information in a dataset so that all its records are flagged as being unchanged.

Identifying and Returning Changed Rows
: Provides links to procedural topics that detail how to determine whether changes were made to individual records, and how to find and retrieve rows with specific changes.

Validating Data
: Provides links to topics that explain and describe dataset events that can be used for validating data.

Related Sections

ADO.NET Datasets
: Provides topics about what datasets are and how to use them.

Introduction to Dataset Updates
: Provides an overview of how changes are made in a dataset and how the dataset tracks information about changes in order to send change information to other processes.

Data Update Events
: Describes the events that are raised in a dataset when you make changes to records.

Suspending Update Constraints
: Describes how to temporarily turn off constraint checking while making changes to individual columns in a record.

Database Updates from Datasets
: Provides an overview of how to move dataset changes to a database.

Creating and Filling Datasets Example Topics
: Presents a series of specific example code topics for common tasks in creating and filling datasets with Visual Basic .NET.

Manipulating Data Example Topics
: Presents a series of specific example code topics for common tasks in working with data within your application using Visual Basic .NET.

Updating Existing Records in a Dataset

In order to edit an existing record in a dataset, you need to access a particular column of data in a particular row. The data in both typed and untyped datasets can be accessed via the following:

- With the indices of the tables, rows, and columns collections.
- By passing the table and column names as strings to their respective collections.

Although typed datasets can use the same syntax as untyped datasets, there are additional advantages to using typed datasets. For more information, see the "To update existing records using typed datasets" section below.

To update existing records in either typed or untyped datasets

- Assign a value to a specific column within a DataRow object.

  The table and column names of untyped datasets are not available at design time and must be accessed through their respective indices.

  The following example shows how to update the data in the first two columns of the fifth record of the first table in `dataset1`, accessing the data values with the tables, rows, and items (columns) collection indices:

  ```
  ' Visual Basic
  dataset1.Tables(0).Rows(4).Item(0) = "Wingtip Toys"
  dataset1.Tables(0).Rows(4).Item(1) = "Buffalo"

  // C#
  dataset1.Tables[0].Rows[4].ItemArray[0] = "Wingtip Toys";
  dataset1.Tables[0].Rows[4].ItemArray[1] = "Buffalo";
  ```

  The following example shows how to update the same data as the preceding example, replacing the collection indices with table, and column names passed as strings:

  ```
  ' Visual Basic
  dataset1.Tables("Customers").Rows(4).Item("CompanyName") = "Wingtip Toys"
  dataset1.Tables("Customers").Rows(4).Item("City") = "Buffalo"

  // C#
  dataset1.Tables["Customers"].Rows[4]["CompanyName"] = "Wingtip Toys";
  dataset1.Tables["Customers"].Rows[4]["City"] = "Buffalo";
  ```

To update existing records using typed datasets

- Assign a value to the specific column within a **DataRow** object.

  Typed datasets utilize early binding, which exposes the table and column names as properties at design time. This results in code that is both easier to read and write.

  The following example shows how to update the data in the `CompanyName` and `City` columns of the fifth record in the `Customers` table in the `DsCustomers1` dataset:

  ```
  ' Visual Basic
  DsCustomers1.Customers(4).CompanyName = "Wingtip Toys"
  DsCustomers1.Customers(4).City = "Buffalo"

  // C#
  dsCustomers1.Customers[4].CompanyName = "Wingtip Toys";
  dsCustomers1.Customers[4].City = "Buffalo";
  ```

See Also

Inserting New Records into a Dataset

In order to add new records into a dataset, a new data row must be created and added to the DataRow collection of a data table. The following procedure details how to insert additional rows into a DataTable object in a dataset. For this example, it is assumed the `ExistingTable` is in a dataset and has two columns named `FirstName` and `LastName`.

To add new records to a typed or untyped dataset

1. Call a data table's NewRow method to create a new, blank record.

   This new record inherits its column structure from the data table's DataColumnCollection.

   ```
   ' Visual Basic
   Dim anyRow as DataRow = ExistingTable.NewRow

   // C#
   DataRow anyRow = ExistingTable.NewRow();
   ```

2. Update the new row as if it were an existing record.

   ```
   ' Visual Basic
   anyRow(0) = "Jay"
   anyRow(1) = "Stevens"
   ' or
   anyRow("FirstName") = "Jay"
   anyRow("LastName") = "Stevens"

   // C#
   anyRow[0] = "Jay";
   anyRow[1] = "Stevens";
   // or
   anyRow["FirstName"] = "Jay";
   anyRow["LastName"] = "Stevens";
   ```

3. Add the new record to the table by calling the Add method of the **DataRowCollection** object.

   ```
   ' Visual Basic
   ExistingTable.Rows.Add(anyRow)

   // C#
   ExistingTable.Rows.Add(anyRow);
   ```

Inserting New Records with Typed Datasets

Typed datasets expose the column names as properties of the **DataRow** object.

To add new records using typed datasets

- The following example illustrates the same three steps above, except this time the code is modified for use with a typed dataset:

   ```
   ' Visual Basic
   Dim anyRow as DataRow = DatasetName.ExistingTable.NewRow
   anyRow.FirstName = "Jay"
   anyRow.LastName = "Stevens"
   ExistingTable.Rows.Add(anyRow)

   // C#
   DataRow anyRow = DatasetName.ExistingTable.NewRow();
   anyRow.FirstName = "Jay";
   ```

```
anyRow.LastName = "Stevens";
ExistingTable.Rows.Add(anyRow);
```

See Also

Editing Data in a Table | Adding Data to a Table | Deleting a Row from a Table | NewRow | Add |
Updating, Inserting, and Deleting Records in a Dataset |
Code: Creating a New DataRow Based on an Existing DataTable (Visual Basic)

Deleting Records in a Dataset

In order to retain the information that the dataset needs to properly update the data source, use the **Delete** method to remove rows in a data table. If your application does not need to send updates back to a data source then it is possible to remove records by directly accesing the data row collection.

To delete records from a data table

- Call the data row's Delete method.

  This method does not physically remove the record; instead, it marks the record for deletion.

  > **Note** If you get the count property of a **DataRowCollection**, the resulting count includes records that have been marked for deletion. To get an accurate count of only records that are not marked for deletion, you can loop through the collection looking at the RowState property of each record. Alternatively, you can create a data view of a dataset that filters based on row state and get the count property from there.

  The following example shows how to call the **Delete** method to mark the first row in the Customers table as deleted:

```
' Visual Basic
DsCustomers1.Customers.Rows(0).Delete()

// C#
dsCustomers1.Customers.Rows[0].Delete();
```

See Also

Editing Data in a Table | Adding Data to a Table | Deleting a Row from a Table | Delete | Updating, Inserting, and Deleting Records in a Dataset

Suspending Update Constraints

If a dataset contains constraints (such as a foreign-key constraint), updating a column in a record can violate the constraint: After you have finished updating one column but before you get to the next one, the record can temporarily be in an error state. For example, imagine that you have a data table in your dataset with two columns that do not allow null values. As soon as you start to write a new record there will always be a null value in at least one of the columns (you can only write one column at a time). If there were no mechanism to allow temporary constraint suspension, an error would be raised every time after writing the first column and before writing the second.

To suspend update constraints

1. Before changing data in a data row, call the **DataRow** object's BeginEdit method.

   Note Changes you make to the row after calling **BeginEdit** are saved.

2. Make the updates to that row.
3. Call the EndEdit method to commit changes to the row and re-enable constraint checking.

   The RowChanging event is raised.

4. Call the row's CancelEdit method to discard changes to the row.

   The following example shows a simple update that is bracketed with **BeginEdit** and **EndEdit** method calls. This example assumes that you have a table called `Customers` in a typed dataset called `DsCustomers1`.

```
' Visual Basic
DsCustomers1.Customers(4).BeginEdit()
DsCustomers1.Customers(4).CompanyName = "Wingtip Toys"
DsCustomers1.Customers(4).City = "Buffalo"
DsCustomers1.Customers(4).EndEdit()

// C#
dsCustomers1.Customers[4].BeginEdit();
dsCustomers1.Customers[4].CompanyName = "Wingtip Toys";
dsCustomers1.Customers[4].City = "Buffalo";
dsCustomers1.Customers[4].EndEdit();
```

See Also

Introduction to Dataset Updates | Updating, Inserting, and Deleting Records in a Dataset | Data Update Events | BeginEdit | EndEdit | CancelEdit | RowChanging Event | Code: Turning Off Constraints While Filling a DataSet (Visual Basic)

Merging Datasets

You can merge the contents of one dataset (referred to as the *source* dataset) into another dataset (referred to as the *target* dataset). Some typical examples of when you would perform a merge are:

- You have a main dataset containing your working data, and you get a second dataset containing updates or records to be added to it. This is a typical scenario when you are working with an XML Web service that returns a dataset to you as the result of a method call.
- Your application gets a dataset from two different components but needs to work with the data in a single dataset. For example, you might get company information from one component and stock price information from another.

To copy information from one dataset to another, you use a dataset's Merge method. You can specify merge options to achieve a variety of goals:

- Add new records to one dataset by copying them from another dataset.
- Update the state of records in one dataset based on another. In this situation, the second dataset contains information not only about new records, but also changed records and deleted records, and you want your first dataset to reflect all these changes.
- Copy not only new records and changes, but specify what to do if the datasets do not have the same schemas.

To merge one dataset into another

- Call the Merge method of the target dataset, passing it the source dataset containing the records to be merged.

  The following example shows how to merge two datasets with the default setting of **PreserveChanges** = **false**:

  ```
  ' Visual Basic
  targetDataset.Merge(sourceDataset)

  // C#
  targetDataset.Merge(sourceDataset);
  ```

  The following example shows how to merge two datasets with the **PreserveChanges** argument set to **true**:

  ```
  ' Visual Basic
  targetDataset.Merge(sourceDataset, True)

  // C#
  targetDataset.Merge(sourceDataset, true);
  ```

See Also

Dataset Updates in Visual Studio .NET | Merge | DataRowVersion Enumeration

Committing Changes in a Dataset

As you make changes to records in a dataset by updating, inserting, and deleting records, the dataset maintains original and current versions of the records. In addition, each row's **RowState** property is set to indicate whether the records are in their original state or if they have been updated, inserted, or deleted. This information is useful when you need to find a particular version of a row. Typically, you would get a subset of all changed records to send to another process. For more information, see Retrieving Changed Rows. After you have processed all the changed rows, you can commit the changes by calling the **AcceptChanges** method of the dataset.

To commit changes in a dataset

- Call the dataset's AcceptChanges method.

  The following example shows how to call the **AcceptChanges** method to commit changes in the `Customers` table after updating a data source:

```
' Visual Basic
OleDbDataAdapter1.Update(Dataset1, "Customers")
Dataset1.Customers.AcceptChanges()

// C#
OleDbDataAdapter1.Update(Dataset1, "Customers");
Dataset1.Customers.AcceptChanges();
```

Copying Information from Another Dataset Before Committing Changes

To copy information from another dataset and then commit the changes

1. Call the dataset's Merge method.
2. Call the dataset's AcceptChanges method.

   The following example shows how to commit changes in a dataset after using the **Merge** method to copy information from a second dataset:

```
' Visual Basic
Dataset1.Merge(Dataset2)
Dataset1.AcceptChanges()

// C#
Dataset1.Merge(Dataset2);
Dataset1.AcceptChanges();
```

See Also

Dataset Updates in Visual Studio .NET | Retrieving Changed Rows | DataSet.AcceptChanges Method | Data Update Events

Identifying and Returning Changed Rows

Datasets maintain multiple versions of data rows in order to locate the original records in a data source. Before performing an update to the data source, you may want to examine specific rows. The topics in this section provide details on determining if records have changed, as well as retrieving particular versions of records .

In This Section

Checking for Changed Rows
Explains how to determine whether dataset records have changed and how to determine whether a record is new, changed, or deleted.
Retrieving Changed Rows
Describes how to extract only changed records from a dataset.
Locating an Individual Row in a Dataset
Describes how to find a specific data row by searching for a primary key value.
Getting Specific Versions of a Row
Describes how to return a record by passing a **DataRowVersion** enumeration value.
Locating Rows that Have Errors
Describes how to determine if errors exist in a dataset and the method used to find rows that have errors.

Related Sections

Updating, Inserting, and Deleting Records in a Dataset
Provides links to topics about updating ADO.NET data.
ADO.NET Datasets
Provides links to topics about understanding and working with datasets.
Creating and Filling Datasets Example Topics
Presents a series of specific example code topics for common tasks in creating and filling datasets with Visual Basic .NET.
Manipulating Data Example Topics
Presents a series of specific example code topics for common tasks in working with data within your application using Visual Basic .NET.

Checking for Changed Rows

When changes are made to records in a dataset, information about those changes is stored until you commit them (calling the **AcceptChanges** method of a dataset, data table, or data row commits the changes).

Changes are tracked in each data row two ways:

- Each record retains the type of change that has been made to it through the **DataRow** object's **RowState** property.
- The dataset maintains multiple copies of changed rows — the original (before changes) and current (after changes) version — that you can access. During the short period when a change is pending (during the time that you can respond to the **RowChanging** event), a third version — the proposed version — is available as well. For more information, see Introduction to Dataset Updates.

Determining If There Are Changed Rows

The **HasChanges** method of a dataset returns **true** if changes have been made in the dataset.

To determine if any changes have been made to any rows

- Call the HasChanges method of a dataset to check for changed rows.

  The following example shows how to check the return value from the **HasChanges** method to detect whether there are any changed rows in a dataset named `myDataset`.

```
' Visual Basic
If myDataset.HasChanges() Then
     ' Changed rows were detected, add appropriate code.
Else
     ' NO changed rows were detected, add appropriate code.
End If

// C#
if (myDataSet.HasChanges())
{
   // Changed rows were detected, add appropriate code.
}
else
{
   // NO changed rows were detected, add appropriate code.
}
```

Determining the Type of Changes

You can also check to see what type of changes have been made in a dataset by passing a value from the DataRowState Enumeration to the **HasChanges** method.

To determine what type of changes have been made to a row

- Pass a **DataRowState** value to the **HasChanges** method.

The following example shows how to check a dataset named `myDataset` to determine if there have been any new rows added to it:

```
' Visual Basic
If myDataset.HasChanges(DataRowState.Added) Then
   ' New rows have been added to the dataset, add appropriate code.
Else
   ' No new rows have been added to the dataset, add appropriate code.
End If

// C#
if (myDataSet.HasChanges(DataRowState.Added))
{
```

```
    // Changed rows were detected, add appropriate code.
}
else
{
    // NO changed rows were detected, add appropriate code.
}
```

See Also

Introduction to Dataset Updates | DataSet.HasChanges Method | DataRowState Enumeration |
Code: Locating Changed Data in a DataSet (Visual Basic)

Retrieving Changed Rows

You often want to extract from a dataset only the records that have been changed. This is often true, for example, when your application has processed some records in a dataset and needs to send only the changed records back to the data source or to another process.

> **Note** After you commit changes to a dataset or data table (by calling the **AcceptChanges** method), all rows are set to unchanged and the information needed by the **GetChanges** method no longer exists. If your application needs to process changed rows, you must do so before calling the **AcceptChanges** method.

Calling the GetChanges method of a dataset or data table returns a new dataset or data table that contains only records that have been changed. If you want to get only specific records — for example, only new records or only changed records — you can pass as a parameter a row state value defined using the DataRowState enumeration.

To get all changed records from a dataset or data table

- Call the GetChanges method of a dataset or data table.

  The following example creates a new dataset called `myChangedRecords` and populates it with all the changed records from another dataset called `myDataSet`.

```
' Visual Basic
Dim myChangedRecords as DataSet
myChangedRecords = myDataSet.GetChanges()

// C#
DataSet myChangedRecords = myDataSet.GetChanges();
```

To get changed records that are of a specific RowState

- Call the **GetChanges** method of a dataset or data table and pass a **DataRowState** value as an argument.

  The following example shows how to create a new dataset called `myChangedRecords` and populate it with only records that have been added to the `myDataSet` dataset.

```
' Visual Basic
Dim myChangedRecords as DataSet
myChangedRecords = myDataSet.GetChanges(DataRowState.Added)

// C#
DataSet myChangedRecords = myDataSet.GetChanges(DataRowState.Added);
```

See Also

Dataset Updates in Visual Studio .NET | DataSet.GetChanges Method | DataRowState Enumeration | DataRowVersion Enumeration | Code: Locating Changed Data in a DataSet (Visual Basic)

Locating an Individual Row in a Dataset

Most applications that consume data need to access specific records that satisfy some kind of criteria. In order to find a particular row in a dataset you invoke the Find Method of the **DataRowCollection**. If the primary key exists, a **DataRow** object is returned. If the primary key cannot be found, a null value is returned.

To find a specific row within a dataset

- Call the **Find** method of the **Rows** collection passing the primary key as an argument.

  The following example shows how to declare a new row (`rowFoundRow`) and assign it the return value of the **Find** method. If the primary key is found the contents of column index one are displayed in a message box.

```
' Visual Basic
Dim n As String = "primaryKeyValue to be found"
Dim rowFoundRow as DataRow = anyTable.Rows.Find(n)
If Not (rowFoundRow Is Nothing) Then
   MessageBox.Show(CType(rowFoundRow(1), String))
Else
   MessageBox.Show("A row with the primary key of " & _
      n & " could not be found")
End If

// C #
string n = "primaryKeyValue to be found";
DataRow rowFoundRow = anyTable.Rows.Find(n);
if (rowFoundRow != null)
{
   MessageBox.Show((string)rowFoundRow[1]);
}
else
{
MessageBox.Show("A row with the primary key of " +
   n + " could not be found");
}
```

See Also

Updating, Inserting, and Deleting Records in a Dataset | DataRowCollection.Find Method | DataTable.Select Method | Code: Locating Changed Data in a DataSet (Visual Basic)

Getting Specific Versions of a Row

Particular versions of records are available before changes are committed to that data row. For example, before calling the **AcceptChanges** method, your application can access the different versions of a record (as defined in the **DataRowVersion** enumeration) and process the changes accordingly.

> **Note** Different versions of a row exist only after a row has been edited, and before the row has had the **AcceptChanges** method called. After the **AcceptChanges** method has been called, the current and original versions are the same.

Passing the **DataRowVersion** value along with the column index (or column name as a string) returns the value from that column's particular row version. The changed column is identified during the **ColumnChanging** and **ColumnChanged** events, so that is a good time to inspect the differing row versions for validation purposes. However, if you have temporarily suspended constraints, those events will not be raised and you will need to programmatically identify which columns have changed. You can do this by iterating through the **Columns** collection and comparing the different **DataRowVersion** values.

To get a specific version of a record

- Access the value of a column and add a parameter to the index indicating which version of a row you want to return. Possible values for the row version are defined in the DataRowVersion enumeration.

  The following example shows how you can use a **DataRowVersion** value to get the original value of the `CompanyName` field:

```
' Visual Basic
Dim sOrigCompName As String
sOrigCompName = CType(DsCustomers1.Customers(0)("CompanyName", _
DataRowVersion.Original), String)

// C#
string origCompName;
origCompName = (string)
(dsCustomers.Customers[0]["CompanyName", DataRowVersion.Original]);
```

See Also

Introduction to Dataset Updates | DataRowState Enumeration | DataRowVersion Enumeration

Locating Rows that Have Errors

If your application does not use the **RowUpdated** or **ColumnUpdated** event handlers to catch errors during the update process, you will need to find the error programmatically.

To locate a row that has errors as a result of an update attempt

1. Set the **HasErrors** property to locate in the dataset the row that has errors.
2. If you do not find the error in the dataset, iterate through the collections of tables, then the rows to find the row with the error.

```
' Visual Basic
Private Sub finderr2()
        Dim dt As DataTable
        Dim dr As DataRow
        For Each dt In anyDataset.Tables
            For Each dr In dt.Rows
                If dr.HasErrors Then
                    ' Row found! Add code to reconcile error.
                    dr.ClearErrors() ' Clear the error.
                    ' Re-attempt the update.
                End If
            Next
        Next
    End Sub

// C#
private void Find_Errors2()
{
   DataSet anyDataset = null;
   foreach (DataTable dt in anyDataset.Tables)
   {
      foreach (DataRow dr in dt.Rows)
      {
         if (dr.HasErrors)
         {
            // Row found! Add code to reconcile error.
            dr.ClearErrors();
            // Re-attempt the update.
         }
      }
   }
}
```

See Also

Dataset Updates in Visual Studio .NET | Writing Event Handlers (Visual Basic) | UpdateStatus Enumeration | DataSet.HasErrors Property

Validating Data

Datasets provide specific events that are raised as values in data rows change. Event handlers can be created to validate these changes based on your particular application requirements. The following topics detail these events and provide an explanation of the information passed to each event.

In This Section

Data Validation in Datasets
Provides an overview of events that can be used to analyze data before being committed to the dataset.
Validating Data During Column Changes
Describes the information returned during the **ColumnChanging** event.
Validating Data During Row Changes
Describes the information returned during the **RowChanging** event.

Related Sections

Updating, Inserting, and Deleting Records in a Dataset
Provides links to topics about updating ADO.NET data.
DataTable.ColumnChanging Event
Provides syntax and other reference information.
DataTable.RowChanging Event
Provides syntax and other reference information.

Data Validation in Datasets

You can confirm that data being written to a dataset is valid by building validation checks into the dataset itself. The dataset can check the data no matter how the update is being performed — whether directly by controls in a form, within a component, or in some other way. Because the dataset is part of your application, it is a logical place to build application-specific validation (unlike building the same checks into the database backend). Validation within a dataset can be accomplished:

- By creating keys, unique constraints, and so on as part of the actual schema definition of the dataset. For more information on incorporating validation into the schema definition, see Creating Primary and Unique Keys in XML Schemas.
- By creating your own application-specific validation that can check data during column and row changing events.
- By setting the DataColumn object's properties, such as **MaxLength**, **AllowDBNull**, and **Unique**.

There are several events that are raised by the **DataTable** object when a change is occurring in a record:

- The ColumnChanging and ColumnChanged events are raised during and after each change to an individual column. The **ColumnChanging** event is useful when you want to validate changes in specific columns. Information about the proposed change is passed as an argument with the event. For more information see Validating Data During Column Changes.
- The RowChanging and RowChanged events are raised during and after any change in a row. The **RowChanging** event is more general, in that it simply indicates that a change is occurring somewhere in the row; you do not know which column has changed. For more information see Validating Data During Row Changes.

By default, each change to a column therefore raises four events: first the **ColumnChanging** and **ColumnChanged** events for the specific column being changed, and then the **RowChanging** and **RowChanged** event. If multiple changes are being made to the row, the events will be raised for each change.

> **Note** The data row's **BeginEdit** method turns off the **RowChanging** and **RowChanged** events after each individual column change. In that case, the event is not raised until the **EndEdit** method has been called, when the **RowChanging** and **RowChanged** events are raised just once. For more information, see Suspending Update Constraints.

The event you choose depends on how granular you want the validation to be. If it is important that you catch an error immediately when a column is changed, build validation using the **ColumnChanging** event. Otherwise, use the **RowChanging** event, which might result in catching several errors at once. Additionally, if your data is structured in such a way that the value of one column is validated based on the contents of another column then you should perform your validation during the **RowChanging** event.

See Also

Dataset Updates in Visual Studio .NET | Introduction to Validating User Input in Web Forms | Validating Data During Column Changes | Validating Data During Row Changes | DataTable.ColumnChanging Event | DataTable.RowChanging Event | DataColumn Properties

Validating Data During Column Changes

You can validate data when the column changes by responding to the **ColumnChanging** event. This event passes an event argument object with properties that give you access to the column being changed. Validation can also be performed during the **RowChanging** event, For more information see Validating Data During Row Changes.

To validate when a column changes

1. Create an event handler for the ColumnChanging event of a data table. For more information, see Writing Event Handlers (Visual Basic) or Events Tutorial (C#).
2. In the event handler, you can return the proposed and original values by examining the **ProposedValue** and **Row** properties:

```
' Visual Basic
Dim newValue As String = CType(e.ProposedValue, String)
Dim origvalue As String = CType(e.Row(e.Column), String)

// C#
string newValue = (string)e.ProposedValue;
string origValue = (string)(e.Row[e.Column]);
```

3. To get information about the column that is changing, examine the **Column** property that is passed to the handler:

```
' Visual Basic
Dim colDataType As String = e.Column.DataType.ToString()
Dim colName As String = e.Column.ColumnName

// C#
string colDataType = e.Column.DataType.ToString();
string colName = e.Column.ColumnName;
```

4. To reject the change, throw an exception.

```
' Visual Basic
' Throw the exception within the ColumnChanging event handler –
' in this case we will validate for a value that is < 10.

Private Sub dt_ColumnChanging(ByVal sender As Object, ByVal e As System.Data.DataColumnCh
angeEventArgs) Handles dt.ColumnChanging
   Dim newvalue As Integer = CType(e.ProposedValue, Integer)
   If newvalue < 10 Then
      MessageBox.Show(newvalue.ToString() & " is not less than 10")
      Throw New Exception(newvalue.ToString() & " is not less than 10")
   End If
End Sub
// C#
private void dt_ColumnChanging(object sender, System.Data.DataColumnChangeEventArgs e)
{
   int newvalue = (int)(e.ProposedValue);
   if (newvalue < 10)
   {
      MessageBox.Show(newvalue.ToString() + " is not less than 10");
      throw new Exception(newvalue.ToString() +
         " is not less than 10");
   }
}
```

See Also

Dataset Updates in Visual Studio .NET | Introduction to Validating User Input in Web Forms | Validating Data During Row Changes | DataTable.ColumnChanging Event | DataTable.RowChanging Event | DataColumn Properties | Data Validation in Datasets | Writing Event Handlers (Visual Basic) | Events Tutorial (C#)

Validating Data During Row Changes

You can validate data when the row changes by responding to the **RowChanging** event. Typed dataset classes expose a table-specific **RowChanging** event. For example, if the typed dataset contains a table called Employees, the **Employees** data table object raises a **EmployeesRowChanging** event. This event passes an event argument object that includes strongly typed properties to give you access to the column being changed. Validation can also be performed during the **ColumnChanging** event, For more information see Validating Data During Column Changes.

To validate when a row changes

1. Create an event handler for the RowChanging event of a data table. For more information see Writing Event Handlers (Visual Basic) or Events Tutorial (C#).

   If you are working with a typed dataset, you can respond to the table-specific event, such as **EmployeesRowChanging** or you can respond to the standard **RowChanging** event. However, both events are raised so performing the same validation in each event is unnecessary.

2. In the event handler, get changes to the row by extracting individual columns from the **Proposed** version of rows using syntax such as the following:

```
' Visual Basic
newFName = CType(e.Row("FirstName", DataRowVersion.Proposed), String)
// C#
newFName = (string)(e.Row["FirstName", DataRowVersion.Proposed]);
```

   Note If you are working with the table-specific event, the proposed versions of the columns are available as typed properties of the row, and you can access them using syntax such as the following:
   newFName = e.Row.FirstName
   .

3. Get the existing versions of columns using syntax such as the following:

```
' Visual Basic
oldFName = CType(e.Row("FirstName", DataRowVersion.Original), String)
// C#
oldFName = (string)(e.Row["FirstName", DataRowVersion.Original]);
```

4. Perform the validation. To reject the change, throw an exception. To accept the change, you do not need to perform any special logic.

   The following example shows how you can validate data in a row-changing event. The code checks that if a value already exists for the column `SecurityCode`, the new value cannot be null — that is, it prevents the column value from being deleted. In the event handler, the code makes sure that there is an existing value by using the data row's **HasVersion** property.

```
' Visual Basic
Private Sub employeetable_EmployeesRowChanging(ByVal sender As _
System.Object, ByVal e As dsEmployees.EmployeesRowChangeEvent) _
Handles employeetable.EmployeesRowChanging
   Dim original As String = ""
   Dim proposed As String = ""
   If e.Row.HasVersion(DataRowVersion.Original) Then
      original = _
         CType(e.Row("SecurityCode", DataRowVersion.Original), String)
   Else
      original = ""
   End If
   proposed = e.Row.SecurityCode

   If original <> "" Then
```

```
      If proposed = "" Then
         Throw (New Exception("Security Code cannot be blank"))
      End If
   End If
End Sub

// C#
private void employeetable_EmployeesRowChanging(object sender, dsEmployees.EmployeesRowCh
angeEvent e)
{
   string original = "";
   string proposed = "";
   if (e.Row.HasVersion(DataRowVersion.Original))
   {
      original =
      (string)(e.Row["SecurityCode", DataRowVersion.Original]);
   }
   else
   {
      original = "";
   }

   proposed = e.Row.SecurityCode;
   if (original != "")
   {
      if (proposed == "")
      {
         throw (new System.Exception("Security Code cannot be blank"));
      }
   }
}
```

See Also

Dataset Updates in Visual Studio .NET | Introduction to Validating User Input in Web Forms | Validating Data During Column Changes | DataTable.ColumnChanging Event | DataTable.RowChanging Event | DataColumn Properties | Data Validation in Datasets | Writing Event Handlers (Visual Basic) | Events Tutorial (C#)

Writing Dataset Changes to a Data Source

After the contents of a dataset are modified and validated, most applications will typically need to write the changes through to an underlying database. For more information about modifying datasets, see Updating, Inserting, and Deleting Records in a Dataset. For more information about validating datasets, see Validating Data.

The topics in this section describe some important dataset fundamentals regarding data source updating, as well as details on performing an update and handling potential errors.

In This Section

Database Updates from Datasets
Provides an overview of how to move dataset changes to a database.
Updating a Data Source with a Dataset
Describes the process of writing changes to a data source.
Responding to Database Update Errors
Describes how to determine if errors occur while writing changes to a dataset.

Related Sections

Dataset Updates in Visual Studio .NET
Provides links to topics about updates and concurrency control.
ADO.NET Datasets
Provides links to topics about understanding and working with datasets.
Creating and Filling Datasets Example Topics
Presents a series of specific example code topics for common tasks in creating and filling datasets with Visual Basic .NET.
Manipulating Data Example Topics
Presents a series of specific example code topics for common tasks in working with data within your application using Visual Basic .NET.

Updating a Data Source with a Dataset

After your dataset has been modified and validated, it is likely that your application will want to send the new data back to a data source. In order to update a data source with the contents of a dataset, you call the Update method of a data adapter. The data adapter will execute the appropriate command (INSERT, UPDATE, or DELETE) based on the **RowState** of the row in the dataset.

> **Note** Because attempting to update a data source with the contents of a dataset can result in errors, you should invoke the data adapters **Update** method from within a **Try...Catch** block.

The exact procedure to update a data source can vary depending on your business needs, but your application should include the following steps:

1. Invoke the data adapters update method within a **Try...Catch** block.
2. If an exception is caught, locate the data row that caused the error. For more information, see Responding to Database Update Errors.
3. Reconcile the problem in the data row (programmatically if possible, or by presenting the erroneous row to the user for modification), and then reattempt the update.

To update a data source with a dataset

- Call the **Update** method of a data adapter, passing the name of the dataset that contains the values to be written to the data source.

  The following example shows how to attempt an update to a data source from within a **Try...Catch** block with the contents of `myDataset`.

```
' Visual Basic
Try
   oleDbDataadapter1.Update(myDataset)
Catch x As Exception
   ' Error during Update, add code to locate error, reconcile
   ' and try to update again.
End Try

// C#
try
{
   oleDbDataAdapter1.Update(myDataset);
}
catch (Exception e)
{
   // Error during Update, add code to locate error, reconcile
   // and try to update again.
}
```

Updating Related Tables in a Dataset

When updating related tables in a dataset, it is important to update in the proper sequence to reduce the chance of violating referential integrity constraints. The order of command execution will also follow the indices of the DataRowCollection in the dataset. To prevent data integrity errors from being raised, the best practice is to update the data source in the following sequence:

1. Child table: delete records.
2. Parent table: insert, update, and delete records.
3. Child table: insert and update records.

To update a data source with a dataset that contains two or more related tables

- Call the **Update** method of each data adapter.

  The following example shows how to update a data source with a dataset that contains related tables. In order to follow the

above sequence, three temporary DataTables will be created to hold the differing records. Then the **Update** method will be called for each subset of rows from within a **Try...Catch** block. If update errors occur, the suggested course of action is to branch off and resolve them. Then the dataset commits the changes. Finally, dispose of the temporary datatables to release the resources.

```
' Visual Basic
Private Sub UpdateDB()
    Dim DeletedChildRecords As DataTable = _
        DsNorthwind1.Orders.GetChanges(DataRowState.Deleted)
    Dim NewChildRecords As DataTable = _
        DsNorthwind1.Orders.GetChanges(DataRowState.Added)
    Dim ModifiedChildRecords As DataTable = _
        DsNorthwind1.Orders.GetChanges(DataRowState.Modified)
    Try
        If Not DeletedChildRecords Is Nothing Then
            daOrders.Update(DeletedChildRecords)
            DeletedChildRecords.Dispose()
        End If
        daCustomers.Update(DsNorthwind1, "Customers")
        If Not NewChildRecords Is Nothing Then
            daOrders.Update(NewChildRecords)
            NewChildRecords.Dispose()
        End If
        If Not ModifiedChildRecords Is Nothing Then
            daOrders.Update(ModifiedChildRecords)
            ModifiedChildRecords.Dispose()
        End If
        DsNorthwind1.AcceptChanges()
    Catch ex As Exception
        ' Update error, resolve and try again
    End Try
End Sub

// C#
void UpdateDB()
{
    DataTable DeletedChildRecords = dsNorthwind1.Orders.GetChanges(DataRowState.Deleted);
    DataTable NewChildRecords = dsNorthwind1.Orders.GetChanges(DataRowState.Added);
    DataTable ModifiedChildRecords = dsNorthwind1.Orders.GetChanges(DataRowState.Modified
);
    try
    {
        if (DeletedChildRecords != null)
        {
            daOrders.Update(DeletedChildRecords);
            DeletedChildRecords.Dispose();
        }
        if (NewChildRecords != null)
        {
            daOrders.Update(NewChildRecords);
            NewChildRecords.Dispose();
        }
        if (ModifiedChildRecords != null)
        {
            daOrders.Update(ModiifiedChildRecords);
            ModifiedChildRecords.Dispose();
        }
        DsNorthwind1.AcceptChanges();
    }
```

```
    catch (Exception ex)
    {
        // Update error, resolve and try again
    }
}
```

See Also

Dataset Updates in Visual Studio .NET | Introduction to Dataset Updates | DataAdapter.Update Method | Responding to Database Update Errors | DataSet.GetChanges Method | Code: Updating a Data Source with the Contents of Related DataTables (Visual Basic)

Responding to Database Update Errors

Errors may arise when attempting to update data. A typical error is that the record in the database has changed since it was read into your dataset. Regardless of what caused the error, you must create logic to handle the reconciliation of the record.

In order to resolve the row that raised the error, your application should locate the data row that caused the error, reconcile the problem, clear the error, and re-attempt the update. For more information, see Locating Rows that Have Errors.

When an update command has been executed, the data adapters **RowUpdated** event is raised for each record affected. By writing an event handler for this event, you can determine whether the update was successful. You can also set the **Status** property to determine whether the Update command should try to continue processing additional changes, or abort after an error is raised.

To respond to an update error

1. Create an event handler for the **RowUpdated** event. For more information, see Writing Event Handlers.
2. Check the **Status** property of the **RowUpdated** event.

   Note If there is an error, the value will be **ErrorsOccurred**.

3. If you want to control whether the update continues or not, set the **Status** property to a valid value as defined in the UpdateStatus Enumeration.You can also set the **IgnoreUpdateErrors** property of the data adapter.
4. Optionally, set the **RowError** property of the current row in the event object by assigning a value to it. This information is picked up by the dataset's **HasErrors** property.

   The following example illustrates how you can test whether a row was successfully updated within the **RowUpdated** event handler. If errors exist, the **RowError** property of the offending data row is set to the message of the error which automatically sets the **HasErrors** property to **true**. At this point you can handle the error based on the type of error raised as well as the needs of your application. Additionally you can set the **Status** property to continue processing the update, skip the remaining rows, or skip the current row.

```
' Visual Basic
Private Sub OleDbDataAdapter1_RowUpdated(ByVal sender As Object, _
ByVal e As System.Data.OleDb.OleDbRowUpdatedEventArgs) _
Handles OleDbDataAdapter1.RowUpdated
   If e.Status = UpdateStatus.ErrorsOccurred Then
      e.Row.RowError = e.Errors.Message
      ' Add code to reconcile the error.
   End If
End Sub

// C#
private void oleDbDataAdapter1_RowUpdated(object sender, System.Data.OleDb.OleDbRowUpdate
dEventArgs e)
{
   if (e.Status == UpdateStatus.ErrorsOccurred)
   {
      e.Row.RowError = e.Errors.Message;
      // Add code to reconcile the error.
   }
}
```

See Also

Dataset Updates in Visual Studio .NET | Locating Rows that Have Errors | Writing Event Handlers | UpdateStatus Enumeration | DataSet.HasErrors Property

Concurrency Control in ADO.NET

Concurrency control is the process used to identify and resolve updates to data that are made by multiple users simultaneously. The following topics explain common methods of concurrency control, as well as specific ADO.NET features for handling concurrency errors.

In This Section

Introduction to Data Concurrency in ADO.NET
Provides an overview of the different methods of concurrency control.
Walkthrough: Handling a Concurrency Exception
Provides step-by-step instructions for identifying and resolving a concurrency error.
Implementing Optimistic Concurrency with Dynamic SQL
Describes how to configure an ADO.NET data adapter to automatically create SQL commands to identify and raise concurrency errors.
Implementing Optimistic Concurrency with Stored Procedures
Describes how to configure an ADO.NET data adapter to automatically create stored procedures that will identify and raise concurrency errors.
Handling Concurrency Errors
Describes how to use the **DBConcurrencyException** object to identify concurrency exceptions and the actual record that caused the error.
Transactions in ADO.NET
Provides an overview of data transactions in ADO.NET.

Related Sections

Optimistic Concurrency
Defines optimistic concurrency and the different approaches for implementing it.
DBConcurrencyException Class
Describes the **DBConcurrencyException** class and provides links to its members.
Performing Transactions
Details how to programmatically create and execute a data transaction.

Introduction to Data Concurrency in ADO.NET

When multiple users attempt to modify data at the same time, controls need to be established in order to prevent one user's modifications from adversely affecting modifications from simultaneous users. The system of handling what happens in this situation is called *concurrency control*.

Types of Concurrency Control

In general, there are three common ways to manage concurrency in a database:

- **Pessimistic concurrency control** - a row is unavailable to users from the time the record is fetched until it is updated in the database.
- **Optimistic concurrency control** - a row is unavailable to other users only while the data is actually being updated. The update examines the row in the database and determines whether any changes have been made. Attempting to update a record that has already been changed results in a concurrency violation.
- **"Last in wins"** - a row is unavailable to other users only while the data is actually being updated. However, no effort is made to compare updates against the original record; the record is simply written out, potentially overwriting any changes made by other users since you last refreshed the records.

Pessimistic Concurrency

Pessimistic concurrency is typically used for two reasons. First, in some situations there is high contention for the same records. The cost of placing locks on the data is less than the cost of rolling back changes when concurrency conflicts occur.

Pessimistic concurrency is also useful for situations where it is detrimental for the record to change during the course of a transaction. A good example is an inventory application. Consider a company representative checking inventory for a potential customer. You typically want to lock the record until an order is generated, which would generally flag the item with a status of ordered and remove it from available inventory. If no order is generated, the lock would be released so that other users checking inventory get an accurate count of available inventory.

However, pessimistic concurrency control is not possible in a disconnected architecture. Connections are open only long enough to read the data or to update it, so locks cannot be sustained for long periods. Moreover, an application that holds onto locks for long periods is not scalable.

> **Note** If your underlying data source supports transactions, you can simulate pessimistic concurrency by updating your data within a transaction. For more information, see Transactions in ADO.NET.

Optimistic Concurrency

In optimistic concurrency, locks are set and held only while the database is being accessed. The locks prevent other users from attempting to update records at the same instant. The data is always available except for the exact moment that an update is taking place. For more information, see Optimistic Concurrency.

When an update is attempted, the original version of a changed row is compared against the existing row in the database. If the two are different, the update fails with a concurrency error. It is up to you at that point to reconcile the two rows, using business logic that you create.

Last in Wins

With "last in wins," no check of the original data is made and the update is simply written to the database. It is understood that the following scenario can occur:

- User A fetches a record from the database.
- User B fetches the same record from the database, modifies it, and writes the updated record back to the database.
- User A modifies the 'old' record and writes it back to the database.

In the above scenario, the changes User B made were never seen by User A. Be sure that this situation is acceptable if you plan to use the "last in wins" approach of concurrency control.

Concurrency Control in ADO.NET and Visual Studio .NET

ADO.NET and Visual Studio .NET use optimistic concurrency, because the data architecture is based on disconnected data. Therefore, you need to add business logic to resolve issues with optimistic concurrency.

If you choose to use optimistic concurrency, there are two general ways to determine if changes have occurred: the version approach (true version numbers or date-time stamps) and the saving all values approach.

Version Number Approach

In the version number approach, the record to be updated must have a column that contains a date-time stamp or version number. The date-time stamp or a version number is saved on the client when the record is read. This value is then made part of the update.

One way to handle concurrency is to update only if value in the WHERE clause matches the value on the record. The SQL representation of this approach is:

```
UPDATE Table1 SET Column1 = @newvalue1, Column2 = @newvalue2
WHERE DateTimeStamp = @origDateTimeStamp
```

Alternatively, the comparison can be made using the version number:

```
UPDATE Table1 SET Column1 = @newvalue1, Column2 = @newvalue2
WHERE RowVersion = @origRowVersionValue
```

If the date-time stamps or version numbers match, the record in the data store has not changed and can be safely updated with the new values from the dataset. An error is returned if they don't match. You can write code to implement this form of concurrency checking in Visual Studio .NET. You will also have to write code to respond to any update conflicts. To keep the date-time stamp or version number accurate, you need to set up a trigger on the table to update it when a change to a row occurs.

Saving All Values Approach

An alternative to using a date-time stamp or version number is to get copies of all the fields when the record is read. The DataSet object in ADO.NET maintains two versions of each modified record: an original version (that was originally read from the data source) and a modified version, representing the user updates. When attempting to write the record back to the data source, the original values in the data row are compared against the record in the data source. If they match, it means that the database record has not changed since it was read. In that case, the changed values from the dataset are successfully written to the database.

Each data adapter command has a parameters collection for each of its four commands (DELETE, INSERT, SELECT, and UPDATE). Each command has parameters for both the original values, as well as the current (or modified) values.

> **Note** Adding new records (the INSERT command) only requires the current values since no original record exists and removing records (the DELETE command) only requires the original values in order to locate the record to delete.

The following example shows the command text for a dataset command that updates a typical Customers table. The command is specified for dynamic SQL and optimistic concurrency.

```
UPDATE Customers SET CustomerID = @currCustomerID, CompanyName = @currCompanyName, ContactNam
e = @currContactName,
       ContactTitle = currContactTitle, Address = @currAddress, City = @currCity,
       PostalCode = @currPostalCode, Phone = @currPhone, Fax = @currFax
WHERE (CustomerID = @origCustomerID) AND (Address = @origAddress OR @origAddress IS NULL AND
Address IS NULL) AND (City = @origCity OR @origCity IS NULL AND City IS NULL)
      AND (CompanyName = @origCompanyName OR @origCompanyName IS NULL AND CompanyName IS NULL
) AND (ContactName = @origContactName OR @origContactName IS NULL AND ContactName IS NULL) AN
D (ContactTitle = @origContactTitle OR @origContactTitle IS NULL AND ContactTitle IS NULL)
      AND (Fax = @origFax OR @origFax IS NULL AND Fax IS NULL) AND (Phone = @origPhone OR @or
igPhone IS NULL AND Phone IS NULL) AND (PostalCode = @origPostalCode OR @origPostalCode IS NU
LL AND PostalCode IS NULL);
SELECT CustomerID, CompanyName, ContactName, ContactTitle, Address, City,
       PostalCode, Phone, Fax
FROM Customers WHERE (CustomerID = @currCustomerID)
```

Note that the nine SET statement parameters represent the current values that will be written to the database, whereas the nine WHERE statement parameters represent the original values that are used to locate the original record.

The first nine parameters in the SET statement correspond to the first nine parameters in the parameters collection. These parameters would have their **SourceVersion** property set to **Current**.

The next nine parameters in the WHERE statement are used for optimistic concurrency. These placeholders would correspond to the next nine parameters in the parameters collection, and each of these parameters would have their **SourceVersion** property set to **Original**.

The SELECT statement is used to refresh the dataset after the update has occurred. It is generated when you set the **Refresh the DataSet** option in the **Advanced SQL Generations Options** dialog box.

> **Note** The above SQL uses named parameters, whereas **OleDbDataAdapter** commands use question marks (?) as parameter placeholders.

By default Visual Studio will create these parameters for you if you select the Optimistic Concurrency option in the DataAdapter Configuration Wizard. It is up to you to add code to handle the errors based upon your own business requirements. ADO.NET provides a DBConcurrencyException object that returns the row that violates the concurrency rules. For more information, see Handling Concurrency Errors.

See Also

Responding to Database Update Errors| Optimistic Concurrency | DBConcurrencyException Class | Handling Concurrency Errors | Implementing Optimistic Concurrency with Dynamic SQL | Implementing Optimistic Concurrency with Stored Procedures

Walkthrough: Handling a Concurrency Exception

In this walkthrough you will create a Windows application that raises a concurrency error and illustrates one strategy for handling it. The walkthrough simulates two users working with the same data at the same time. The Windows Form you will create allows you to act as both users from a single form.

This walkthrough will demonstrate the following tasks:

1. User 1 and User 2 fill there respective datasets with the same data.
2. User 2 edits a record, updates the dataset, and writes the changes to the data source.
3. User 1 edits the same record, updates the dataset, and attempts to write the changes to the data source, which results in a concurrency error being raised.

You will catch the error, then display the different versions of the record, allowing the user (you) to determine what should happen with the pending changes made by User 1.

> **Note** Using a dataset is only one option for data access, and is not the optimal choice in some scenarios. Nonetheless, datasets are usually the right choice in Windows Forms applications, and this walkthrough illustrates one scenario in which datasets are an appropriate choice. For more information, see Recommendations for Data Access Strategies.

Prerequisites for this walkthrough:

- Access to the Pubs SQL Server sample database with permission to perform updates.

Create a New Project and Data Connection

You begin your walkthrough by creating a new Windows application in Visual Basic or Visual C#.

To create a new project

1. From the **File** menu, point to **New**, and then click **Project** to display the **New Project** dialog box.
2. Depending on what language you want to use, select **Visual Basic Projects** or **Visual C# Projects** in the **Project Types** pane, and then select **Windows Application**.
3. Name the project **ConcurrencyWalkthrough**, then click **OK**.

   Visual Studio will add the ConcurrencyWalkthrough project to Solution Explorer and display a new Windows Form in the designer.

To create a new data connection

1. In Server Explorer, create a new connection to the Pubs sample database. For more information, see Adding New Data Connections in Server Explorer.

   > **Security Note** Storing connection-string details (such as the server name, user name, and password) can have implications for the security of your application. Using Windows Integrated Security is a more secure way to control access to a database. For more information, see Database Security.

2. In Server Explorer, expand the connection you created in the previous step.
3. Expand the **Tables** area.
4. Drag the **authors** table onto your form.

   A **Connection** object and a **DataAdapter** object appear in the component tray below the form.

Create the Datasets

In this section you will create two datasets named **DsAuthors1** and **DsAuthors2**. These datasets will represent the data that the two simultaneous users are working with. Next you will add two DataGrid controls to the form and bind them to the datasets. Finally, two Button controls will be added to the form: an **Update** button and a **Reset** button. The **Update** button will change a record in **DsAuthors1** and attempt to write the change to the database. The **Reset** button resets the database back to the original record so the walkthrough will work if run more than once.

To create two new datasets

1. Select the **DataAdapter** object.
2. From the **Data** menu, select **Generate Dataset**.

   The **Generate Dataset** dialog box appears.

3. Select **New** and name the dataset **DsAuthors**.

   An instance named **DsAuthors1** appears in the component tray.

4. From the **Data** tab of the Toolbox, drag a **DataSet** onto the form.

   The **Add Dataset** dialog box appears.

5. Confirm that **Typed dataset** is selected and **ConcurrencyWalkthrough.DsAuthors** appears in the **Name** box.

   An instance named **DsAuthors2** appears in the component design tray.

Bind the Data and Add Buttons

The data grids are used only to display the data. No data should be edited in the data grids during this walkthrough.

To add two DataGrids to the form

1. From the **Windows Forms** tab of the Toolbox, drag a **DataGrid** object onto the left side of your form.
2. From the **Windows Forms** tab of the Toolbox, drag a **DataGrid** object onto the right side of your form.

To bind the datasets to the DataGrid Controls

1. Select **DataGrid1** and set the following properties in the Properties window:

Property	Setting
DataSource	**DsAuthors1**
DataMember	**authors**
CaptionText	**DsAuthors1**

2. Select **DataGrid2** and set the following properties in the Properties window:

Property	Setting
DataSource	**DsAuthors2**
DataMember	**Authors**
CaptionText	**DsAuthors2**

To add the Update and Reset buttons to the form

1. From the **Windows Forms** tab of the Toolbox, drag a **Button** control onto the from and place it above **DataGrid1**.
2. With the button selected, in the Properties window name the button **btnUpdate** and set its **Text** property to **Update**.
3. From the **Windows Forms** tab of the Toolbox, drag a second **Button** object onto the from and place it above **DataGrid2**.
4. With the button selected in the Properties window, name the button **btnReset** and set its **Text** property to **Reset**.

Reset the Database

You will add code to the form that will reset the database to known values (in case you want to run the walkthrough more than once).

To add code to reset the database

- Right-click the form, choose **View Code** from the shortcut menu, and then insert the following code:

```
' Visual Basic
Private Sub resetDatabase()
   ' Fill the DsAuthors1 dataset with data.
   SqlDataAdapter1.Fill(DsAuthors1)
   ' Reset the au_fname in the first row to 'John'.
   DsAuthors1.authors(0).au_fname = "John"
   ' Write the record back to the database.
   SqlDataAdapter1.Update(DsAuthors1)
End Sub
```

```
// C#
private void resetDatabase()
{
   // Fill the dsAuthors dataset with data.
   sqlDataAdapter1.Fill(dsAuthors1);
   // Reset the au_fname in the first row to "John".
   dsAuthors1.authors[0].au_fname = "John";
   // Write the record back to the database.
   sqlDataAdapter1.Update(dsAuthors1);
}
```

Fill the Datasets

This step fills both datasets with the same data.

To add code to fill the datasets from the database

- Insert the following code into the Code Editor:

```
' Visual Basic
Private Sub FillTheDataSets()
   SqlDataAdapter1.Fill(DsAuthors1)
   SqlDataAdapter1.Fill(DsAuthors2)
End Sub

// C#
private void FillTheDataSets()
{
   sqlDataAdapter1.Fill(dsAuthors1);
   sqlDataAdapter1.Fill(dsAuthors2);
}
```

Simulate Changes by User 2

To add code to simulate changes by User 2

- Insert the following code into the Code Editor:

```
' Visual Basic
Private Sub user2changes()
   ' Simulate a second user changing a record.
   DsAuthors2.authors(0).au_fname = "User 2"
   ' Write it back to the database.
   SqlDataAdapter1.Update(DsAuthors2.GetChanges())
   ' Refresh DsAuthors2 with the updated data.
   SqlDataAdapter1.Fill(DsAuthors2)
End Sub

// C#
private void user2changes()
{
   // Simulate a second user changing a record.
   dsAuthors2.authors[0].au_fname = "User 2";
   // Write it back to the database.
   sqlDataAdapter1.Update(dsAuthors2.GetChanges());
   // Refresh dsAuthors2 with the updated data.
   sqlDataAdapter1.Fill(dsAuthors2);
}
```

Create the Form_Load Event Handler

To add code into the Form_Load event to initialize the walkthrough

1. Double-click an empty area of the form to automatically create the **Form_Load** event handler.
2. Add code so the event handler looks like the following:

```
' Visual Basic
Private Sub Form1_Load(ByVal sender As System.Object, _
ByVal e As System.EventArgs) Handles MyBase.Load
   resetDatabase()
   FillTheDataSets()
   user2changes()
End Sub

// C#
private void Form1_Load(object sender, System.EventArgs e)
{
   resetDatabase();
   FillTheDataSets();
   user2changes();
}
```

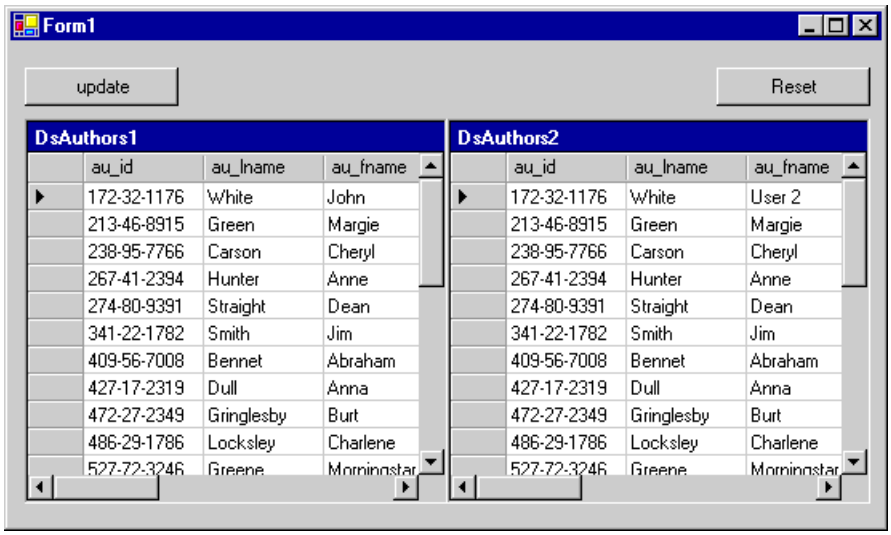
3. Save your project.

Run the Application

1. Press **F5** to run the application

   The form appears with two datagrids filled with data from the authors table in the Pubs database. The `au_fname` field of the first record in DsAuthors1 should be **John**. The `au_fname` field of the first record in DsAuthors2 should be **User 2**.

   Your form will look similar to the following:



2. From the **Debug** menu, select **Stop Debugging**.

Update the Dataset and Write the Changes to the Database

Next, you will write code that will attempt to update the database with changes from the **DsAuthors1** dataset. If successful, the AcceptChanges method of the dataset is called and a message box displays a success message. If the update fails for any reason, the error is caught and a message box displays the error message, with the type of error object as the title of the message box. Because this walkthrough is designed to raise a concurrency error, the successful message box is shown for completeness.

> **Note** This **Try...Catch** block will catch any error. Later in this walkthrough you will add an additional catch statement to specifically handle the concurrency error.

To update the database

1. Call the Update method.

2. Create an exception handler that displays a message box.

Your code should look like this:

```
' Visual Basic
Private Sub updateDatabase()
    Try
        ' Update the database with the changes from DsAuthors1.
        SqlDataAdapter1.Update(DsAuthors1.GetChanges)
        DsAuthors1.AcceptChanges()
        MessageBox.Show("The update was successful!")
    Catch ex As Exception
        ' Display information about update errors.
        MessageBox.Show("Update Failed", ex.GetType.ToString)
    End Try
End Sub

// C#
private void updateDatabase()
{
    try
    {
        // Update the database with the changes from dsAuthors1.
        sqlDataAdapter1.Update(dsAuthors1.GetChanges());
        dsAuthors1.AcceptChanges();
        MessageBox.Show("The update was successful.");
    }
    catch (Exception ex)
    {
        // Display information about update errors.
        MessageBox.Show("Update Failed", ex.GetType().ToString());
    }
}
```

Next, you will add code that changes the au_fname column in the DsAuthors1 dataset. The code then calls the **updateDatabase** procedure to try to write this change to the database. Because the value was changed earlier by User 2, a concurrency error will be raised.

To update the DsAuthors1 dataset

1. Double-click the **Update** button.
2. Create the **btnUpdate_Click** event handler**:**

```
' Visual Basic
Private Sub btnUpdate_Click(ByVal sender As System.Object, _
ByVal e As System.EventArgs) Handles btnUpdate.Click
   ' Change au_fname in the first row of DsAuthors1 to User 1.
   DsAuthors1.authors(0).au_fname = "User 1"
   updateDatabase()
End Sub

// C#
private void btnUpdate_Click(object sender, System.EventArgs e)
{
    // Change au_fname in the first row of dsAuthors1 to "User 1".
    dsAuthors1.authors[0].au_fname = "User 1";
    updateDatabase();
}
```

3. Save your project.
4. Press **F5** to run the application.
5. Click the **Update** button to change the **au_fname** field of the first record to **User 1**.

   The concurrency error will be raised.

Handling Concurrency Errors

How you handle the error is dependent upon the specific business rules that govern your application. The following strategy to handle the error will be used as an illustration. The application will present the user with three versions of the record:

- The current record in the database
- The original record in the dataset
- The proposed changes in the dataset

The user will then be able to either overwrite the database with the proposed change or cancel the update and refresh the dataset.

Creating a Custom Error Handler

When you perform an update, you generally want to do so in a structured exception handler so that you can catch errors. In the code you used above, you used a **try...catch** block that caught all errors — that is, a structure that includes a generic **catch** statement for any error.

You can also catch specific errors, so that you can respond appropriately. To illustrate, this walkthrough will add an exception handler for one specific error — namely, a concurrency error, which you can examine using the DbConcurrencyException object. Here, you will handle this error by displaying information to the user.

To add specific handling for the DBConcurrencyException error

1. If the application is still running, exit run mode to return to the Code Editor.
2. Add a second **catch** statement above the existing one in the **updateDatabase** method.
3. Pass the **DBConcurrencyException** object to the `createMessage` procedure, which you will create in the next section.

```
' Visual Basic
Private Sub updateDatabase()
    Try
        ' Update the database with the changes from dsAuthors1.
        SqlDataAdapter1.Update(DsAuthors1.GetChanges())
        DsAuthors1.AcceptChanges()
        MessageBox.Show("The update was successful!")
    Catch dbcx As DBConcurrencyException
        createMessage(dbcx)
    Catch ex As Exception
        MessageBox.Show("Update Failed", ex.GetType().ToString())
    End Try
End Sub

// C#
private void updateDatabase()
{
    try
    {
        // Update the database with the changes from dsAuthors1.
        sqlDataAdapter1.Update(dsAuthors1.GetChanges());
        dsAuthors1.AcceptChanges();
        MessageBox.Show("The update was successful!");
    }
    catch (DBConcurrencyException dbcx)
    {
        createMessage(dbcx);
    }
    catch (Exception ex)
    {
        MessageBox.Show("Update Failed", ex.GetType().ToString());
    }
```

}

Displaying Choices to the User

The code you just wrote calls the `createMessage` procedure to display error information to the user. For this walkthrough, you will use a message box to display the different versions of the record to the user and allow the user to choose whether to overwrite the record with new changes or cancel the edit.

> **Note** For simplicity's sake, this walkthrough uses the second dataset (DsAuthors2) as the data source for fetching the current record in the database. In a real-world application you would requery the actual data source to fetch the current value of the record that raised the error.

To create the createMessage procedure

- Create the error handler by adding the following code to the Code Editor:

```
' Visual Basic
Private Sub createMessage(ByVal dbcx As DBConcurrencyException)
   ' Declare variables to hold the row versions for display
   ' in the message box.
   Dim strInDs As String = "Original record in DsAuthors1:" _
      & ControlChars.CrLf
   Dim strInDB As String = "Current record in database:" _
      & ControlChars.CrLf
   Dim strProposed As String = "Proposed change:" & ControlChars.CrLf
   Dim strPromptText As String = "Do you want to overwrite the current record in the data
base with the proposed change?" & ControlChars.CrLf
   Dim strMessage As String
   Dim response As System.Windows.Forms.DialogResult


   ' Locate the current record in the data source that caused the exception
   Dim rowInDB As DataRow = _
      DsAuthors2.authors.FindByau_id(CType(dbcx.Row("au_id"), String))

   ' Loop through the column values.
   Dim i As Integer
   For i = 0 To dbcx.Row.ItemArray.Length - 1
      strInDs &= dbcx.Row(i, DataRowVersion.Original) & _
         ControlChars.Tab
      strInDB &= rowInDB(i, DataRowVersion.Current) & ControlChars.Tab
      strProposed &= dbcx.Row(i, DataRowVersion.Current) & _
         ControlChars.Tab
   Next

   ' Create the message box text string.
   strMessage = strInDs & ControlChars.CrLf
   strMessage &= strInDB & ControlChars.CrLf
   strMessage &= strProposed & ControlChars.CrLf
   strMessage &= strPromptText

   ' Display the message box.
   response = MessageBox.Show(strMessage, "Update Failed", _
      MessageBoxButtons.YesNo)
   processResponse(response)
End Sub

// C#
private void createMessage(DBConcurrencyException dbcx)
{
    // Declare variables to hold the row versions for display
    // in the message box.
    string strInDs = "Original record in dsAuthors1:\n";
```

```
    string strInDB = "Current record in database:\n";
    string strProposed = "Proposed change:\n";
    string strPromptText = "Do you want to overwrite the current " +
      "record in the database with the proposed change?\n";
    string strMessage;
    System.Windows.Forms.DialogResult response;

    // Loop through the column values.
   DataRow rowInDB =
      dsAuthors2.authors.FindByau_id(dbcx.Row["Au_ID"].ToString());
   for (int i = 0; i < dbcx.Row.ItemArray.Length; i++)
   {
      strInDs += dbcx.Row[i, DataRowVersion.Original] + "\n";
      strInDB += rowInDB[i, DataRowVersion.Current] + "\n";
      strProposed += dbcx.Row[i, DataRowVersion.Current] + "\n";
   }

    // Create the message box text string.
    strMessage = strInDs + "\n" + strInDB + "\n" + strProposed + "\n"
        + strPromptText;

    // Display the message box.
    response = MessageBox.Show(strMessage, "Update Failed",
       MessageBoxButtons.YesNo);
    processResponse(response);
}
```

Processing the User's Response

You will also need code to process the user's response to the message box. The options are either to overwrite the current record in the database with the proposed change or not. If the user chooses yes, the Merge method of **DsAuthors1** is called with the *preserveChanges* argument set to **true**. This takes the original versions of the data rows in **DsAuthors2** and merges them with the current versions of data rows in **DsAuthors1**. This will cause the update attempt to be successful, because the original version of the record now matches the database.

To process the user input from the message box

- Add the following code to the Code Editor:

```
' Visual Basic
Private Sub processResponse(ByVal response As _
System.Windows.Forms.DialogResult)
   ' Execute the appropriate code depending on the button
   ' selected in the message box.
   Select Case response
      Case System.Windows.Forms.DialogResult.Yes
         ' Overwrite the database with the proposed record.
         DsAuthors1.Merge(DsAuthors2, True)
         SqlDataAdapter1.Update(DsAuthors1)
         DsAuthors1.AcceptChanges()
      Case System.Windows.Forms.DialogResult.No
         ' Cancel proposed changes and refresh DsAuthors1.
         DsAuthors1.Merge(DsAuthors2)
   End Select
End Sub

// C#
private void processResponse(
System.Windows.Forms.DialogResult response)
{
```

```
    // Execute the appropriate code depending on the button selected
    // in the message box.
    switch (response)
    {
        case System.Windows.Forms.DialogResult.Yes :
            // Overwrite the database with the proposed record.
            dsAuthors1.Merge(dsAuthors2, true);
            sqlDataAdapter1.Update(dsAuthors1);
            dsAuthors1.AcceptChanges();
            break;
        case System.Windows.Forms.DialogResult.No :
            // Cancel proposed changes and refresh dsAuthors1.
            dsAuthors1.Merge(dsAuthors2);
            break;
    }
}
```

Resetting the Data

To reset the form, the **Form1_Load** method will be modified to execute when the **Reset** button is clicked. Because Visual Basic and Visual C# have different event models, there are different ways to associate an event handler with an event.

> **Note** One method can be the event handler for multiple events of multiple objects. For more information, see Connecting Multiple Events to a Single Event Handler in Windows Forms.

To create an event handler for the btnReset button with Visual Basic

1. In the Code Editor, locate the **Form_Load** event handler.
2. Insert a comma and then **btnReset.Click** to the end of the handler to make the **Form1_Load** method the event handler for the **Click** event of `btnReset`.

```
Private Sub Form1_Load(ByVal sender As System.Object, _
ByVal e As System.EventArgs) Handles MyBase.Load, btnReset.Click
   resetDatabase()
   FillTheDataSets()
   user2changes()
End Sub
```

To create an event handler for the btnReset button with Visual C#

1. Open the form in the designer.
2. Click the **Reset** button.
3. In the Properties window, click the **Events** button in the Properties window toolbar.
4. Locate the **Click** event, and then click the arrow to see all the methods that could respond to the event.

   In this case, **Form1_Load** appears because it has the the right signature for a **button.click** event.

5. Select **Form1_Load**.

   The code to associate the Form1_Load method with the **btnReset.Click** event is automatically generated. **Form1_Load** now responds to both **Form1.Load** and **btnReset.Click**.

   > **Tip** To view the generated code, double-click the form and expand the dimmed "Windows Form Designer generated code" section.

Run the Application

1. Press **F5** to run the application.
2. Click the **Update** button.

   The concurrency error will be raised, and the message box will appear.

See Also

Adding New Data Connections in Server Explorer | Updating the Database with a DataAdapter and the DataSet | Dataset Updates in Visual Studio .NET | ADO.NET Datasets | Data Walkthroughs | Concurrency Control in ADO.NET | Debugging Managed Code | Deploying Applications

Implementing Optimistic Concurrency with Dynamic SQL

Visual Studio can implement optimistic concurrency with dynamic SQL. Ultimately, this approach builds an SQL command string with a WHERE clause that contains all of the original data-store values. This string is then invoked as a SQL command on the data store. The WHERE clause will not return any records if changes have occurred to the record in the data store. These concurrency exceptions must be handled according to business needs.

To implement optimistic concurrency with dynamic SQL

1. If you do not already have a connection to a database, add one using Server Explorer. For more information, see Adding New Data Connections in Server Explorer.
2. Drag a table from Server Explorer onto the designer surface.

   You will get a **DataAdapter** object and a **Connection** object. If you are working with a Windows Application project, a component designer tray will appear at the bottom of the screen. This tray allows visual development of components that have no run-time user interface. If you started with an XML Web service, the entire main window will provide a component design surface.

3. Right-click the **DataAdapter**, and then select **Configure Data Adapter** to display the **Data Adapter Configuration Wizard**.
4. Click **Next** and then click **Next** again to bring up the **Choose a Query Type** page.
5. Make sure that **Use SQL statement** is selected, and then click **Next**.
6. In the **Generate the SQL Statements** page, click the **Query Builder** button.
7. In the **Query Builder**, select the fields you want.
8. Select the columns you want, and then click **OK**.
9. In the **Generate the SQL Statements** page, click the **Advanced Options** button and note that **Use optimistic concurrency** is selected by default.
10. Click **OK** and then **Finish** to complete the wizard.

You can inspect the results of the configuration by examining the three update-related data adapter command properties: **DeleteCommand**, **InsertCommand**, and **UpdateCommand**.

See Also

Optimistic Concurrency | Implementing Optimistic Concurrency with Stored Procedures | Handling Concurrency Errors

Implementing Optimistic Concurrency with Stored Procedures

Visual Studio can implement optimistic concurrency with stored procedures. Ultimately this approach builds an SQL command with a WHERE clause that contains all of the original data store values. Parameters that include the WHERE clause values are passed to a stored procedure within the data store. The WHERE clause will not return any records if changes have occurred to the data store. These changes can be handled according to business needs.

To implement optimistic concurrency with stored procedures

1. If you do not already have a connection to a database, add one. For more information, see Adding New Data Connections in Server Explorer.
2. Drag a table from Server Explorer onto the designer surface.

   You will get an adapter object and a connection object.

   > **Note** If you are working with a project that has an interface (such as a Windows application or a Web application), a component tray will appear at the bottom of the designer. This tray allows visual development of components that have no run-time user interface. If you are working with other project types (such as an XML Web service), the objects will appear on the Component Designer — that is, the entire main window will provide a component design surface.

3. Right-click the adapter object, and then select **Configure Data Adapter** from the shortcut menu to display the **Data Adapter Configuration Wizard**.
4. Click **Next** and then click **Next** again for the **Choose a Query Type** page.
5. Make sure that **Create new stored procedures** is selected, and then click **Next**.
6. In the **Generate the stored procedures** page, click the **Query Builder** button.
7. In the **Query Builder**, check the individual fields you want.

   > **Note** In the **Generate the stored procedures** page, click the **Advanced Options** button, and notice the **Use optimistic concurrency** option is selected by default.

8. Click **OK**, and then click **Next** to bring up the **Create the Stored Procedures** page.
9. In the **Create the Stored Procedures** page, assign names to each of the four stored procedures that will be generated, and you can have the wizard automatically create them in the database for you or choose to manually create them later. Click **Next** then **Finish** to complete the wizard.

You can inspect the results of the configuration by examining the three update-related data adapter command properties: the **DeleteCommand**, **InsertCommand**, and the **UpdateCommand**.

See Also

Optimistic Concurrency | Implementing Optimistic Concurrency with Dynamic SQL | Handling Concurrency Errors

Handling Concurrency Errors

ADO.NET provides a DBConcurrencyException object to assist in resolving issues arising from concurrency violations. The **DBConcurrencyException** object returns the data row that caused the error. For more information, see DBConcurrencyException Members.

The following example attempts to update a data source with the contents of `myDataset` from within a **Try...Catch** block, if an error is raised an error message along with the first column of the offending data row is displayed.

> **Note** The code below is an illustration of one strategy in handling a database update error. The code assumes several things; an existing connection to a database, an existing dataset named myDataset, as well as the assumption that execution of the update command will raise a concurrency violation. For more information and a complete example, see Walkthrough: Handling a Concurrency Exception.

To resolve a concurrency violation

1. Execute the command to update the database from within a **Try...Catch** block.
2. If an exception is raised, inspect the **Catch** statement's **Row** property to determine what caused the violation.
3. Add code to resolve the error based on your application's business rules.

   The following code uses SqlDataAdapter1 and myDataset as examples of the data adapter and dataset in your application.

```
' Visual Basic
Try
   SqlDataAdapter1.Update(myDataset)
Catch ex As DBConcurrencyException
   Dim customErrorMessage As String
   customErrorMessage = "Concurrency violation" & vbCrLf
   customErrorMessage += Ctype(ex.Row.Item(0), String)
   MessageBox.Show(customErrorMessage)
   ' Replace the above code with appropriate business logic
   ' to resolve the concurrency violation.
End Try

// C#
try
{
   SqlDataAdapter1.Update(myDataset);
}
catch (DBConcurrencyException ex)
{
   string customErrorMessage;
   customErrorMessage = "Concurrency violation\n";
   customErrorMessage += ex.Row[0].ToString();
   // Replace the above code with appropriate business logic
   // to resolve the concurrency violation.
}
```

See Also

DBConcurrencyException Class| SqlDataAdapter Class | OleDbDataAdapter Class | Try...Catch...Finally Statements (Visual Basic) | DBConcurrencyException Members

Transactions in ADO.NET

Transactions are groups of database commands that execute as a package. Using a transaction gives your application the ability to abort (roll back) all changes executed from within the transaction if any errors occur during any part of the transaction process. For example, consider an application with two tables, where one table represents inventory and the other represents orders. When an order is added to one table, the corresponding inventory needs to reduced in the other table. If an update to the Orders table was successful and an update to the Inventory table failed, the integrity of your data would be compromised. In order to guarantee that both tables were updated successfully, you could package both data commands into a single transaction. If one table updated successfully and the other table did not, you can rollback the entire transaction, resolve the issues that caused the failure, and re-attempt the transaction again.

> **Note** Your underlying data source must support transactions in order to use the transaction features supported in Visual Studio .NET.

Transactions are created by calling the **BeginTransaction** method of a data connection object. For more information, see Performing Transactions.

See Also

Processing Transactions | Performing Transactions | SqlTransaction Class | SqlConnection.BeginTransaction Method | OleDbTransaction Class | OleDbConnection.BeginTransaction Method

XML Schemas and Data

XML plays a large role in the way data is handled in Visual Studio .NET. XML is used in the following ways:

- Datasets can serialize their data as XML. The schema of a dataset — what tables, columns, data types, and constraints are in the dataset — is defined using an XML Schema (.xsd file).
- You can infer XML Schemas from existing datasets and create datasets from existing XML Schemas.

Visual Studio makes working with XML files, XML Schemas, and ADO.NET datasets easy by providing you with a visual tool, the XML Designer.

The XML Designer is used to create and edit XML Schema files, ADO.NET datasets, XML data files, and XML source code. The XML Designer has three views:

- **Schema view** for visually editing XML Schemas and ADO.NET datasets.
- **Data view** with a structured data grid for visually editing XML data files.
- **XML view** with color-coding, IntelliSense, and statement completion for working directly with XML source code.

In This Section

Introduction to XML Schemas
Provides an overview of what XML Schemas are, how they are used to validate XML data, and some of the advantages over previous schema technologies.
XML Technology Backgrounder
Provides an overview of what XML is, how it relates to other markup languages such as HTML, and some of the common uses of XML.
XML Schema Syntax
Provides an overview of the structure of a XML Schema, and an explanation of the opening schema tags.
XML Walkthroughs
Introduces several step-by-step scenarios that demonstrate working with the XML Designer.
XML Schemas and the XML Designer
Introduces topics that describe the different elements in XML Schemas and how they are created using the XML Designer.
Creating XML Schemas and Datasets
Introduces topics that describe the different ways that schemas and datasets can be created using the XML Designer as well as describing the connection between datasets and schemas.
Relational Data in XML Schemas
Introduces topics that describe the different options in XML Schemas for representing relational data.
XML Files
Introduces topics that explain how to use the XML Designer when working with XML documents.

Related Sections

XML in Visual Studio
Provides links to a variety of topics about XML.
Introduction to Data Access with ADO.NET
Provides a high level overview of the most important concepts of data access in ADO.NET.
Introduction to Datasets
Provides an overview of what ADO.NET datasets are, how they can be typed or untyped, what they are used for, and how they are structured.
System.Xml Namespace
Describes the **System.Xml** namespace and provides a table of relevant classes.
World Wide Web Consortium (W3C) Web site (http://www.w3.org/)
Links to the XML Schema primer and other XML info posted by the W3C.
XML Developer Center on MSDN Online (http://msdn.microsoft.com/xml)
Provides various resources about XML.
XML Schema Reference (XSD)
Provides reference topics for schema elements and data types.
XML Designer
Explains the XML design tool in Visual Studio.

Introduction to XML Schemas

XML Schemas are documents that are used to define and validate the content and structure of XML data, just as a database schema defines and validates the tables, columns, and data types that make up a database.

An XML Schema defines and describes certain types of XML data by using the XML Schema definition language (XSD). XML Schema elements (elements, attributes, types, and groups) are used to define the valid structure, valid data content, and relationships of certain types of XML data. XML Schemas can also provide default values for attributes, and elements. For more information, see the World Wide Web Consortium (W3C) Web site (http://www.w3.org/).

You use an XML Schema to guarantee consistency among certain types of XML data that is shared between applications and organizations. An XML Schema can be used as a contract for data exchange between two applications. Organizations can publish schemas that describe the format of the XML their applications produce and consume. Other organizations and applications that wish to exchange data can then build their applications around these schemas so their xml messages will be understood.

For example, a purchase order represented in XML can be validated with an XML Schema before it is sent between buyer and seller. This validation verifies that all of the elements (individual pieces) of data exist, are in the expected sequence, and are all of the correct data type. This ensures that the recipient of the purchase order will be able to interpret the data correctly when it is received.

XML Schemas in Visual Studio conform to the W3C recommendation regarding XML Schema based on the XML Schema definition language (XSD). These are some of the advantages that XML Schemas have over previous technologies, such as DTDs:

- XML Schemas use XML syntax, so you do not need to learn a new syntax to define your data structure.
- XML Schemas support reusable types and allow you to create new types using inheritance.
- XML Schemas allows you to group elements to control the recurrence of elements and attributes.

For more information about schemas and how the elements they represent constrain data, see XML Elements, Attributes, and Types.

For more information about XML Schemas, see the XML Schema Reference (XSD) and MSDN Online XML Developer Center (http://msdn.microsoft.com/xml).

See Also

XML Schemas and Data | Creating XML Schemas and Datasets |
XML Schema Primer on the World Wide Web Consortium Web Site (http://www.w3.org/) | XML Designer |
Walkthrough: Creating an XML Schema with the XML Designer | Introduction to Data Access with ADO.NET |
Introduction to Datasets

XML Technology Backgrounder

Extensible Markup Language (XML) provides a way to describe structured data. Unlike HTML tags, which are primarily used to control the display and appearance of data, XML tags are used to define the structure and data types of the data itself.

XML uses a set of tags to delineate elements of data. Each element encapsulates a piece of data that may be very simple or very complex. You can define an unlimited set of XML tags. For example, you might define XML tags to declare pieces of data from a purchase order, such as the price, tax, shipping address, billing address, and so on. As XML tags are adopted throughout an organization and across organizations, data from all kinds of different data stores will be easier to exchange and manipulate.

XML is simple, platform-independent, and a widely adopted standard. The advantage of XML over HTML is that it separates the user interface from the structured data. This separation of data from presentation enables the integration of data from diverse sources. Customer information, purchase orders, research results, bill payments, medical records, catalog data, and other information can be converted to XML.

XML vs. HTML

The following are a few important points to keep in mind about the relationship between XML and HTML:

- XML is not a replacement for HTML; in fact, XML can be considered a complement to HTML. XML and HTML have different goals: HTML is designed to display data and is focused on how data looks, while XML is designed to describe data and to focus on what data is.
- Like HTML, XML does not do anything. While XML tags can be used to describe the structure of an item such as a purchase order, it does not contain any code that can be used to send that purchase order, process it, or ensure that it is filled. Other people must write code to actually do these things with your XML-formatted data.
- Unlike HTML, XML tags are defined by the author of a schema or document and are unlimited. HTML tags are predefined; HTML authors can only use tags that are supported by the current HTML standard.

Uses of XML

XML is an extremely flexible way to pass around data. The following are all examples where XML can be used:

- An ordinary document
- A structured record, such as an appointment record or purchase order
- Internet/intranet Web applications that move data
- An object with data, such as the persistent format of an object or ActiveX control
- A data record, such as the result set of a query
- Meta-content about a Web site, such as Channel Definition Format (CDF)
- Graphical presentation, such as an application's user interface
- Links between information and people on the Web
- C# code, which can be documented with XML; for more information, see XML Documentation
- Discovery documents used to locate available XML Web services; For more information, see XML Web Service Discovery.

The following lists several advantages XML has over other formats when storing information:

- XML formats are text-based, making them more readable, easier to document, and sometimes easier to debug.
- XML documents can use much of the infrastructure already built for HTML, including the HTTP protocol and some browsers. HTTP allows XML to be transferred across firewalls.
- XML parsing is well defined and widely implemented, making it possible to retrieve information from XML documents in a variety of environments.
- Applications can rely on XML parsers to do some structural validation, as well as data type checking (when schemas are used).
- XML is built on a Unicode foundation, making it easier to create internationalized documents.

XML is not appropriate for every situation, however. XML documents tend to be more verbose than the binary formats they replace. They take up more network bandwidth and storage space, or require more processor time for compression. XML parsing can be slower than parsing highly optimized binary formats and can require more memory. However, careful application design can avoid some of these problems.

Validating XML Documents

To validate that XML documents contain the desired data and structure, an XML Schema must be associated with the XML document. XML Schemas are the rules that define how elements and attributes are structured to form XML documents. You can share schemas between organizations to make it simple to transfer and process shared data. For more information, see Introduction to XML Schemas.

Displaying XML Data

There are several ways that can be used to display (or present) XML data.

There are even mechanisms of data binding that can be used along with style sheets to arrange XML data into a visual presentation and to add interactivity.

Among others, here are a few methods to display XML:

- XSLT – Extensible Style Sheet Language
- CSS – Cascading Style Sheets
- Microsoft Internet Explorer

For more information, see MSDN Online XML Developer Center Web site (http://msdn.microsoft.com/xml/default.asp).

XML Resources

If you are familiar with HTML, you can learn to create XML documents, requiring only that they be valid and well formed. For more information about XML, see the XML SDK and MSDN Online XML Developer Center Web site (http://msdn.microsoft.com/xml/default.asp).

For more information, see the World Wide Web Consortium (W3C) Specification for XML (http://www.w3.org/XML/).

See Also

XML Schemas and Data | XML Schemas and the XML Designer| XML Elements, Attributes, and Types | XML in Visual Studio | XML Documentation | Introduction to Datasets | Introduction to Data Access with ADO.NET

XML Schema Syntax

The example XML Schema below contains the following items:

- The first statement, `<?xml version="1.0" encoding="utf-8"?>`, specifies the version of XML in use.
- The second statement consists of several parts:

  The `xs:schema` declaration indicates that this is a schema and that the xs: prefix will be used before schema items.

  The `xmlns:xs="http://www.w3.org/2001/XMLSchema"` declaration indicates that all tags in this schema should be interpreted according to the default namespace of the World Wide Web Consortium (W3C), which was created in 2001.

  The `targetnamespace` declaration names this schema as XMLSchema1.xsd and indicates its default location, which will be in a default URI (universal resource identifier) on your development server called tempuri.org.

  For more information about xmlns and targetnamespace, see schema Element.

  > **Note** When you create a schema with the XML Designer, the version and namespace statements are automatically added for you. You might modify the contents of these two declarations in some cases. For example, if you want to use a tag prefix other than "XS," you would need to define the default namespace for that tag in addition to the XS tag prefix.

- An unnamed complex type called "addressType" is defined that will contain five elements of various data types. Note that each of the elements it contains is a simple, named type.
- A complex element named "purchaseOrder" is defined. It contains elements that make use of the previously defined addressType. This creates a nested relationship between the purchaseOrder element and these elements.
- The final element in the purchaseOrder, Item, contains a facet called **maxOccurs**. Facets set limits on the type of content a simple type can contain. **maxOccurs** is used to constrain how many times an element can occur in the documents created from this schema. By default, **maxOccurs** is equal to one. In this case, setting it to unbounded indicates that the Item element can reoccur as many times as needed.

The following example shows a sample XML Schema:

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema targetNamespace="http://tempuri.org/XMLSchema1.xsd" xmlns:xs="http://www.w3.org/20
01/XMLSchema">
<xs:complexType name="addressType">
   <xs:sequence>
      <xs:element name="street1" type="xs:string"/>
      <xs:element name="street2" type="xs:string"/>
      <xs:element name="city" type="xs:string"/>
      <xs:element name="state" type="xs:string"/>
      <xs:element name="zip" type="xs:integer"/>
   </xs:sequence>
</xs:complexType>

<xs:element name="purchaseOrder">
      <xs:complexType>
         <xs:sequence>
            <xs:element name="shipTo" type="addressType" />
            <xs:element name="billTo" type="addressType" />
            <xs:element name="shipDate" type="xs:date" />
            <xs:element name="item" maxOccurs="unbounded"/>
         </xs:sequence>
      </xs:complexType>
   </xs:element>
</xs:schema>
```

See Also

Introduction to XML Schemas | Walkthrough: Creating an XML Schema with the XML Designer

XML Walkthroughs

This section provides step-by-step topics that demonstrate some of the common ways to use XML files and XML Schemas based on the XML Schema definition language (XSD) defined by the World Wide Web Consortium (W3C).

In the Visual Basic and Visual C# Documentation

Walkthrough: Creating an XML File with an Associated XML Schema
Describes how to create an XML Schema and an XML file and use the XML editor.
Walkthrough: Creating a Dataset with Tables, Keys, and One-to-Many Relationships
Explains how to create a dataset to represent a one-to-many relationship .
Walkthrough: Creating an XML Schema with the XML Designer
Describes how to create a schema to represent a purchase order.
XML Schemas and Data
Lists topics about working with XML data and XML Schemas.
Accessing Data
Provides a list of topics about working with ADO.NET in Visual Studio.
Data Walkthroughs
Lists all the step-by-step topics about working with data in Visual Basic and Visual C#.
Walkthroughs in Visual Basic and Visual C#
Lists all the step-by-step topics that use Visual Basic and Visual C#.

Additional Information

Samples and Walkthroughs
Provides links to all sample applications and step-by-step topics in Visual Studio.
XML in Visual Studio
Summarizes the support for XML in all Visual Studio products.
Visual Basic Code Example Topics
Shows quickstart-like code examples that help you perform common tasks in Visual Basic .NET.

XML Schemas and the XML Designer

XML Schemas are combinations of element, attribute, type, and group definitions; whereas ADO.NET typed datasets are classes that are defined by XML Schemas (.xsd files). XML Schemas in Visual Studio use the XML Schema definition language (XSD) that conforms to the recommendation for XML Schema of the World Wide Web Consortium (W3C). For more information about XML Schema, see XML Schema Reference (XSD).

This section will explain how to add new schema (.xsd) files to your project and how to create the individual pieces that XML Schemas are constructed of.

In This Section

XML Elements, Attributes, and Types
Provides an overview of the common elements that are used in constructing XML Schemas.
Named vs. Unnamed Types and Groups
Describes the different ways of defining types and groups in XML Schemas and when to use one practice over the other.
Creating Elements in XML Schemas
Describes how to use the XML Designer to declare elements.
Creating Attributes in XML Schemas
Describes how to use the XML Designer to declare attributes.
User-Defined Simple Types in XML Schemas
Provides an overview of what simple types are, how facets are used, and how the underlying XML code is visually represented in the XML Designer.
Creating Simple XML Types
Describes how to use the XML Designer to define named simple types.
Creating Unnamed Simple Types
Describes how to use the XML Designer to define unnamed simple types.
Complex Types in XML Schemas
Provides an overview of what complex types are, provides an example demonstrating a typical use of a complex type, and how the underlying XML code is visually represented in the XML Designer.
Creating Complex XML Types
Describes how to use the XML Designer to define named complex types.
Creating Unnamed Complex Types
Describes how to use the XML Designer to define unnamed complex types.
Element Groups
Provides an overview of the different groups in XML Schemas, the reason you would use each group, and how the underlying XML code is visually represented in the XML Designer.
Creating Element Groups in XML Schemas
Describes how to use the XML Designer to define the different types of element groups.
Attribute Groups
Explains what attribute groups are in XML Schemas, and how the underlying XML code is visually represented in the XML Designer.
Creating Attribute Groups in XML Schemas
Describes how to use the XML Designer to define attribute groups.
Keyboard Shortcuts for the XML Designer
Describes how to use the keyboard to navigate the XML designer.

Related Sections

XML Schemas and Data
Introduces topics and sections that describe how to use the XML Designer to work with XML Schemas and datasets.
Creating XML Schemas and Datasets
Introduces topics that describe the different ways that schemas and datasets can be created using the XML Designer as well as describing the connection between datasets and schemas.
Relational Data in XML Schemas
Introduces topics that describe the different options in XML Schemas for representing relational data.
XML Files
Introduces topics that explain how to use the XML Designer when working with XML documents.
Introduction to Data Access with ADO.NET
Provides a high level overview of the most important concepts of data access in ADO.NET.
Introduction to Datasets

Provides an overview of what ADO.NET datasets are, how they can be typed or untyped, what they are used for, and how they are structured.

XML in Visual Studio

Provides links to a variety of topics about XML.

World Wide Web Consortium (W3C) Web site (http://www.w3.org/)

Links to the XML Schema primer and other XML info posted by the W3C.

XML Developer Center on MSDN Online (http://msdn.microsoft.com/xml/default.asp)

Provides various resources about XML.

XML Schema Reference (XSD)

Provides reference topics for schema elements and data types.

XML Designer

Overview of the XML Design tool in Visual Studio.

XML Elements, Attributes, and Types

The basic building blocks of XML Schemas are elements and attributes. Data types define the valid content that elements and attributes contain. When you create XML Schemas, you define the individual elements and attributes and assign valid types to them. Elements describe data, whereas attributes are like properties of an element, in that they provide further definition about the element the way that properties describe characteristics of objects and classes.

Elements

An element describes the data that it contains. Elements can also contain other elements and attributes. For more information, see element.

When an element definition contains additional elements or attributes, it is a complex type. For more information on complex types, see the "Types" section later in this topic.

A basic element definition consists of a name and a data type. The following example shows how to define an element named `quantity`, with a simple type value of integer.

```
<xs:element name="quantity" type="xs:integer" />
```

The element classifies the data — in this case as a quantity. The type defines the valid content that the element contains — in this case an integer.

The following example shows a valid instance of XML data that conforms to the `quantity` element defined in the previous example:

```
<quantity>63</quantity>
```

The following example shows an invalid instance of the quantity element:

```
<quantity>sixty three</quantity>
```

Attributes

An attribute is a named simple-type definition that cannot contain other elements. Attributes can also be assigned an optional default value and they must appear at the bottom of complex-type definitions. Additionally, if multiple attributes are declared, they may occur in any order. For more information, see attribute.

The following code shows how to declare an attribute named `OrderDiscount` that is defined with the simple type `number`. The use of an attribute makes sense here since attributes are optional. If no `OrderDiscount` is provided the XML data will still be valid.

```
<xs:element name="OrderInfo">
   <xs:complexType>
      <xs:sequence>
         <xs:element name="CustomerName" type="xs:string" />
         <xs:element name="OrderNumber" type="xs:positiveInteger" />
         <xs:element name="OrderTotal" type="xs:number" />
      </xs:sequence>
      <xs:attribute name="OrderDiscount" type="xs:number" />
   </xs:complexType>
</xs:element>
```

Naming Elements and Attributes

The following are a few relevant details about elements and attributes:

- XML is case-sensitive. Thus an element named "statecode" is different from an element named "stateCode."
- Attribute values must always be in quotation marks. For more information, see attribute.
- Element names cannot start with a number or underscore, or with the letters "XML."
- Element names cannot contain spaces.

Should I Use an Element or an Attribute?

Elements are used to encapsulate pieces of data, and attributes are generally used to provide accompanying information about an element, rather than to encapsulate raw data itself. Whether you actually use an element or an attribute is up to the needs of your application.

Use attributes if your information requires data of a simple type and:

- Your information requires a default or fixed value.
- Your information requires data that is metadata for an existing element
- If the size of your XML file is an issue, attributes tend to take up fewer bytes than elements

The following list describes the key differences between elements and attributes from a schema perspective:

- A schema can define whether the ordering of elements is significant, but attributes can occur in any order.
- Elements can be nested with the `<choice>` tag, which means one and only one of the listed elements can appear.
- Elements can occur more than once, but attributes cannot.

Types

Types in XML Schemas define the valid data type that elements or attributes may contain, and can be simple or complex. Additionally, types can be named or unnamed. For more information, see Named vs. Unnamed Types and Groups.

Simple Types

There are two major categories of simple types:

- *Built-in types*, which are defined by the World Wide Web Consortium's XML Schema specification - for example, **string, boolean**, and **float**.

  Built-in types include both primitive types and derived types.

  Primitive data types are not derived from other data types. For example, float is a mathematical concept that is not derived from other data types. For more information, see Primitive XML Data Types.

  Derived data types are defined in terms of existing data types. For example, integer is a special case derived from the decimal data type. For more information, see Derived XML Data Types.

- *User-defined simple types* are derived from the W3C built-in types by applying user-defined values to items called facets. For more information, see User-Defined Simple Types in XML Schemas.

Facets

Facets constrain the acceptable values of simple types. User-defined simple types can be created by applying facets. For more information, see Data Type Facets.

In the following example the `maxInclusive` facet is applied to a simple type named `qtyLimiter` in order to restrict the acceptable values of the `positiveInteger` type to quantities from 1 through 100:

```
<xs:simpleType name="qtyLimiter">
   <xs:restriction base="xs:positiveInteger">
      <xs:maxInclusive value="100" />
   </xs:restriction>
</xs:simpleType>
```

Complex Types

Complex types are element definitions that can include other elements, attributes, and groups. An important difference between complex and simple types is that complex types can include elements and attributes declared as simple or complex types, whereas simple types cannot include other elements, attributes, or groups but can only include facets. For more information, see Complex Types in XML Schemas.

See Also

element | attribute| Named vs. Unnamed Types and Groups | User-Defined Simple Types in XML Schemas | Element Groups | Attribute Groups |Creating XML Schemas and Datasets | Relational Data in XML Schemas |
Introduction to Data Access with ADO.NET | Introduction to Datasets

Named vs. Unnamed Types and Groups

Data types and groups in XML Schemas can be either named or unnamed. If you plan to reuse a data type within your schema, create a named type and refer to it in all the elements that require that data type. Named types can also make your schema easier to read.

Named Types

A named type is a *globally declared* data type that has a **name** attribute. Globally declared elements are direct children of the `<schema>` element, as opposed to being nested within other element definitions. The following example shows how to globally declare the complex type `usAddress`. The element `customerInfo` uses `usAddress` twice: as part of BillTo and ShipTo elements. The elements `name`, `city`, `ShipTo`, and so on, are all declared nested within definitions.

```
<?xml version="1.0" encoding="utf-8" ?>
<xs:schema id="XMLSchema1" xmlns:xs="http://www.w3.org/2001/XMLSchema">
   <xs:complexType name="usAddress">
      <xs:sequence>
         <xs:element name="name" type="xs:string" />
         <xs:element name="street" type="xs:string" />
         <xs:element name="city" type="xs:string" />
         <xs:element name="state" type="xs:string" />
         <xs:element name="zip" type="xs:string" />
      </xs:sequence>
   </xs:complexType>
   <xs:element name="customerInfo">
      <xs:complexType>
         <xs:sequence>
            <xs:element name="Name" type="xs:string" />
            <xs:element name="ShipTo" type="usAddress" />
            <xs:element name="BillTo" type="usAddress" />
         </xs:sequence>
      </xs:complexType>
   </xs:element>
</xs:schema>
```

In the XML Designer named types are declared by dragging a **simpleType**, **complexType,** or **group** from the XML Schema tab of the Toolbox onto the design surface. It is important to note that when creating unnamed types, the data type is defined within the design grid of an element, not within a **simpleType** or **complexType**.

Unnamed Types

An unnamed type is a data type that is nested (in-line) within the definition of an element. You could use an unnamed type when a type is only needed once within your schema.

In contrast with the preceding example, the following example shows how to define an address as an unnamed type — nested within the `ShipTo` element. In this schema, you know that you only need one address.

```
<?xml version="1.0" encoding="utf-8" ?>
<xs:schema id="XMLSchema1" xmlns:xs="http://www.w3.org/2001/XMLSchema">
   <xs:element name="customerInfo">
      <xs:complexType>
         <xs:sequence>
            <xs:element name="Name" type="xs:string" />
            <xs:element name="ShipTo">
               <xs:complexType>
                  <xs:sequence>
                     <xs:element name="name" type="xs:string" />
                     <xs:element name="street" type="xs:string" />
                     <xs:element name="city" type="xs:string" />
                     <xs:element name="state" type="xs:string" />
                     <xs:element name="zip" type="xs:string" />
                  </xs:sequence>
               </xs:complexType>
            </xs:element>
         </xs:sequence>
```

```
        </xs:complexType>
    </xs:element>
</xs:schema>
```

See Also

XML Schemas and the XML Designer| XML Elements, Attributes, and Types | User-Defined Simple Types in XML Schemas | Complex Types in XML Schemas | Element Groups | Attribute Groups | XML Namespaces | Creating XML Schemas and Datasets | Relational Data in XML Schemas | XML Files

Creating Elements in XML Schemas

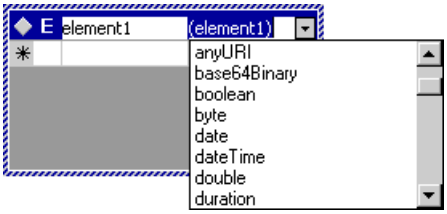
Elements are the basic building blocks of XML Schemas. Declaring an element involves associating a name with a data type. For more information, see XML Elements, Attributes, and Types.

For an overview of where elements fit into the process of creating an XML Schema, see
Creating XML Schemas: High-Level Process for a summary of the steps, and
Walkthrough: Creating an XML Schema with the XML Designer for step-by-step instructions for a representative scenario: creating a purchase-order schema.

Elements can be added to the following:

- Schema
- Other elements
- ComplexTypes
- Groups

Visual representation of an element in the XML Designer



To create an element with the XML Designer

1. Add an **XML Schema** or **Dataset** to your project. For more information, see Adding New Schemas.
2. Drag an **element** from the **XML Schema** tab of the Toolbox and do the following:

   - If you want to add an element to an existing element, complex type, or group: drop the **element** on its design grid.

     The element will appear as a design grid below the existing grid, showing the hierarchical (nested) relationship between them.

     - or -

   - If you want the new element to have global scope: drop the element on the design surface.

     A design grid appears.

3. In the grid header, select **element1** and type a new value for the name of the element. For more information, see the "Naming Elements and Attributes" section of XML Elements, Attributes, and Types.
4. In the cell to the right of the element name choose a data type from the drop-down list.
5. In the Properties window set the desired properties for your element, such as **minOccurs**, **maxOccurs**. For more information, see element.

See Also

element | XML Schemas and the XML Designer | XML Elements, Attributes, and Types | Named vs. Unnamed Types and Groups | User-Defined Simple Types in XML Schemas | Complex Types in XML Schemas | Creating XML Schemas and Datasets | Relational Data in XML Schemas | XML Files

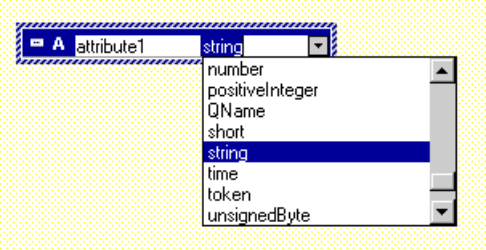
Creating Attributes in XML Schemas

Attributes are like properties of an element. When you declare an attribute, you name it and associate the name with a simple data type. Attribute declarations are defined as simple types. For more information, see XML Elements, Attributes, and Types.

Attributes can be added to the following:

- Schema
- Elements
- AttributeGroups
- ComplexTypes

Visual representation of an attribute in the XML Designer



To create an attribute with the XML Designer

1. Add an **XML Schema** or **Dataset** to your project. For more information, see Adding New Schemas.
2. In Schema view, drag an **attribute** from the **XML Schema** tab of the Toolbox to the XML Designer.

   - If you want to create an attribute with global scope within the schema: drop the attribute on the design surface.

     A design grid appears.

     - or -

   - If you want to add an attribute to an existing element, attributeGroup, or complexType, drop the attribute on the element.

     The attribute appears as a row within the design grid.

3. In the design grid, select **attribute1** and type a new value for the name of the attribute. For more information, see the "Naming Elements and Attributes" section of XML Elements, Attributes, and Types.
4. In the cell to the right of the attribute name, choose a data type from the drop-down list.

See Also

attribute | XML Schemas and the XML Designer| XML Elements, Attributes, and Types | Named vs. Unnamed Types and Groups | User-Defined Simple Types in XML Schemas | Complex Types in XML Schemas | Creating XML Schemas and Datasets | Relational Data in XML Schemas | XML Files

User-Defined Simple Types in XML Schemas

User-defined simple types are restrictions put on the content of elements or attributes. A user-defined simple type will always restrict the contents of an element to a subset of the existing simple type it was derived from. For example, `string` is a simple type. You can apply the `length` facet with a value of 2 to restrict data content to a two-character string. The existing simple type can be either one of the built-in types or a user-defined simple type already declared in your XML Schema.

If an element only needs to be a subset of an existing built-in data type, then a user-defined simple type should be used. Additionally, if this type of element definition is only needed once in your schema, an unnamed simple type should be used. If you need several occurrences of the same type, use a named simple type.

Simple types are defined using the **simpleType** element and cannot contain subelements or attributes. There are over 40 simple types built into the XML Schema definition of the World Wide Web Consortium (W3C). For more information, see Primitive XML Data Types.

Restricting an existing simple type is accomplished by applying facets. Facets define a set of legal values for a simple type. For more information, see Data Type Facets.

The following example shows how to define your own simple type and apply the `maxInclusive` facet to it.

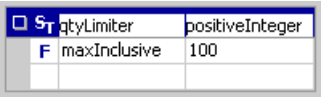
The following defines an element named `quantity` that is of the simple built-in type **positiveInteger**:

```
<xs:element name="quantity" type="xs:positiveInteger" />
```

In order to conform to the above declaration, the valid values of the `quantity` element would be limited to any positive integer. If you needed to limit the `quantity` element to a value less than 101, you would create a user-defined simple type. The following example shows how to create a simple type named `qtyLimiter` and apply the **maxInclusive** facet with a value of 100.

```
<xs:simpleType name="qtyLimiter">
   <xs:restriction base="xs:positiveInteger">
      <xs:maxInclusive value="100" />
   </xs:restriction>
</xs:simpleType>
```

This is how the previous code is represented visually in the XML Designer:



This user-defined simple type can now be applied to element declarations within the same XML Schema (.xsd file). The following example shows how to declare a `quantity` element with the **qtyLimiter** type.

```
<xs:element name="quantity" type="qtyLimiter" />
```

If there were elements other than `quantity` that had the same content requirements, they could be declared with the user-defined **qtyLimiter** type as well.

See Also

XML Schemas and the XML Designer| XML Elements, Attributes, and Types | Named vs. Unnamed Types and Groups | Complex Types in XML Schemas | Creating XML Schemas and Datasets | Relational Data in XML Schemas | XML Files

Creating Simple XML Types

You can define simple types in your XML Schema. Simple types allow you to create new types by adding restrictions (such as limiting the length of a data entry) to existing types. For example, you could create a stateCode simple type that is limited to two characters. These new types can then be applied to elements and attributes.

You restrict the contents of a simple type by applying facets to the type. Facets set limits on the type of content a simple type can contain.

> **Tip** If you need to define an element that contains subelements or attributes, create a complex type. For details, see Creating Complex XML Types.

To create a new simple type

1. Add an **XML Schema** or **DataSet** to your project. For details, see Adding New Schemas.
2. Drag a **simpleType** object from the **XML Schema** tab of the **Toolbox** to the design surface.

   A grid appears in the designer.
3. In the edit box in the grid header, select **simpleType1** and type a new value to rename the object.
4. In the drop-down list box in the grid header, select the appropriate data type.
5. If you want to add restrictions, set the facets you want to use, following these steps:
   a. In the first blank row of the grid, click in the first cell and choose the facet you want to set from the drop-down list box.
   b. In the second cell for that same row, specify the appropriate value. For details, see Data Type Facets.

You can now use this type in defining new elements and complex types.

See Also

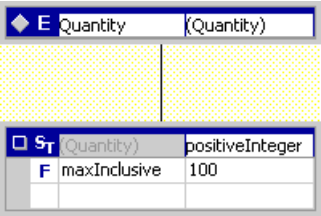
Creating XML Schemas: High-Level Process | User-Defined Simple Types in XML Schemas | Walkthrough: Creating an XML Schema with the XML Designer | simpleType Element

Creating Unnamed Simple Types

An unnamed type is a data type that is nested (in-line) within the definition of an element. Use an unnamed type when a type is only needed once within your schema.

Unnamed simple types are constructed of built-in types and contain only facets to restrict their contents. For more information, see User-Defined Simple Types in XML Schemas.

Visual representation of an unnamed simple type in the XML Designer



To create an element or attribute defined as an unnamed simpleType

1. Add an **XML Schema** or **Dataset** item to your project. For more information, see Adding New Schemas.
2. Drag an **element** or **attribute** from the **XML Schema** tab of the Toolbox to the design surface. For more information about whether to use an element or attribute, see the "Should I use an Element or an Attribute?" section of XML Elements, Attributes, and Types.

   A design grid appears in the designer.

3. In the grid header, select the default name (element1 or attribute1) and type a new name for the element or attribute.
4. In the drop-down list box in the grid header, select **Unnamed simpleType**.

   A simpleType design grid appears below the original grid, which represents that it has a child relationship.

5. In the simpleType design grid, select a data type in the drop-down list to the right of the disabled name box.

   This will be the base data type that this simple type will be derived from.

6. In the first blank row of the grid, click the cell below the disabled name box and choose a facet from the drop-down list. For more information, see Data Type Facets.

See Also

simpleType Element | XML Schemas and the XML Designer | XML Elements, Attributes, and Types | Named vs. Unnamed Types and Groups | User-Defined Simple Types in XML Schemas | Complex Types in XML Schemas | Creating XML Schemas and Datasets | Relational Data in XML Schemas | XML Files

Complex Types in XML Schemas

Complex types are user defined data types that can include other elements or attributes. Complex types can contain elements defined as either simple or complex. Complex types can also include attributes and groups, whereas simple types can only contain facets.

Complex types are defined using the **complexType** element and typically contain combinations of element, attribute, and group declarations, as well as references to globally declared elements and groups. A complex type can be thought of as a mini-schema that defines the valid structure and data contained within a specific element.

Although complex types can be either named or unnamed, the following examples demonstrate named complex types. For more information about unnamed types, see Named vs. Unnamed Types and Groups.

Typical Use of a Complex Type

A common scenario where a complex type would be used is as part of a schema that validates a purchase order by requiring a shipping address and a billing address. You would define a complex type that represents an address and declare shipTo and billTo elements of that complex type.

The following example shows how to create a complex type named `usAddress`:

```
<xs:complexType name="usAddress">
   <xs:sequence>
      <xs:element name="name" type="xs:string" />
      <xs:element name="street" type="xs:string" />
      <xs:element name="city" type="xs:string" />
      <xs:element name="state" type="xs:string" />
      <xs:element name="zip" type="xs:string" />
   </xs:sequence>
</xs:complexType>
```

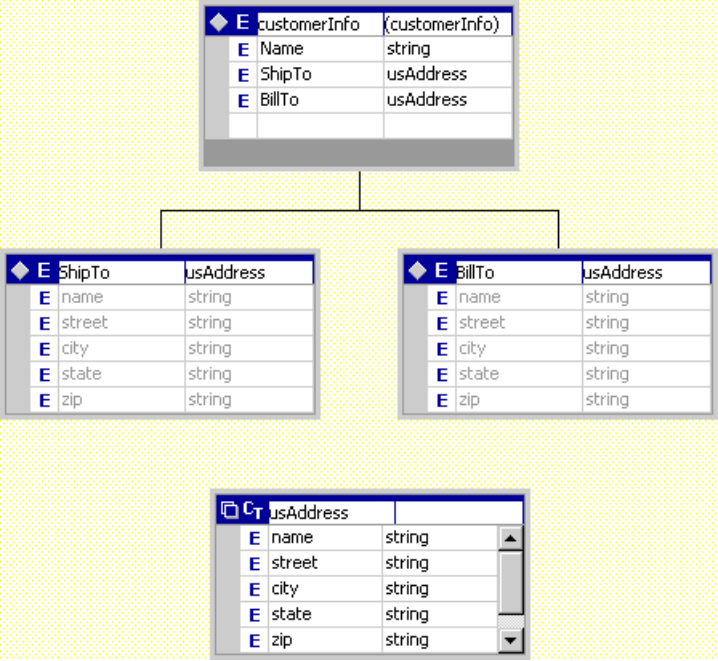
An element declared with the `usAddress` type would look like this:

```
<xs:element name="ShipTo" type="usAddress" />
```

The following example shows how to declare multiple elements using your defined complex type:

```
<xs:element name="customerInfo">
  <xs:complexType>
     <xs:sequence>
        <xs:element name="Name" type="xs:string" />
        <xs:element name="ShipTo" type="usAddress" />
        <xs:element name="BillTo" type="usAddress" />
     </xs:sequence>
  </xs:complexType>
</xs:element>
```

This is a visual representation of the preceding code in the XML Designer. Notice the globally declared complex type named `usAddress` at the bottom, and how the `ShipTo` and `BillTo` elements are defined with the `usAddress` type in the `customerInfo` element at the top.



See Also

Walkthrough: Creating an XML Schema with the XML Designer | World Wide Web Consortium (http://www.w3.org/) | XML Schemas and the XML Designer | XML Elements, Attributes, and Types | Named vs. Unnamed Types and Groups | User-Defined Simple Types in XML Schemas | Creating XML Schemas and Datasets | Relational Data in XML Schemas | XML Files

Creating Complex XML Types

Complex types allow you define elements with attributes or subelements in your XML Schema. For example, you could define an addressType made up of name, address, and postal code. This new type could then be used by an Address element.

To create a new complexType object

1. Add an **XML Schema** or **DataSet** to your project. For details, see Adding New Schemas.
2. Drag a **complexType** object from the **XML Schema** tab of the **Toolbox** onto the design surface.
3. In the edit box in the grid header, select **complexType1** and type a new value to rename the object.
4. To specify the elements for the complex type, follow these steps:
   a. In the first empty row in the grid, select the small cell next to the asterisk to access the element type drop-down list box.
   b. Choose the type of element from the list.
   c. In the second cell, replace the default name with a name for the element you chose.
   d. In the third cell, select the type for the element. You can choose an existing type, an unnamed type, or a type you have created.
5. Continue adding elements to the complex type until it is fully declared.

To delete an element from a complexType

- Right-click the row you want to delete, and then click **Delete** on the shortcut menu.

To delete an entire complexType

- Right-click the header of the grid, and then click **Delete**.

See Also

Creating XML Schemas: High-Level Process | Complex Types in XML Schemas|
Walkthrough: Creating an XML Schema with the XML Designer | complexType Element

Creating Unnamed Complex Types

An unnamed complex type is a user defined type definition that is constructed in-line as part of the content model definition (as opposed to a global declaration being named and explicitly referenced). Unnamed types are typically used when a particular content model is needed to describe a single occurrence of an element. If you need to reuse a type definition then you should create a named type. For more information, see Named vs. Unnamed Types and Groups.

Unnamed complex types can be created with the XML Designer by adding elements and attributes to the design grid of an existing element.

To gain an understanding of where unnamed complex types fit into the overall process of creating an XML Schema, see Creating XML Schemas: High-Level Process for a summary of the steps and Walkthrough: Creating an XML Schema with the XML Designer for step-by-step instructions for a representative scenario: creating a purchase-order schema.

To create a new unnamed complexType

1. Add an **XML Schema** or **Dataset** item to your project. For more information, see Adding New Schemas.
2. Drag an **element** from the **XML Schema** tab of the Toolbox onto the design surface.

   > **Note** An unnamed type is defined inline as part of an element declaration. On the design surface, you create an unnamed complex type by building up the content model in the design grid of an element.

3. In the grid header, select **element1** and type in a new name for the element.
4. Choose **unnamed complex type** from the list in the grid header.
5. To define the content of an unnamed complex types, follow these steps:
   a. In the first empty row in the grid, select the small cell next to the asterisk to access the element type drop-down.
   b. Choose the type of element from the list.
   c. Use the TAB key to navigate to the next cell and replace the default name with a new name for this element.
   d. In the third cell, select the type for the element. You can choose an existing type, an unnamed type, or an existing user-defined type.

      > **Note** If you choose an unnamed type, a child element will appear on the design surface. Follow the procedure in step 5 to define that type as well.

6. Repeat Step 5 until the type is fully defined.

See Also

Walkthrough: Creating an XML Schema with the XML Designer | complexType Element | XML Schemas and the XML Designer | XML Elements, Attributes, and Types | Named vs. Unnamed Types and Groups | User-Defined Simple Types in XML Schemas | Complex Types in XML Schemas | Relational Data in XML Schemas | XML Files

Element Groups

Groups of elements can be defined in order to control how the individual elements within a group occur in XML data. For more information, see Creating Element Groups in XML Schemas.

You use element groups when you want to impose the following occurrence constraints on a group of elements:

Use the:	Occurrence Constraint:
sequence group	all elements occur in the exact order
choice group	only one element in the group occurs
all group	all elements occur or none at all

Groups can be named or unnamed. For more information, see Named vs. Unnamed Types and Groups.

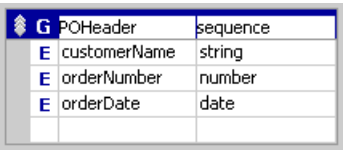
Sequence Group

Grouping with the sequence element declares that subelements must appear in the exact order as indicated. The following example shows how to define a group named `POHeader`. By declaring the group of subelements within the `<xs:sequence>` element, any XML data validated by this schema must contain the `customerName`, `orderNumber`, and `orderDate` elements in the exact order.

The following XML Schema code defines a **sequence group** named `POHeader`:

```
<xs:group name="POHeader">
   <xs:sequence>
      <xs:element name="customerName" type="xs:string" />
      <xs:element name="orderNumber" type="xs:number" />
      <xs:element name="orderDate" type="xs:date" />
   </xs:sequence>
</xs:group>
```

Visual representation of a sequence group named `POHeader` in the XML Designer



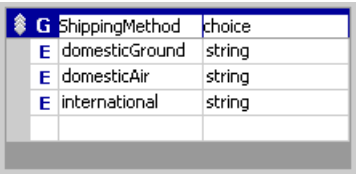
Choice Group

Grouping with the choice element declares that only one of the subelements can appear in XML data. The following example shows how to define a choice group named `ShippingMethod`. By declaring subelements within the `<xs:choice>` element, any XML data validated by this schema must contain only one shipping method.

Sample bit of XML Schema code defining a **choice group** named `ShippingMethod`:

```
<xs:group name="ShippingMethod">
   <xs:choice>
      <xs:element name="domesticGround" type="xs:string" />
      <xs:element name="domesticAir" type="xs:string" />
      <xs:element name="international" type="xs:string" />
   </xs:choice>
</xs:group>
```

Visual representation of a choice group named `ShippingMethod` in the XML Designer



All Group

Grouping with the **all** element declares that an instance document can contain all or none of the subelements, and they may appear in any order.

The following example declares a group named `PaymentInfo`. By declaring subelements within the `<xs:all>` element, any XML data validated by this schema can contain either all the elements, in any order, or none. (If no payment were made, there would not be any payment info at all).

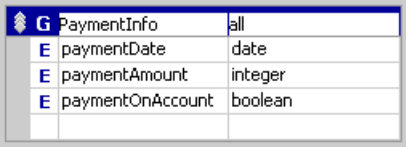
Additionally the **All** group has the following specific rules governing its usage:

- The all group must be at the top level of any content model.
- The subelements must be individual elements (no groups).
- It must be the sole child at the top of the content model. This means that no subelements are allowed outside the group tag.
- No element in the content model may appear more than once. This is equivalent to setting the values of minOccurs=0 and maxOccurs=1.

The following XML Schema code defines an **all group** named `PaymentInfo`:

```
<xs:group name="PaymentInfo">
   <xs:all>
      <xs:element name="paymentDate" type="xs:date" />
      <xs:element name="paymentAmount" type="xs:integer" />
      <xs:element name="paymentOnAccount" type="xs:boolean" />
   </xs:all>
</xs:group>
```

Visual representation of an `all` group named PaymentInfo in the XML Designer



See Also

XML Schemas and the XML Designer| XML Elements, Attributes, and Types | Named vs. Unnamed Types and Groups | User-Defined Simple Types in XML Schemas | Complex Types in XML Schemas | Creating XML Schemas and Datasets | Relational Data in XML Schemas | XML Files

Creating Element Groups in XML Schemas

Elements can be constructed into groups in order to control the sequence and occurrence behavior of the elements within the group. For more information, see Element Groups.

Create a group when you need to have:

- All elements occur in the exact order (sequence group).
- Only one element in the group occur (choice group).
- All elements occur, or none at all (all group).

Additionally, groups can be either named or unnamed. You should have a good understanding of the difference between named and unnamed definitions before creating a group. For more information, see Named vs. Unnamed Types and Groups.

For an overview of where element groups fit into the process of creating an XML Schema, see Creating XML Schemas: High-Level Process for a summary of the steps and Walkthrough: Creating an XML Schema with the XML Designer for step-by-step instructions for a representative scenario: creating a purchase-order schema.

Creating Named Groups

Named groups are declared globally and explicitly named. This provides the means for reusing the group in multiple complex type definitions.

To create a named group with the XML Designer

1. Add an **XML Schema** or **Dataset** to your project. For more information, see Adding New Schemas.
2. Drag a **group** from the **XML Schema** tab of the Toolbox to the design surface.

   A design grid appears.

3. In the grid header, select **group1** and type a new name for the group.
4. In the cell to right of the group name, do one of the following:

   - If you want all elements occur in the exact order, select **sequence.**

     - or -

   - If you want only one element in the group occur, select **choice**.

     - or -

   - If you want all or none of the elements occur, select **all**.

5. To define the contents of the group, follow these steps:
   a. In the first empty row in the grid, select the small cell to the right of the asterisk to access the drop-down list.
   b. Use the TAB key to navigate to the next cell and replace the default name with a new name for this element.
   c. In the third cell, select the data type for the element.
6. Continue adding elements to the group until it is fully defined.

Creating Unnamed Groups

An unnamed group is defined as an inclusive part of the content model of a complex type definition, as opposed to being named and explicitly referenced. For more information, see Named vs. Unnamed Types and Groups.

Unnamed sequence groups are actually created by default with complex type definitions. If you drag an element onto the design surface and switch to XML view, you will see the `<sequence />` element within the content model. You just created an unnamed sequence group.

The actual grouping option is defined by setting the **Order** property as defined in step 8 below.

To create an unnamed group with the XML Designer

1. Drag an **element** from the Toolbox onto the design surface.

   A design grid appears.

2. In the grid header, select **element1** and type in a new name for the element.
3. To define the contents of the group, follow these steps:
    a. In the first empty row in the grid, select the small cell to the right of the asterisk to access the drop-down list.
    b. Choose the type of element from the list. For more information, see XML Schema Elements.
    c. Use the TAB key to navigate to the next cell and replace the default name with a new name for this element.
    d. In the third cell, select a data type for the element.
4. Continue adding elements to the group until it is fully defined.
5. In the Properties window, access the drop-down list and select the element you just created.
6. Expand the **complexType** section.
7. Locate the **Order** property.
8. Set the property to the desired group type.

    - If you want all elements to occur in the exact order, set the **Order** property to **sequence**.

      - or -

    - If you want only one element in the group to occur, set the **Order** property to **choice**.

      - or -

    - If you want all or none of the elements to occur, set the **Order** property to **all**.

See Also

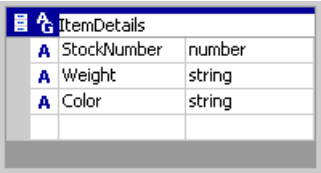
XML Schemas and the XML Designer | XML Elements, Attributes, and Types | Named vs. Unnamed Types and Groups | Element Groups | Attribute Groups | Creating XML Schemas and Datasets | Relational Data in XML Schemas | XML Files

Attribute Groups

Attributes can be arranged into a named group that can be included as part of a complex type definition. Attribute groups are defined and named in one place so that they can be reused and easily updated.

The following example shows how to define an attribute group named `ItemDetails` that is comprised of three attributes: `StockNumber`, `Weight`, and `Color`.

Visual representation of an attribute group named `ItemDetails` in the XML Designer



The following example shows the XML code that represents the attribute group illustrated above:

```
<xs:attributeGroup name="ItemDetails">
      <xs:attribute name="StockNumber" type="xs:number" />
      <xs:attribute name="Weight" type="xs:string" />
      <xs:attribute name="Color" type="xs:string" />
</xs:attributeGroup>
```

See Also

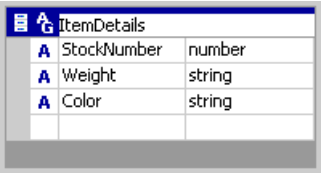
attributeGroup | XML Schemas and the XML Designer | XML Elements, Attributes, and Types | Creating XML Schemas and Datasets | Relational Data in XML Schemas | XML Files

Creating Attribute Groups in XML Schemas

An attribute group allows you to reuse the same set of attributes within your schema. For more information, see Attribute Groups.

To gain an understanding of where creating attribute groups fits into the overall process of creating an XML Schema, see Creating XML Schemas: High-Level Process for a summary of the steps and Walkthrough: Creating an XML Schema with the XML Designer for step-by-step instructions for a representative scenario: creating a purchase-order schema.

Visual representation of an attribute group in the XML Designer



To define an attributeGroup with the XML Designer

1. Add an **XML Schema** or **Dataset** to your project. For more information, see Adding New Schemas.
2. In **Schema** View, drag an **attributeGroup** from the **XML Schema** tab of the Toolbox to the design surface.

   A design grid appears.

3. In the grid header, select **attributeGroup1** and type a new value for the name of the **attributeGroup**.
4. In the next empty row in the grid, select the small cell next to the asterisk and choose the desired attribute type to include in the attribute group from the drop-down list:

   - If you want to include any attribute from the specified namespace in this attribute group, choose **anyAttribute**.
   - If you want to include a standard attribute in this attribute group definition, choose **attribute**.
   - If you want to include some other attribute group in this attribute group definition, choose **attributeGroup**.

5. Use the following table to complete the attribute definition in the design grid based on your selection in step 4:

Attribute Type (step 4)	**Name field**	**Type field**
anyAttribute	No input	No input
attribute	Type a name for the attribute	Select a type from the drop-down box
attributeGroup	Type 'ref=*x*' where *x* is the name of a top level attribute group you want to include	No input

6. Repeat steps 4 and 5 until your attribute group is fully defined.

See Also

attributeGroup | XML Schemas and the XML Designer | Adding New Schemas | XML Elements, Attributes, and Types | Element Groups | Attribute Groups | Relational Data in XML Schemas | XML Files

Keyboard Shortcuts for the XML Designer

The following tables list the keyboard shortcuts that can be used for navigation while working with the XML Designer:

Navigating the Environment

Action	Shortcut
Move focus to the Toolbox	CTRL+ALT+X
Move focus to the design surface	ALT+W, then select the number corresponding to the window you want
Move focus to Solution Explorer	CTRL+ALT+L
Move focus to the Properties Window	F4
Move focus to the Menu bar	ALT
Move focus to Server Explorer	CTRL+ALT+S

Navigating in Schema View

To select a specific grid in **Schema** view, press **F4** to navigate to the Properties window. Press TAB until the main drop-down list is selected. Arrow to the desired object and press ENTER.

Action	Shortcut
Move from grid to grid	TAB
Move from grid to grid	Arrow keys
Place selected grid into edit mode	ENTER
Move selected grid in the direction of the ARROW key pressed	CTRL+Arrow keys
Toggles between **XML Source** and **Schema** view	CTRL+PAGEDOWN

Navigating within an Element Grid in Schema View

Action	Shortcut
Move forward from cell to cell (including header row)	TAB
Move from cell to cell (excluding the header row) if cells are empty	Arrow keys
Expand the drop-down list in cells that have them	ALT+Arrow key
Exit edit mode and return to navigating the designer	ESC

Navigating the Toolbox

Action	Shortcut
Move up through the list of items	UP ARROW
Move down through the list of items	DOWN ARROW
Cycle through the list of items	TAB

See Also

Keyboard, Environment, Options Dialog Box | XML Designer | XML Schema Tab, Toolbox

Creating XML Schemas and Datasets

ADO.NET typed datasets are classes that are generated from XML Schemas (.xsd file). As changes are made to the schema file the definition of the dataset class changes as well. When working with XML Schemas and typed dataset representations in Visual Studio at design time there is essentially no difference. They are both .xsd files in the XML Designer, the difference being that typed datasets have an associated class file and a predefined document (or root) node that represents the encompassing dataset.

A schema serves these functions:

- It describes the "shape" of the XML document — for example, whether the XML document is constructed hierarchically or in a keyed relationship.
- It validates data being imported from an XML stream or document into a dataset.
- It establishes the relational structure of the dataset's tables and columns, the key columns, constraints, relationships between tables, and so on; the information is used when generating a DataSet Class. In this respect, the schema serves a special role that requires it to support dataset-specific features.

For more information, see Introduction to Datasets.

In This Section

Relationship Between XML Schemas and Datasets
Provides an overview of how schemas are used to define datasets, as well as links to more dataset information.
Creating XML Schemas: High-Level Process
Provides an overview of the process of defining an XML Schema, including links to the various steps involved.
Adding New Schemas
Describes how to add new XML Schemas and Datasets to a project and what actual files are included with each option.
Creating XML Schemas and Datasets from Existing Data
Describes how to define an XML Schema based on data that can be accessed through Server Explorer.
Creating New Typed Datasets with the XML Designer
Details the steps involved when creating a schema that will represent relational data.
Creating ADO.NET Typed DataSets from Schemas
Describes how to use the XML Designer to generate typed datasets from existing XML Schemas.
Importing External Elements into XML Schemas
Describes how to add references to other schemas in order to use globally declared data types from other schemas.
XML Namespaces
Provides an overview of how namespaces are used in XML Schemas.
Changing the Target Namespace
Explains how the XML Designer provides a default namespace value, and the steps involved in adding a valid namespace Uniform Resource Identifier (URI) to your schema.
Walkthrough: Creating an XML Schema with the XML Designer
Provides step-by-step instructions for how to create an XML Schema that represents a purchase order.
Walkthrough: Reading XML Data into a Dataset
Shows how to load XML data into a dataset, display it in a Windows Forms **DataGrid** control, then display an XML Schema for the XML file in a text box.

Related Sections

XML Schemas and Data
Introduces topics and sections that describe how to use the XML Designer to work with XML Schemas and datasets.
XML Schemas and the XML Designer
Introduces topics that describe the different elements in XML Schemas and how they are created using the XML Designer.
Relational Data in XML Schemas
Introduces topics that describe the different options in XML Schemas for representing relational data.
XML Files
Introduces topics that explain how to use the XML Designer when working with XML documents.
DataSet Class
Provides information on datasets.
Introduction to Data Access with ADO.NET
Provides a high level overview of the most important concepts of data access in ADO.NET.
Introduction to Datasets
Provides an overview of what ADO.NET datasets are, how they can be typed or untyped, what they are used for, and how they

are structured.

XML in Visual Studio

Provides links to a variety of topics about XML.

World Wide Web Consortium (W3C) Web site (http://www.w3.org/)

Links to the XML Schema primer and other XML info posted by the W3C.

XML Developer Center on MSDN Online (http://msdn.microsoft.com/xml/default.asp)

Provides various resources about XML.

XML Schema Reference (XSD)

Provides reference topics for schema elements and data types.

XML Designer

Provides an overview of the XML Design tool in Visual Studio.

Relationship Between XML Schemas and Datasets

A typed ADO.NET dataset is a class with an associated XML Schema, as opposed to an untyped dataset that is not a class and has no associated schema. The typed dataset class is derived from a dataset class, which inherits all of the methods, events and properties of a dataset. Additionally, a typed dataset provides strongly typed methods, events, and properties meaning you can access tables and columns by name, instead of iterating through collections. A typed dataset also incorporates table and column names into the statement completion feature of Visual Studio. This facilitates code that is both easier to write as well as read. Additionally, with a typed dataset, type mismatch errors are caught when the code is compiled rather than at run-time.

When a typed dataset is generated at design time, a dataset class and an associated XML Schema is created in your project. XML Schemas define and validate the data being imported from XML streams or documents into typed datasets. The XML Schema associated with the dataset can be loading into the XML Designer providing a visual representation of the dataset structure that can be edited and viewed.

> **Note** An untyped ADO.NET dataset is a dataset that does not have an associated XML Schema. Additionally, at design time, datasets do not contain any data. Since an untyped dataset contains no data or structure, XML Schemas cannot be generated from untyped datasets at design time. During run time, once the dataset has been populated with data, an XML Schema based on that data could be generated programmatically.

XML Schemas in Visual Studio conform to the specification for schemas published by the World Wide Web Consortium (W3C). Schema elements, whether created to validate XML documents or to define associated dataset classes, are all defined in accordance with the W3C schema specification. In situations where schema syntax is not sufficient to describe certain features of ADO.NET datasets, custom attributes are used to define ADO.NET specific characteristics. While viewing a schema in XML view these custom attributes can be identified with the msdata: qualifier. However, it is important to note that the W3C spec permits custom attributes, which will be ignored by applications that do not process ADO.NET datasets.

For more reading on	See
The relationship between datasets and XML Schemas	Introduction to Data Access with ADO.NET
Creating datasets from schemas	Creating ADO.NET Typed Datasets from Schemas
.NET Framework reference (syntax and class members)	DataSet Class
XML Schemas	XML Schema Reference (XSD) XML Schema Primer on the World Wide Web Consortium Web Site (http://www.w3.org/)

See Also

Creating XML Schemas and Datasets | Creating XML Schemas and Datasets from Existing Data | Creating New Typed Datasets with the XML Designer | Relational Data in XML Schemas | XML Files | XML Schemas and Data | Introduction to Data Access with ADO.NET | Introduction to Datasets

Creating XML Schemas: High-Level Process

There are several steps involved in creating an XML Schema (.xsd file) from scratch. First, you must examine the data needs of your organization and map out the structure of your data to determine the types, elements, and relationships you will need in the schema. After you have planned your schema out, you can quickly create it using the XML Designer. For an example of how to create a schema, see Walkthrough: Creating an XML Schema with the XML Designer.

> **Note** In addition to creating a schema from scratch, you can infer schemas from existing XML documents (.xml files), and then edit them as needed in the XML Designer. For more information, see Creating XML Schemas from XML Files.

To create an XML Schema

1. Determine the design needed for your schema.
2. Add a schema to your project. For more information, see Adding New Schemas.
3. Define the types that will be used in your schema. For more information, see any of the following topics:

For instructions on	See
Creating a simple named type	Creating Simple XML Types
Creating a complex named type	Creating Complex XML Types
Creating a simple unnamed type	Creating Unnamed Simple Types
Creating a complex unnamed type	Creating Unnamed Complex Types

4. Add the appropriate elements to the schema, making use of the types you have defined.

For instructions on	See
Creating an element definition	Creating Elements in XML Schemas
Creating an attribute definition	Creating Attributes in XML Schemas
Grouping sets of elements	Creating Element Groups in XML Schemas

See Also

Adding New Schemas | Creating Simple XML Types | Creating Complex XML Types | Creating Unnamed Simple Types | Creating Unnamed Complex Types | Creating Elements in XML Schemas | Creating Attributes in XML Schemas | Creating Element Groups in XML Schemas

Adding New Schemas

XML Schemas are used to ensure consistency among XML documents (.xml files) by describing the structure of data. To create new XML Schemas in the XML Designer, start by adding a dataset or schema to your project, depending on your needs:

- If you add a DataSet item from the Add New Item dialog box, the output will be both a schema (an .xsd file) and an automatically generated dataset.
- If you add an XML Schema item, the output will be only a schema file. You can manually generate a dataset from the schema.

For details about the relationship between XML Schemas and datasets, see Introduction to Data Access with ADO.NET.

To add a new schema to the project

1. From the **Project** menu, choose **Add New Item**.
2. Do one of the following:

    - To add a schema, access the appropriate folder and double-click **XML Schema.**

      An XML Schema file (.xsd file) is added to your project.

    - To add a dataset, access the appropriate folder and double-click **DataSet**.

      An XML Schema file and typed **DataSet** class file (.vb or .cs file) are added to your project.

      Note You may have to click the **Show All Files** button in Solution Explorer to see the files that have been generated.

3. To create the schema, add elements and attributes. For details, see Creating XML Schemas: High-Level Process.

See Also

Walkthrough: Creating an XML Schema with the XML Designer | Introduction to XML Schemas| Relational Data in XML Schemas | Creating ADO.NET Typed DataSets from Schemas

Creating XML Schemas and Datasets from Existing Data

The XML Designer allows you to easily create schemas and datasets based on any table, stored procedure, or view that can be browsed in the data connection area of Server Explorer. Additionally, you can pick and choose individual columns if the entire data structure is not needed.

You create a schema based on existing data sources when the existing source contains the desired structure for your application. This is a quick way to build up your schema without having to define each individual element.

Schemas and datasets can also be created programmatically from any data source that you have access to at run time.

To create a schema to match an existing data source

1. In Solution Explorer, right-click the project you want to add the dataset to, select **Add**, and then select **Add New Item**.
2. In the **Add New Item** dialog box, select **Dataset**, give the dataset a name, and then click **Open**.

   A new, blank schema file (.xsd) and a corresponding **Dataset** class file are added to your project.

   > **Note** The name you use will also be the name of the **Dataset** class file generated from the schema.

3. In Server Explorer, expand the **Data Connections** node and then expand the node for the connection to the data source you want to use.

   > **Note** If you do not already have a connection, create one. For more information, see Adding New Data Connections in Server Explorer.

4. In the **Connection** node, expand the node for **Tables**, **Stored Procedures**, or **Views**, depending on what you want to infer the schema structure from.
5. If you want to build the schema based on only specific columns (not on all columns in the data source), open the source node and select the columns you want.
6. Drag the selected table, view, stored procedure, or columns onto the XML Designer.
7. Save the schema.

   Visual Studio creates the schema structure from the elements you specify and regenerates the corresponding class file for your typed dataset.

8. If you have added multiple tables to the dataset and want to establish a relationship between them, add a **DataRelation** object to the schema. For more information, see Creating DataRelation Objects with the XML Designer.

If you need to make changes to the schema, you can use the XML Designer to add, edit, and remove elements.

See Also

Creating XML Schemas and Datasets | Creating XML Schemas from XML Files | Relationship Between XML Schemas and Datasets | Relational Data in XML Schemas | XML Files | XML Schemas and Data | Introduction to Data Access with ADO.NET | Introduction to Datasets

Creating New Typed Datasets with the XML Designer

A typed dataset is a class with an associated XML Schema. If your application needs a typed dataset and there is no available data source to base your associated schema on, you can create a typed dataset schema manually using the XML Designer. If there is a data source that has the structure you want for your dataset, see Creating XML Schemas and Datasets from Existing Data.

To create a schema manually using the XML Designer

1. In Solution Explorer, right-click the project you want to add the dataset to, select **Add**, and then select **Add New Item**.
2. In the **Add New Item** dialog box, select **Dataset**, give the dataset a name, and then click **Open**.

   Visual Studio adds a new, blank schema file (.xsd) to the project as well as a corresponding **Dataset** class file.

   > **Note** The name you use will also be the name of the **Dataset** class file generated from the schema.

3. To add a table definition to the schema, drag an **element** object from the **XML Schema** tab of the Toolbox onto the designer.
4. Name the table by overwriting **element1** in the box at the top of the element.

   This will change the type of the element to match what typed in.

5. To add a column definition to the table, move to a blank line in the table definition and select either **element** or **attribute** from the type drop-down.
6. Assign a name and a data type to the new column definition.

   > **Note** When you create a column definition in a schema, you can create it as either an element or an attribute. The choice you make depends primarily on your own application requirements. If you are defining the schema to match a specification for sharing data with another application, then the specification will dictate your choice. If your dataset requires data types that are not built in to the W3C spec then you should define your columns as elements because elements are not restricted to the built in types, whereas attributes are.

7. To add a primary key definition to the table, drag a **key** element from the **XML Schema** tab of the Toolbox onto the table definition,
8. In the **Key Editor** dialog box:
   a. If the default name is not appropriate, type in a new name.
   b. In the **Fields** list, select the next empty line and use the drop-down list to choose the column to serve as the primary key.
   c. If there are multiple parts to the primary key, choose each part of the primary key separately, one per line.
   d. Check **Primary Key**.

   The option to allow null keys is disabled.

   > **Tip** To delete a key element, select the row with the key to be deleted, right-click it and choose **Delete Key**.

9. Repeat Steps 3 through 8 for each table you want to add to the dataset.
10. Save the schema.

    The corresponding class file is regenerated to incorporate the changes made.

See Also

Creating XML Schemas and Datasets | Relationship Between XML Schemas and Datasets | Relational Data in XML Schemas | XML Files | XML Schemas and Data | Introduction to Data Access with ADO.NET | Introduction to Datasets

Creating ADO.NET Typed DataSets from Schemas

The XML Designer can generate an ADO.NET typed dataset from an XML Schema (.xsd file).

To create an ADO.NET typed dataset from an existing schema

1. From the **Project** menu, choose **Add Existing Item.**
2. In the **Add Existing Item** dialog box, navigate to the schema you want to add, select the file, and click **Open**.
3. In Solution Explorer double-click the .xsd file you added to the project in the previous step.
4. From the **Schema** menu, choose **Generate DataSet**.

   A typed dataset has now been created and placed below the XML Schema file in Solution Explorer.

   > **Note** You may have to click the **Show All Files** button in Solution Explorer to see the files that have been generated.

See Also

Creating XML Schemas and Datasets from Existing Data | Creating XML Schemas from XML Files

Importing External Elements into XML Schemas

Elements from external namespaces can be imported and declared within your schema. Elements imported from multiple namespaces can be used in a schema. The only restriction is that the elements must be globally declared within the external schema.

Importing from other schemas enables types from different target namespaces to be used together.

To import existing data types into XML Schemas

1. Load an XML Schema into the XML Designer.
2. Place focus onto the design surface in **Schema** view.
3. Click **import** in the Properties window.
4. Open the import collection by clicking the ellipses next to the word **(Collection)**.

   The XmlImport Collection Editor appears.

5. Click **Add** to add an import reference to the collection.
6. The **id** field can be assigned any id.
7. The **namespace** field can be assigned any Uniform Resource Identifier (URI).
8. The **schemaLocation** field can be assigned any URI.

For more information about the **Collection Editor Dialog Box**, see Collection Editor Dialog Box.

See Also

Creating XML Schemas and Datasets | Relationship Between XML Schemas and Datasets | Relational Data in XML Schemas | XML Files | XML Namespaces | XML Schemas and Data

XML Namespaces

XML namespaces provide a method for uniqueness, which differentiates between multiple schema authors who use the same element names. Namespaces qualify the names of XML elements and XML attributes in XML document instances. A qualified name consists of a prefix and a local name, separated by a colon. The prefix functions only as a placeholder; it is mapped to a Uniform Resource Identifier (URI) that specifies a namespace. The combination of the universally managed URI namespace and the local name produces a name that should be universally unique.

Namespaces in XML must conform to the World Wide Web Consortium (WC3) specification called *Namespaces in XML*. For more information, see the W3C spec on Namespaces on the W3C Web site (http://www.w3.org/).

By default in Visual Studio each of the elements in a schema has a prefix `xs:` which is associated with the XML Schema namespace through the declaration, `xmlns:xs="http://www.w3.org/2001/XMLSchema"`, that appears in the schema element. The prefix `xs:` is used by convention to denote the XML Schema namespace, although any prefix can be used. The same prefix, and hence the same association, also appears on the names of built-in simple types — for example, `xs:string`. The purpose of the association is to identify the elements and simple types as belonging to the vocabulary of the XML Schema language rather than the vocabulary of the schema author.

See Also

XML Schemas and the XML Designer| XML Elements, Attributes, and Types | Changing the Target Namespace | Creating XML Schemas and Datasets | Relational Data in XML Schemas | XML Files

Changing the Target Namespace

When creating new XML Schemas, Visual Studio creates a default namespace entry within the opening schema tag. These entries need to be updated to reflect the proper namespace for your schema. The following example highlights the entries that need to be changed in a new XML Schema.

> **Note** New dataset schemas will have slightly different code than the following example, but you need only be concerned with the http://tempuri.org/ entries.

```
<?xml version="1.0" encoding="utf-8" ?>
<xs:schema id="XMLSchema1" targetNamespace="http://tempuri.org/XMLSchema1.xsd" elementFormDef
ault="qualified" xmlns="http://tempuri.org/XMLSchema1.xsd" xmlns:xs="http://www.w3.org/2001/X
MLSchema">
```

To change the target namespace in XML Schemas

1. Open your schema in XML view.
2. Locate the `targetNamespace="http://tempuri.org/..."` entry and replace it with the correct Uniform Resource Identifier (URI) for your schema.
3. Locate the `xmlns="http://tempuri.org/..."` entry and replace it with the correct URI for your schema.

See Also

XML Schemas and the XML Designer| XML Namespaces | Creating XML Schemas and Datasets | Relational Data in XML Schemas | XML Files

Walkthrough: Creating an XML Schema with the XML Designer

In this walkthrough you will create an XML purchase order schema as part of a Windows Application project. The walkthrough will consist of three main sections:

1. Creating a Windows Application project and adding an XML Schema.
2. Creating a relational table:
   a. Adding and defining a new **simpleType** object.
   b. Adding and defining a new **complexType** object.
   c. Adding and defining a new **Element** object.
3. Editing generated XML using the XML editor.

Creating a Windows Application and Adding an XML Schema

To create a new Windows Application project

1. From the **File** menu, point to **New**, and then click **Project** to display the **New Project** dialog box.
2. Depending on what language you want to use, select **Visual C# Projects** or **Visual Basic Projects** in the **Project Types** pane, and then select **Windows Application**.
3. Name the project **SampleSchema**.

To add an XML Schema to the project

- From the **Project** menu, choose **Add New Item**, then double-click the **XML Schema** icon in the **Add New Item** dialog box.

  The XML Designer appears.

Defining Simple and Complex Types

Before constructing the relational table, you will first build simple and complex type definitions that you will use to format specific elements of the purchase-order schema. The new types are built using existing XML data types, such as string and integer.

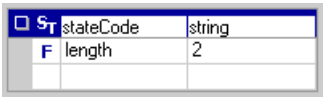
First you will define a simple type, which will be named **stateCode**. This simple type will be used to limit the size of a string to two characters.

To add an XML simpleType to the project

1. If not already open, double click the **XMLSchema1.xsd** file to bring up the XML Designer.
2. Click the **XML Schema** tab of the Toolbox and drag a **simpleType** onto the design surface.
3. Change the name of the simpleType by clicking the first text box in the header and replacing **simpleType1** with **stateCode**.
4. Set the base type for the stateCode type by clicking the drop-down list in the header and selecting **string**.
5. Navigate to the first cell in the next row by pressing the TAB key.
6. Select **facet** from the drop-down list.
7. Press TAB to go to the next cell and select **length** from the drop-down list.
8. TAB to the third cell of the same row, enter the value **2**.

   This requires that the value entered into the State field be two characters.

   Your stateCode should look like this in Schema view:

9. Click the **XML** tab at the bottom left of the **XML Designer** to see the XML that has been added:

```
<xs:simpleType name="stateCode">
   <xs:restriction base="xs:string">
      <xs:length value="2" />
   </xs:restriction>
</xs:simpleType>
```

This `stateCode` simple type will be used to define the `State` element within the complex type you will create in the next section.

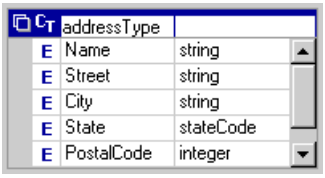
The complexType named `addressType` defines a set of elements that will appear in any element typed as addressType. For example, a `billTo` element will include information on names, addresses, and dates when its type is set to the previously defined addressType. By constructing the complex type and using it within an element, you are generating a nested relationship. For more information, see Creating Complex XML Types.

To add an XML complexType to the project

1. Click the **Schema** tab of the XML Designer.
2. Drag a **complexType** from the **XML Schema** tab of the Toolbox onto the design surface.
3. Change **complexType1** to **addressType** to name the type.
4. Add an XML attribute to addressType by clicking the first cell of the first row and selecting **element** from the drop-down list.
5. In the second column, change **element1** to **Name**.
6. In the third column, accept the default value of **string**.
7. Add the following XML elements and set their names and types as follows:

Element name	Data type
Street	string
City	string
State	stateCode
PostalCode	integer

   Your addressType should look like this in Schema view:



8. To see the XML that has been added to your .xsd file, click the XML tab at the bottom of the designer. You will see the following XML:

```
<xs:complexType name="addressType">
    <xs:sequence>
    <xs:element name="Name" type="xs:string"/>
    <xs:element name="Street" type="xs:string"/>
    <xs:element name="City" type="xs:string"/>
    <xs:element name="State" type="stateCode"/>
    <xs:element name="PostalCode" type="xs:integer"/>
    </xs:sequence>
</xs:complexType>
```

Creating a Relational Table

When you drag the element object from the Toolbox to the design surface, you are really adding an element containing an unnamed complexType. Including the unnamed complex type defines the element to be a relational table. Additional elements can then be added under the complexType to define the relation fields (or columns). If you define one of these new elements to be a new unnamed complexType, you are creating a nested relation inside of the parent relation with its own unique columns. For details see Tables, Columns, Keys, and Constraints in XML Schemas.

Defining new unnamed complex type elements within the PurchaseOrder or Items element creates additional nesting in the schema. Within one purchase order there can be many Items, and within each Item, many additional elements (such as price, size, and so on). In the following procedure, an element Items is added to the purchaseOrder relational table and typed as an unnamed complexType. Because you are defining a new relational table, this causes a new element to appear on the design surface. Within the new items relation, adding the item element and setting its type to unnamed complexType, creates another relational table, which also appears on the design surface.

To add an XML element to the project

1. Click the Toolbox and from the **XML Schema** tab drag an **element** object onto the design surface.

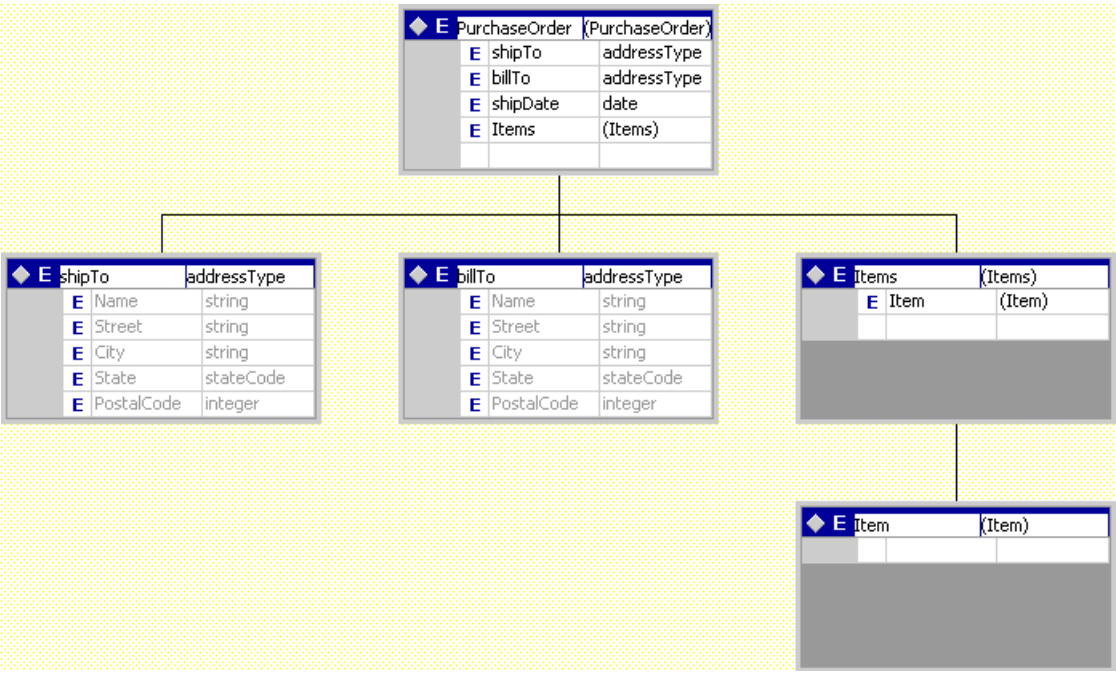
2. Change **element1** to **PurchaseOrder** to name the element. You can leave the data type as (**PurchaseOrder**).
3. Add an element to the purchase order by clicking the first cell of the first row and selecting **element** from the drop-down list.
4. Name the element **shipTo** and set its data type to **addressType**.
5. Add the following XML elements and set their names and types as follows:

Element name	Data type
billTo	addressType
shipDate	date
Items	unnamed complexType

When you type the Item element to be anonymous, an additional element is added to the design surface, which is another relational table.

6. In the **Items element**, add an element, name it **Item,** and set its type to **Unnamed ComplexType**.

Your purchase order should look like this in Schema view:



The following XML has now been added to your .xsd file:

```
<xs:element name="PurchaseOrder">
 <xs:complexType>
    <xs:sequence>
       <xs:element name="shipTo" type="addressType"/>
       <xs:element name="billTo" type="addressType"/>
       <xs:element name="shipDate" type="xs:date"/>
       <xs:element name="Items">
          <xs:complexType>
             <xs:sequence>
                <xs:element name="Item">
                   <xs:complexType>
                       <xs:sequence />
                   </xs:complexType>
                </xs:element>
             </xs:sequence>
          </xs:complexType>
       </xs:element>
     </xs:sequence>
  </xs:complexType>
</xs:element>
```

Editing XML

You can use the XML tab of the XML Designer to edit the XML that was generated when you added elements and types to the designer surface. The XML editor features IntelliSense and statement completion. An invalid statement is tagged with a red wavy line. Mousing over the incorrect statement causes an error message to appear.

To edit XML

1. Click the **XML** tab of the XML Designer to view the XML.
2. Within the `Item` element, change the self-closing tag (`<xs:sequence />`) into separate opening and closing tags (`<xs:sequence></xs:sequence>`).
3. After the `<xs:sequence>` tag in the `Item` element, type the following:

   ```
   <xs:element name="Quantity" type="xs:integer"/>
   <xs:element name="Price" type="xs:decimal"/>
   <xs:element name="ProductID" type="xs:integer"/>
   ```

   You have created three new elements — Quantity, Price, and ProductID — and defined data types for each.

4. Next type **<invalid/>** and note the red wavy line indicating an error. Mouse over the red wavy line to see an error message. Errors will appear in the Task List as well.
5. Delete the **<invalid/>** tag to fix the error.
6. Save the schema.

   The XML beneath the `Items` element should now look like the following in XML view:

   ```
   <xs:element name="Items">
     <xs:complexType>
       <xs:sequence>
         <xs:element name="Item">
           <xs:complexType>
             <xs:sequence>
               <xs:element name="Quantity" type="xs:integer"/>
               <xs:element name="Price" type="xs:decimal"/>
               <xs:element name="ProductID" type="xs:integer"/>
             </xs:sequence>
           </xs:complexType>
         </xs:element>
       </xs:sequence>
     </xs:complexType>
   </xs:element>
   ```

See Also

Introduction to XML Schemas | Relational Data in XML Schemas | Tables, Columns, Keys, and Constraints in XML Schemas | XML Elements, Attributes, and Types| XML Schema Reference (XSD)

Walkthrough: Reading XML Data into a Dataset

ADO.NET provides simple methods for working with XML data. In this walkthrough you will create a Windows application that will load XML data into a dataset. The dataset will then be displayed in a **DataGrid** control. Finally, an XML Schema based on the contents of the XML file will be displayed in a text box.

This walkthrough will consist of five main steps:

1. Create a new Visual Basic or Visual C# project.
2. Create an XML file to be read into the dataset.
3. Create the user interface.
4. Add code to create the dataset, read the XML file, and display it in a **DataGrid** control.
5. Add code to display the XML Schema based on the XML file in a **TextBox** control.

Create a New Project

In this step, you will create a Visual Basic or Visual C# project that will contain this walkthrough.

To create a new project

1. From the **File** menu, point to **New**, and then click **Project** to display the **New Project** dialog box.
2. Depending on what language you want to use, select **Visual Basic Projects** or **Visual C# Projects** in the **Project Types** pane, and then select **Windows Application**.
3. Name the project **ReadingXML**, then click **OK**.

   Visual Studio will add the **ReadingXML** project to Solution Explorer and display a new Windows Form in the designer.

Generate the XML File to Be Read into the Dataset

Because this walkthrough focuses on reading XML data into a dataset, the contents of an XML file is provided. This XML file must be located in a folder that will be accessible from the machine that you are using to create this application.

To create the XML file that will be read into the dataset

1. From the **File** menu, point to **New**, and then click **File**.

   The **New File** dialog box appears.

2. Select **XML File** and click **Open**.

   The XML file loads into the designer and is ready for edit.

3. Paste the following code into the editor below the XML declaration:

   Tip Right-click the editor and select **Paste as HTML**.

```
<Authors_Table>
  <authors>
    <au_id>172-32-1176</au_id>
    <au_lname>White</au_lname>
    <au_fname>Johnson</au_fname>
    <phone>408 496-7223</phone>
    <address>10932 Bigge Rd.</address>
    <city>Menlo Park</city>
    <state>CA</state>
    <zip>94025</zip>
    <contract>true</contract>
  </authors>
  <authors>
    <au_id>213-46-8915</au_id>
    <au_lname>Green</au_lname>
    <au_fname>Margie</au_fname>
    <phone>415 986-7020</phone>
```

```
      <address>309 63rd St. #411</address>
      <city>Oakland</city>
      <state>CA</state>
      <zip>94618</zip>
      <contract>true</contract>
    </authors>
    <authors>
      <au_id>238-95-7766</au_id>
      <au_lname>Carson</au_lname>
      <au_fname>Cheryl</au_fname>
      <phone>415 548-7723</phone>
      <address>589 Darwin Ln.</address>
      <city>Berkeley</city>
      <state>CA</state>
      <zip>94705</zip>
      <contract>true</contract>
    </authors>
    <authors>
      <au_id>267-41-2394</au_id>
      <au_lname>Hunter</au_lname>
      <au_fname>Anne</au_fname>
      <phone>408 286-2428</phone>
      <address>22 Cleveland Av. #14</address>
      <city>San Jose</city>
      <state>CA</state>
      <zip>95128</zip>
      <contract>true</contract>
    </authors>
    <authors>
      <au_id>274-80-9391</au_id>
      <au_lname>Straight</au_lname>
      <au_fname>Dean</au_fname>
      <phone>415 834-2919</phone>
      <address>5420 College Av.</address>
      <city>Oakland</city>
      <state>CA</state>
      <zip>94609</zip>
      <contract>true</contract>
    </authors>
  </Authors_Table>
```

4. From the **File** menu, point to **Save XMLFile1 As**.

   The **Save File As** dialog box appears.

5. Name the file **authors.xml** and click **Save**.

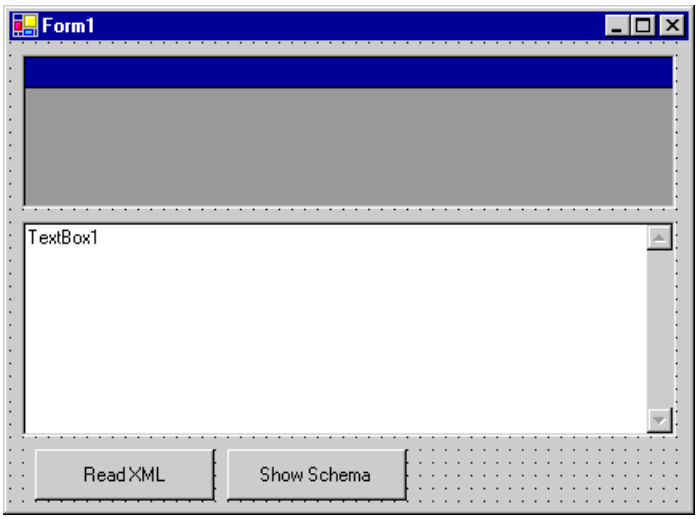
   Note Be sure to remember the complete path, it will be needed later in this walkthrough.

Create the User Interface

The user interface for this application will consist of the following:

- A **DataGrid** control that will display the contents of the XML file as data.
- A **TextBox** control that will display the XML Schema for the XML file.
- Two **Button** controls.
  - One button reads the XML file into the dataset and displays it in the **DataGrid** control.
  - A second button extracts the schema from the dataset, and through a StringWriter displays it in the **TextBox** control.

Initial layout for this application



To add controls to the form

1. From the **Windows Forms** tab of the Toolbox, drag the following controls onto the form:
   - 1 DataGrid control
   - 1 TextBox control
   - 2 Button controls
2. Set the following properties:

Control	Property	Setting
TextBox1	**Multiline**	**true**
	ScrollBars	**Vertical**
Button1	**Name**	**btnReadXML**
	Text	**Read XML**
Button2	**Name**	**btnShowSchema**
	Text	**Show Schema**

Create the Dataset that will Receive the XML Data

You will programmatically create a new dataset named **authors**. It will be declared at form level in order to be accessible from the **button_click** event handlers. For more information about datasets, see Introduction to Datasets.

To add the code to create a new dataset that will receive the XML data

1. With **Form1.vb** (VB users) or **Form1.cs** (C# users) selected in Solution Explorer, click the **View Code** button in the Solution Explorer toolbar.
2. In the declarations area, declare a new dataset named authors:

```
' Visual Basic
Dim dsAuthors As New DataSet("authors")

// C#
DataSet dsAuthors = new DataSet("authors");
```

Create the Event Handler to Read the XML into the Dataset

The **Read XML** button reads the XML file into the dataset and sets properties on the **DataGrid** that bind it to the dataset.

To add code to the btnReadXML_Click event handler

1. In Solution Explorer, select **Form1** and click the **View Designer** button on the Solution Explorer toolbar.
2. Double-click the **Read XML** button.

   The Code Editor opens at the **btnReadXML_Click** event handler.

3. Type the following code into the **btnReadXML_Click** event handler:

```
' Visual Basic
Dim filePath As String
filePath = "Complete path where you saved the XML file"
dsAuthors.ReadXml(filePath)
With DataGrid1
   .DataSource = dsAuthors
   .DataMember = "authors"
   .CaptionText = .DataMember
End With

// C#
string filePath = "Complete path where you saved the XML file";
dsAuthors.ReadXml(filePath);
dataGrid1.DataSource = dsAuthors;
dataGrid1.DataMember = "authors";
dataGrid1.CaptionText = dataGrid1.DataMember;
```

Create the Event Handler to Display the Schema in the Textbox

The Show Schema button creates a **StringWriter** object that is filled with the schema and displayed in the **TextBox**.

To add code to the btnShowSchema_Click event handler

1. In Solution Explorer, select **Form1.vb** and click the **View Designer** button.
2. Double-click the **Show Schema** button.

   The Code Editor opens at the **btnShowSchema_Click** event handler.

3. Type the following code into the **btnShowSchema_Click** event handler.

```
' Visual Basic
Dim swXML As New System.IO.StringWriter()
dsAuthors.WriteXmlSchema(swXML)
TextBox1.Text = swXML.ToString

// C#
System.IO.StringWriter swXML = new System.IO.StringWriter();
dsAuthors.WriteXmlSchema(swXML);
textBox1.Text = swXML.ToString();
```

Run the Application

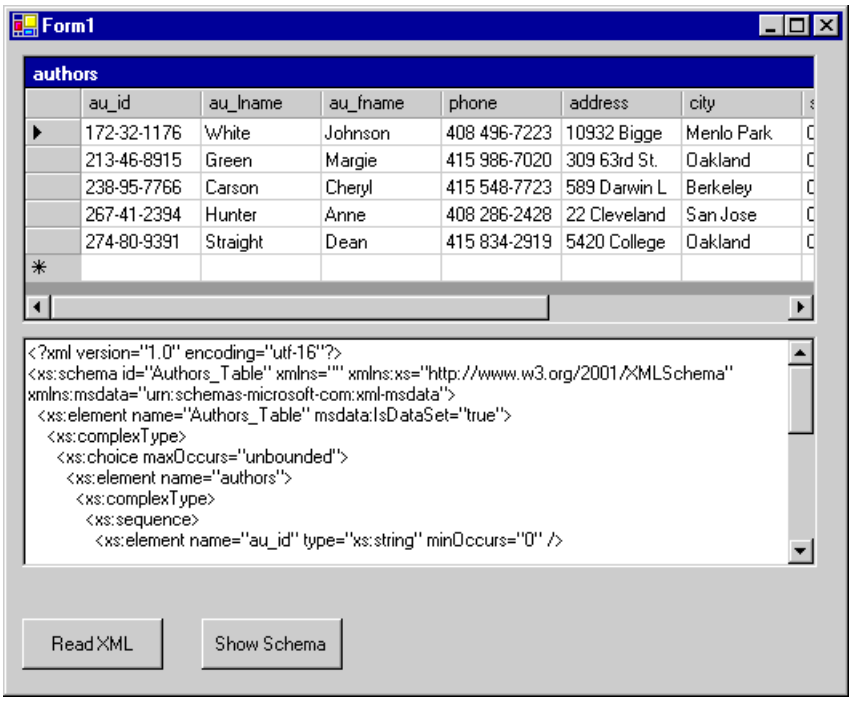
1. Press **F5** to run the application.
2. Click the **Read XML** button.

   The data grid displays the contents of the XML file.

3. Click the **Show Schema** button.

   The text box displays the XML Schema for the XML file.

Result of running the application



Next Steps

This walkthrough shows the basics of reading an XML file into a dataset, as well as creating a schema based on the contents of the XML file. Here are some tasks that might come next:

- Edit the data in the dataset and write it back out as XML. For more information, see DataSet.WriteXml Method.
- Edit the data in the dataset and write it out to a database. For more information, see Updating a Data Source with a Dataset.

See Also

Data Walkthroughs | Introduction to Datasets | DataSet.ReadXml Method | DataSet.WriteXmlSchema Method | Debugging Managed Code | Deploying Applications | XML in Visual Studio

Relational Data in XML Schemas

XML provides the following two approaches for representing relational data, and consequently, the schemas created to validate each construct are different as well.

- **One-to-many relationships** are represented as separate tables of rows that are related by common columns. These common columns are defined as keys within the tables, and a relationship is defined between them. The XML Designer provides the Edit Key and Edit Relation dialog boxes for creating keys and relationships respectively.
- **Nested relationships** are represented hierarchically by parent elements that contain nested child elements.

In This Section

Tables, Columns, Keys, and Constraints in XML Schemas
Provides an overview of how database constructs are represented in XML Schemas.
Nested Relationships in XML
Provides an overview of how nested relationships are represented in XML Schemas.
Creating Tables in XML Schemas
Explains how to create tables with the XML Designer.
Creating Primary and Unique Keys in XML Schemas
Explains how to create keys with the XML Designer.
Deleting Primary Keys and Unique Keys
Explains how to delete keys with the XML Designer.
Creating One-to-Many Relationships in XML Schemas and Datasets
Provides the steps involved in creating a one-to-many relationship with the XML Designer.
Creating Nested Relationships in XML Schemas
Explains how to create a nested relationship using the XML Designer.
Walkthrough: Creating a Dataset with Tables, Keys, and One-to-Many Relationships
Provides step-by-step instructions for how to create an XML Schema that could validate tables, keys, and keyed relationships.

Related Sections

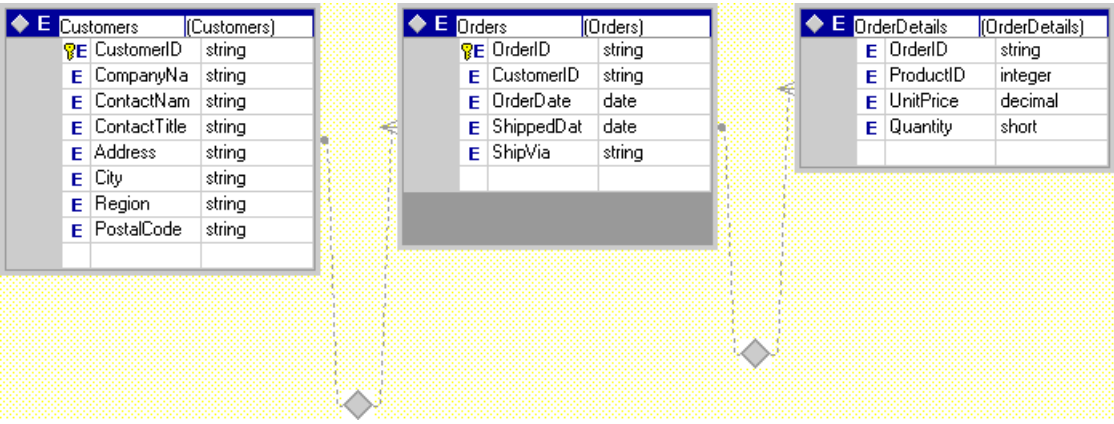
Introduction to DataRelation Objects
Provides an overview of how datasets allow you to specify relationships between tables and how you can take advantage of these relationships.
Creating DataRelation Objects with the XML Designer
Explains how to use the XML Designer to add a **DataRelation** object to a dataset.
XML Schemas and Data
Introduces topics and sections that describe how to use the XML Designer to work with XML Schemas and datasets.
XML Schemas and the XML Designer
Introduces topics that describe the different elements in XML Schemas and how they are created using the XML Designer.
Creating XML Schemas and Datasets
Introduces topics that describe the different ways that schemas and datasets can be created using the XML Designer as well as describing the connection between datasets and schemas.
XML Files
Introduces topics that explain how to use the XML Designer when working with XML documents.
Introduction to Data Access with ADO.NET
Provides a high level overview of the most important concepts of data access in ADO.NET.
Introduction to Datasets
Provides an overview of what ADO.NET datasets are, how they can be typed or untyped, what they are used for, and how they are structured.
XML in Visual Studio
Provides links to a variety of topics about XML.
World Wide Web Consortium (W3C) Web site (http://www.w3.org/)
Links to the XML Schema primer and other XML info posted by the W3C.
XML Developer Center on MSDN Online (http://msdn.microsoft.com/xml/default.asp)
Provides various resources about XML.
XML Schema Reference (XSD)
Provides reference topics for schema elements and data types.
XML Designer
Provides an overview of the XML Design tool in Visual Studio.

Tables, Columns, Keys, and Constraints in XML Schemas

Relationships in XML Schemas involve creating individual tables with columns. If your schema uses a one-to-many relationship, then creating your XML Schema also involves designating the common columns as keys and keyrefs, creating relationships between them, and applying the appropriate constraints. Nested relationships in an XML Schema also involve creating tables with columns, except single column definitions may contain the complete definitions of related tables.

This topic will explain how typical database constructs such as tables, columns, primary keys, foreign keys, unique keys, and constraint rules are represented in XML Schemas and datasets.

Visual representation of relational data in the XML Designer



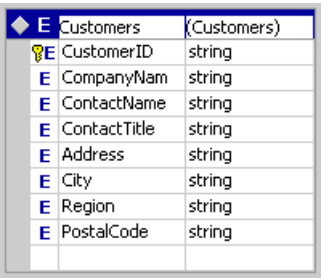
Tables

Tables in XML Schemas are represented by complex elements, and the columns are represented as sub-elements and attributes of that element. The name of the table is equal to the name of the element. The names of the columns are equal to the names of the sub-elements and attributes that make up the complex element. For more information, see Complex Types in XML Schemas.

For example, the following XML Schema could represent a table named `Customers`. The elements `CustomerID` through `PostalCode` represent the columns that make up the table. The `CustomerID` column is defined as the primary key.

> **Note** Notice how the primary key definition is hierarchically outside the
> Customers
> element definition, yet inside of a higher-level
> NewDataSet
> element. If there were other tables defined in this dataset, they would be nested within the
> NewDataSet
> element as well. This ensures that the primary key column is unique within the scope of the entire dataset (as opposed to being unique within the scope of just one record in the
> Customers
> table).

The following is a visual representation of the `Customers` table in the XML Designer:



The following example shows the XML code that represents the Customers table in a dataset named NewDataSet:

```
<xs:element name="NewDataSet" msdata:IsDataSet="true">
 <xs:complexType>
  <xs:choice maxOccurs="unbounded">
   <xs:element name="Customers">
    <xs:complexType>
     <xs:sequence>
```

```
        <xs:element name="CustomerID" type="xs:string"/>
        <xs:element name="CompanyName" type="xs:string"/>
        <xs:element name="ContactName" type="xs:string" minOccurs="0"/>
        <xs:element name="ContactTitle" type="xs:string" minOccurs="0"/>
        <xs:element name="Address" type="xs:string" minOccurs="0"/>
        <xs:element name="City" type="xs:string" minOccurs="0"/>
        <xs:element name="Region" type="xs:string" minOccurs="0"/>
        <xs:element name="PostalCode" type="xs:string" minOccurs="0"/>
       </xs:sequence>
      </xs:complexType>
     </xs:element>
    </xs:choice>
  </xs:complexType>
    <xs:unique name="NewDataSetKey1" msdata:PrimaryKey="true">
     <xs:selector xpath=".//Customers"/>
     <xs:field xpath="CustomerID"/>
    </xs:unique>
  </xs:element>
```

Columns

In the previous example, note how the individual columns (such as `CustomerID`, `CompanyName`, and `ContactName`) are declared as simple elements within a complex-type definition. The two elements without a **minOccurs** attribute are required columns, whereas the elements with the `minOccurs="0"` attribute are optional columns. The attribute `type="string"` refers to a simple type that is built-in to all XML Schemas. More importantly, note how the table is defined as an in-line unnamed complexType (the columns) within the `NameOfTable` element (the table). The following example shows how to define a table and its columns.

```
  <xs:element name="NameOfTable">
   <xs:complexType>
     <xs:sequence>
       <xs:element name="RequiredColumn" type="xs:string"/>
       <xs:element name="OptionalColumn" type="xs:string" minOccurs="0"/>
     </xs:sequence>
   </xs:complexType>
  </xs:element>
```

> **Note** When you create a column definition in a schema, you can create it as either an element or an attribute. The choice you make depends primarily on your own application requirements. If you are defining the schema to match a specification for sharing data with another application, then the specification will dictate your choice.

Primary Keys and Unique keys

Primary Keys

Primary keys ensure columns contain unique values that cannot be NULL. In the XML Designer, adding a key that does not have the **Primary key** option selected creates a unique key. Unique constraints ensure that columns do not contain duplicate values.

Based on the example earlier in this section, the following example shows how to define a `CustomerID` column as a primary key in an XML Schema:

```
  <xs:key name="NewDataSetKey1" msdata:PrimaryKey="true">
     <xs:selector xpath=".//Customers"/>
     <xs:field xpath="CustomerID"/>
```

```
</xs:key>
```

> **Note** By dragging a **key** from the **XML Schema** tab of the Toolbox onto an element and setting the appropriate fields, the previous code would be automatically generated by the XML Designer. For more information, see Creating Primary and Unique Keys in XML Schemas.

Unique Keys

Unique constraints can allow null values, whereas primary key constraints do not allow null values. Another consideration when deciding whether to use a primary or unique key is that tables can have multiple unique constraints but only one primary key.

Using the same example from earlier in this section, if you had an application that required that the `CompanyName` column must be unique, you would impose a unique key constraint on that column. The following example shows how to define a `CompanyName` column as a unique key in an XML Schema:

```
<xs:unique name="NewDataSetKey2">
   <xs:selector xpath=".//Customers" />
   <xs:field xpath="CompanyName" />
</xs:unique>
```

> **Note** By dragging a **key** from the **XML Schema** tab of the Toolbox onto an element and setting the appropriate fields, the previous code would be automatically generated by the XML Designer. For more information, see Creating Primary and Unique Keys in XML Schemas.

KeyRefs

A keyref defines the column on the many side of a one-to-many relationship, much like a foreign key in database terminology. An element that has been designated as a keyref has a relationship with the key or unique element in another table. Using the previous example, assume there is a table named `Orders` in the same dataset that has a column named `CustomerID`. You can create a **keyref** named `CustomerOrders` that creates the relationship between the `Customers` table and the `Orders` table.

```
<xs:keyref name="CustomerOrders" refer="NewDataSetKey1">
   <xs:selector xpath=".//Orders" />
   <xs:field xpath="CustomerID" />
</xs:keyref>
```

> **Note** By dragging a **Relation** from the **XML Schema** tab of the Toolbox onto an element and setting the appropriate fields, the previous code would be automatically generated by the XML Designer. For more information, see Creating DataRelation Objects with the XML Designer.

Referential Integrity

When creating a relationship with the Edit Relation dialog box, you can select the rules that affect what happens to related data. These rules are called *referential integrity*. When updating or deleting a record in a primary key column, there can be many records in another table that reference the primary key. The following table describes the different constraint rules that can be applied using the **Edit Relation** dialog box.

Rule	Description
Cascade	Deletes or updates related rows.
SetNull	Sets values in related rows to DBNull.
SetDefault	Sets values in related rows to DefaultValue.
None	No action taken on related rows.

See Also

Relational Data in XML Schemas | XML Schemas and Data | XML Schemas and the XML Designer | Creating XML Schemas and Datasets | XML Files | Introduction to Data Access with ADO.NET | Introduction to Datasets

Nested Relationships in XML

In the hierarchical representation of data that XML provides, nested relationships are represented by parent elements that contain nested child elements.

The following example shows `Orders` for a customer nested within a `Customers` element. In other words, the data for both a customer and all their orders are grouped in a single nested hierarchy named `Customers`. This differs from a one-to-many relationship where all customer information, as well as order information, would hierarchically exist at the same level and be related through the use of keys and keyrefs. For more information, see Tables, Columns, Keys, and Constraints in XML Schemas.

The follow example shows nested XML data:

```
<CustomerOrders>
  <Customers>
    <CustomerID>ALFKI</CustomerID>
    <Orders>
      <OrderID>10643</OrderID>
        <OrderDetails>
      <CustomerID>ALFKI</CustomerID>
      <OrderDate>1997-08-25</OrderDate>
    </Orders>
    <Orders>
      <OrderID>10692</OrderID>
      <CustomerID>ALFKI</CustomerID>
      <OrderDate>1997-10-03</OrderDate>
    </Orders>
    <CompanyName>Alfreds Futterkiste</CompanyName>
  </Customers>
  <Customers>
    <CustomerID>ANATR</CustomerID>
    <Orders>
      <OrderID>10308</OrderID>
      <CustomerID>ANATR</CustomerID>
      <OrderDate>1996-09-18</OrderDate>
    </Orders>
    <CompanyName>Ana Trujillo Emparedados y helados</CompanyName>
  </Customers>
</CustomerOrders>
```

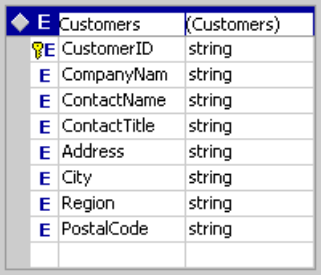
See Also

Relational Data in XML Schemas | Creating Nested Relationships in XML Schemas | XML Schemas and Data | XML Schemas and the XML Designer | Creating XML Schemas and Datasets | XML Files | Introduction to Data Access with ADO.NET | Introduction to Datasets

Creating Tables in XML Schemas

You can create tables and columns in XML Schemas in order to validate that table representations in XML data contain valid structure as well as valid data types. For more information about tables and columns, see Tables, Columns, Keys, and Constraints in XML Schemas.

Visual representation of a table in the XML Designer



E	Customers	(Customers)
E	CustomerID	string
E	CompanyNam	string
E	ContactName	string
E	ContactTitle	string
E	Address	string
E	City	string
E	Region	string
E	PostalCode	string

In XML Schemas, tables can be constructed manually or automatically with the XML Designer depending on the availability of a data source that represents the tables you need to validate within your application. You can also construct your schema both ways. For example: suppose you need to create a schema to validate XML data that includes a table with customer information and a table with customer feedback from your customer service department. Your organization already has the customer information table in an existing database; this can be created in your new schema automatically from your existing data source. The new table with customer feedback can be created manually and a relationship can be defined between the two tables.

Automatic Table Construction

You can use individual tables that are in existing data sources to easily construct schemas and datasets.

To automatically create table representations in XML Schemas using existing data sources.

1. In Solution Explorer, right click the project you want to add the dataset to, select **Add**, and then select **Add New Item**.
2. In the **Add New Item** dialog box, select **Dataset**, give the dataset a name, and then click **Open**.

   Visual Studio adds a new, blank schema file (.xsd) to the project as well as a corresponding **DataSet** class file.

   > **Note** The name you use will also be the name of the **DataSet** class file generated from the schema.

3. In Server Explorer, expand the **Data Connections** node and then expand the node for the connection to the data source you want to use.

   > **Note** If you don't already have a connection, create one. For more information, see Adding New Data Connections in Server Explorer.

4. In the **Connection** node, expand the node for the **Tables** you want to infer the schema structure from.
5. Drag the selected table onto the XML Designer.
6. Save the schema.

   Visual Studio creates the schema structure from the elements you specify and regenerates the corresponding class file for your typed dataset.

7. Repeat steps 4 and 5 to add more tables to your schema.

Manual Table Construction

If your schema requires table representations that have no available data sources to base them on, you can construct them manually with the XML Designer.

To manually create table representations in XML Schemas

1. In Solution Explorer, right-click the project you want to add the dataset to, select **Add**, and then select **Add New Item**.
2. In the **Add New Item** dialog box, select **Dataset**, give the dataset a name, and then click **Open**.

   Visual Studio adds a new, blank schema file (.xsd) to the project as well as a corresponding **DataSet** class file.

   > **Note** The name you use will also be the name of the **DataSet** class file generated from the schema.

3. Drag an **Element** from the **XML Schema** tab of the Toolbox onto the designer.

   A table definition is added to the schema,

4. Name the table by overwriting **Element1** in grid header of the element.
5. Move to a blank line in the table definition and select either **Element** or **Attribute** from the type drop-down. For more information, see Tables, Columns, Keys, and Constraints in XML Schemas.

   A column definition is added to the schema.

6. Assign a name and a data type to the new column definition.
7. Repeat Steps 3 through 5 for each table you want to add to the dataset.
8. Save the schema.

   The corresponding class file is regenerated to incorporate the changes you made.

Next Steps

- You can now define columns as primary and unique keys. For more information, see Creating Primary and Unique Keys in XML Schemas.
- You can also establish relationships between the tables in your schema. For more information, see Creating DataRelation Objects with the XML Designer.

If you need to make changes to the schema, you can use the XML Designer to add, edit, and remove elements.

See Also

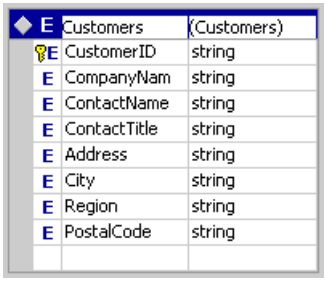
Relational Data in XML Schemas | Tables, Columns, Keys, and Constraints in XML Schemas | Nested Relationships in XML | Creating Primary and Unique Keys in XML Schemas | Creating One-to-Many Relationships in XML Schemas and Datasets | XML Schemas and Data | XML Schemas and the XML Designer | Creating XML Schemas and Datasets | XML Files | Introduction to Data Access with ADO.NET | Introduction to Datasets

Creating Primary and Unique Keys in XML Schemas

A primary key defines a column in a table as unique and required (null values are not allowed), whereas a unique key defines a column in a table as unique and optional (null values are allowed).

You designate columns in tables as keys, which allow you to create relationships and constraints. For more information about keys, see Tables, Columns, Keys, and Constraints in XML Schemas.

The following is a visual representation of a Key definition in the XML Designer



E	Customers	(Customers)
E	CustomerID	string
E	CompanyNam	string
E	ContactName	string
E	ContactTitle	string
E	Address	string
E	City	string
E	Region	string
E	PostalCode	string

To create a primary or unique key in an XML Schema or dataset

1. Right-click to the left of the row of the element you want to designate as a key. Approximately where the key glyph is in the picture above.
2. Click **Add** then click **New Key**.

   The **Edit Key** dialog box appears.

3. In the **Name** box, replace the default name with a name appropriate for this key.

   The **Selector** box and the **Field** area will default to the element you chose in Step 1.

4. If you need a key that is comprised of more than one column definition (a multipart key), select additional elements in the **Field** area.
5. If you need to allow the element to contain a null value, select **nullable**.
6. If you need a primary key, select **Primary Key**. Otherwise, a unique key will be created.

Next Steps

- Construct tables that will share this key data in order to define relationships. For more information, see Creating Tables in XML Schemas.
- You can now create a relationship with other tables that contain columns that are common to the key you just defined. For more information, see Creating DataRelation Objects with the XML Designer.

See Also

Relational Data in XML Schemas | Tables, Columns, Keys, and Constraints in XML Schemas | Nested Relationships in XML | Creating Tables in XML Schemas | Creating One-to-Many Relationships in XML Schemas and Datasets | Creating Nested Relationships in XML Schemas | XML Schemas and Data | XML Schemas and the XML Designer | Creating XML Schemas and Datasets | XML Files

Deleting Primary Keys and Unique Keys

If your applications data requirements change, or if a key was inadvertently defined, you can delete primary and unique keys.

To delete an existing key

1. Select the row with the key definition you want to delete.
2. Right-click the row in the design grid with the key you want to delete.
3. Select **Delete Key** from the shortcut menu.

> **Note** If a row is part of multiple key definitions, the **Choose Key** dialog box will appear, allowing you to select and delete individual key definitions.

See Also

Relational Data in XML Schemas | Tables, Columns, Keys, and Constraints in XML Schemas | Creating Primary and Unique Keys in XML Schemas | Creating One-to-Many Relationships in XML Schemas and Datasets | XML Schemas and Data | XML Schemas and the XML Designer | Creating XML Schemas and Datasets | XML Files

Creating One-to-Many Relationships in XML Schemas and Datasets

You create a schema with one-to-many relationships when XML files contain related elements that are hierarchically at the same level. For more information, see Tables, Columns, Keys, and Constraints in XML Schemas.

> **Tip** If you have access to an XML file (.xml) that contains the structured data that your application needs to validate, see Creating XML Schemas from XML Files.

There are three main steps involved in creating an XML Schema that will be used to validate data that is structured in a one-to-many relationship.

To create a schema that represents a one-to-many relationship

1. Create the tables that will represent your applications data structure.
2. Create the primary keys, unique keys, and keyrefs.
3. Create the relationships between the tables.

For instructions on	See
Constructing Tables	Creating Tables in XML Schemas
Defining Keys	Creating Primary and Unique Keys in XML Schemas
Defining Relationships (keyrefs)	Creating DataRelation Objects with the XML Designer

See Also

Relational Data in XML Schemas | Tables, Columns, Keys, and Constraints in XML Schemas | Nested Relationships in XML | XML Schemas and Data | XML Schemas and the XML Designer | Creating XML Schemas and Datasets | XML Files | Introduction to Data Access with ADO.NET | Introduction to Datasets

Creating Nested Relationships in XML Schemas

You create a schema with nested relationships when XML data is represented by parent elements that contain nested child elements. For more information, see Nested Relationships in XML.

> **Tip** If you have access to an XML file (.xml) that contains the structured data that your application needs to validate then creating a schema can be accomplished by following the instructions for Creating XML Schemas from XML Files.

To construct a schema to validate nested data

1. Create the individual tables to represent the different hierarchies of data that your XML represents. For more information, see Creating Tables in XML Schemas.
2. Select the table that represents the nested child element and drag and drop it onto the table that represents the parent element.
3. If you have any tables that represent nested child elements, repeat Step 2.

The following example shows how to construct an XML Schema that validates nested data. In a nested relationship, the **OrderDetails** table is a nested child of the **Orders** table, which are both nested children of the **Customers** table.

```
<Customers>
   <CustomerID>ANATR</CustomerID>
   <CompanyName>Ana Trujillo Emparedados y helados</CompanyName>
   <Orders>
      <OrderID>10308</OrderID>
      <OrderDate>1996-09-18</OrderDate>
         <OrderDetails>
            <OrderWeight>20kg</OrderWeight>
            <ShipVia>International Air</ShipVia>
         </OrderDetails>
   </Orders>
</Customers>
```

To create a schema to illustrate a nested relationship

1. Following the instructions in the section Creating Tables in XML Schemas and create three tables. For this example there is no need to define columns, just create the tables and name them: **Customers**, **Orders**, and **OrderDetails**.
2. Click and drag the header of the **Orders** table and drop it onto the **Customers** table.

   An **Orders** column appears in the **Customers** table named **Orders**, and a line appears connecting the two tables.

3. Click and hold the header of the **OrderDetails** table and drop it onto the **Orders** table.

   An **OrderDetails** column appears in the **Orders** table named **OrderDetails**, and a line appears connecting the two tables.

   You can see the nested relationship illustrated on the designer by the lines that connect all three tables together, as well as the existence of the nested child tables as columns in each parent table throughout the hierarchy.

You have just created a schema that represents a nested relationship. More importantly switch to XML view (click the XML tab on the bottom of the design surface) and you can now see the apparent nested structure of the schema.

See Also

Relational Data in XML Schemas | Tables, Columns, Keys, and Constraints in XML Schemas | Nested Relationships in XML | Creating Tables in XML Schemas | Creating Primary and Unique Keys in XML Schemas | XML Schemas and Data | XML Schemas and the XML Designer | Creating XML Schemas and Datasets | XML Files

Walkthrough: Creating a Dataset with Tables, Keys, and One-to-Many Relationships

In this walkthrough you will create a dataset to validate data in a one-to-many relationship (a one-to-many relationship can also be called a *keyed relationship*). It will take you through the process of creating a new Visual Basic or Visual C# project and adding a new Dataset item to it. Then you will create complex type elements that represent data tables with columns. You will learn how to define primary keys in tables. Finally, you will learn how to create relationships between tables based on the defined keys. When complete you will be able to examine the underlying XML code that was created by the XML Designer, in order to understand how datasets express keyed relationships in native XML.

This walkthrough will include creating tables, primary keys, and relationships and will consist of the three major sections outlined below:

1. **Add a Dataset to a project and create three tables.**
   a. Create a Customers table.
   b. Create an Orders table.
   c. Create an OrderDetails table.
2. **Create two primary keys.**
   a. Create a primary key in the Customers table.
   b. Create a primary key in the Orders table.
3. **Create two relationships (DataRelation objects).**
   a. Create a relationship between the Customers table (one) and the Orders table (many).
   b. Create a relationship between the Orders table (one) and the OrderDetails table (many).

Create a New Windows Application Project and Add a New Dataset to it

To create and name a new project

1. From the **File** menu, point to **New**, and then click **Project** to display the **New Project** dialog box.
2. Depending on what language you want to use, select **Visual Basic Projects** or **Visual C# Projects** in the **Project Types** pane, and then select **Windows Application**.
3. Name the project **KeyedRelationshipWalkthrough** and then click **OK** to create the project.

Because this walkthrough requires a dataset, you need to add one to your project.

To add a new Dataset item to the project

1. From the **Project** menu, select **Add New Item**.

   The **Add New Item** dialog appears.

2. From the Templates area of the **Add New Item** dialog box select **Dataset**.
3. Name the Dataset **CustomerOrders** and click **Open**.

   Visual Studio will add a file called **CustomerOrders.xsd** (Visual Basic) and **CustomerOrders.vb** (Visual Basic) or **CustomerOrders.cs** (Visual C#) to the project and automatically load it into the designer.

4. From the **Project** menu, if it is not already, select **Show All Files**. (There will be a visible border around the icon when selected)
5. In Solution Explorer expand the **CustomerOrders.xsd** node.

   You can now see the **CustomerOrders.vb** (Visual Basic) or **CustomerOrders.cs** (Visual C#) file that was created in step 3 above. This is the actual dataset, or more accurately a dataset class. The **CustomerOrders.xsd** file is an XML Schema that will define the structure of the dataset.

Create Three Tables in the Dataset

This section will explain how to add elements to your dataset that will represent data tables.

To create the Customers table

1. From the **XML Schema** tab of the Toolbox, select an **element** and drop it onto the designer.

2. In the top cell to the right of the '**E**' highlight the default name of **element1** and type **Customers**.
3. Using the TAB key, navigate to the center cell of the next row and type **CustomerID**. The data type is string for this element, which is the default so you can just leave it. However, if you needed to assign a data type that was not a string, you would use TAB to navigate to the drop-down list to the right of the desired element and select the appropriate data type.

   Note Also notice how you navigated past the small cell to the left of the cell that said **element1**. That is the cell where you choose other types of elements, such as attributes, and so on. Because the default value is **E** for element we can leave that set as well.

4. Repeat Step 3 to create new rows in the Customers element for the following:

Element name	Data type
CompanyName	**string**
ContactName	**string**
ContactTitle	**string**
Address	**string**
City	**string**
Region	**string**
PostalCode	**string**

To create the Orders table

1. From the **XML Schema** tab of the Toolbox select an element and drop it onto the designer.
2. In the top cell to the right of the '**E**' highlight the default name of **element1** and type **Orders**.
3. Using the TAB key navigate to the center cell of the next row and type **OrderID** and set the data type to string.
4. Repeat Step 3 and create new rows in the Orders element for the following:

Element name	Data type
CustomerID	**string**
OrderDate	**date**
ShippedDate	**date**
ShipVia	**string**

To create the OrderDetails table

1. From the **XML Schema** tab of the Toolbox select an element and drop it onto the designer.
2. In the top cell to the right of the '**E**' highlight the default name of **element1** and type **OrderDetails**.
3. Using the TAB key, navigate to the center cell of the next row and type **OrderID** and set the data type to string.
4. Repeat Step 3 and create new rows in the OrderDetails element for the following:

Element name	Data type
ProductID	integer
UnitPrice	decimal
Quantity	short

Create the Two Primary Keys in the Tables

This section will explain how to designate columns in the tables you created in the previous section as primary keys. In order to create relationships you must have at least one key defined in the parent table (the parent table is the table that represents the one side of a one-to-many relationship).

To create the primary key in the Customers table

1. In the **Customers** table, select the **CustomerID** row by clicking to the left of the **E**.
2. Right-click, point to **Add**, select **New key** from the shortcut menu.
3. The **Edit Key** dialog box appears.
4. Change the name of the key to **CustomersIDKey**.
5. The Element drop-down list box should be set to **Customers**, which indicates that you are in the Customers table.

   Note The **Customers** table is really the
   Customers
   element that is defined as a complex type; it is complex type elements that represent tables in XML Schemas and

Datasets. For more information see Creating One-to-Many Relationships in XML Schemas and Datasets.

6. In the Fields area, select **CustomerID** from the drop-down list. If you right-clicked on the **CustomerID** row to add the key this should be the default value, if you clicked on a different row you will need to select the **CustomerID** element.

   Note The **Fields** area is where you select which element (within the complex type definition of your table element) you want to define as the Key.

7. Select the **Dataset primary key** check box, to define this key as a primary key.

   Note If you did not select the **Dataset primary key** checkbox then the key would be defined as a unique key, instead of a primary key.

8. Click **OK** to close the Edit Key dialog box.

   A key icon is placed on the **CustomerID** row to identify it as a key.

To create the primary key in the Orders table

1. In the **Orders** table select the **OrderID** row by clicking to the left of the **E**.
2. Right-click the selected row, point to **Add**, and select **New key** from the shortcut menu.

   The **Edit Key** dialog box appears.

3. Change the name of the key to **OrdersIDKey**.
4. The Element drop-down should be set to **Orders**, which indicates that you are in the Orders table.
5. In the Fields area, select **OrderID** from the drop down list. If you right-clicked on the **OrderID** row to add the key, this should be the default value. If you clicked on a different row, you will need to select the correct element.
6. Select the **Dataset primary key** checkbox to define this key as a primary key.
7. Click **OK** to close the Edit Key dialog box.

   A key icon is placed on the **OrderID** row to identify it as a key.

You now have the keys defined that will be used to define the relationships between the tables. You might be wondering why we did not define a key in the **OrderDetails** table. At the beginning of this section it said, "In order to create relationships you must have at least one key defined in the parent table (the parent table is the table that represents the one side of a one-to-many relationship)." The **OrderDetails** table does not represent the one side in any relationships in this dataset, therefore no key definition is required.

Create the Two Relationships Between the Tables

A relationship is the association between a keyed column in one table and the many records that have an associated column in another table. In XML Schemas relationships are defined with the `keyref` element. This section will explain how to define these relationships.

To create a relationship (DataRelation object) between the Customers and Orders table

1. Right-click anywhere on the **Orders** table, point to **Add**, and select **New Relation** from the shortcut menu.

   The **Edit Relation** dialog box appears.

   Note Be sure to add the New Relation to the table representing the many side of a one-to-many relationship (in this case add the **Relation** to the **Orders** table).

2. The **Edit Relation** dialog box assigns a default name of **CustomersOrders**. You can leave this as is.
3. The **Parent Element** should be set to **Customers**. The parent is the element (or table) that represents the one side in a one-to-many relationship. In this case the **Customers** table represents the one side.
4. The **Child Element** should be set to **Orders**. The child is the element (or table) that represents the many side in a one-to-many relationship. In this case, the **Orders** table represents the many side.
5. The Key should be set to **CustomersIDKey** that we defined earlier.

   Note When creating a one-to-many relationship the key definition needs to be in the Parent table, because that is the table that contains the unique value.

6. The key fields are read only and contain the key column or columns defined in the key definition.

Note Multipart keys are key definitions that have multiple columns in a table that are part of the key definition. If a multipart key were defined in **CustomersIDKey** then all the columns (or fields) would be listed in the key fields area. This walkthrough does not include any multipart keys.

7. The Foreign Key Fields area is where you select the field in the child table that is equivalent to the key in the Parent table. This field should be set to **CustomerID** to match the key field in the parent table.
8. Click **OK** to accept the remaining default settings.

   A **DataRelation** object is created and a visual representation appears on the design surface.

To create a relationship (DataRelation object) between the Orders and OrderDetails table

1. Right-click anywhere on the **OrderDetails** table, point to **Add**, and select **New Relation** from the shortcut menu.

   The **Edit Relation** dialog box appears.

   Note Be sure to add the New Relation to the table representing the many side of a one-to-many relationship (in this case add the Relation to the **OrderDetails** table).

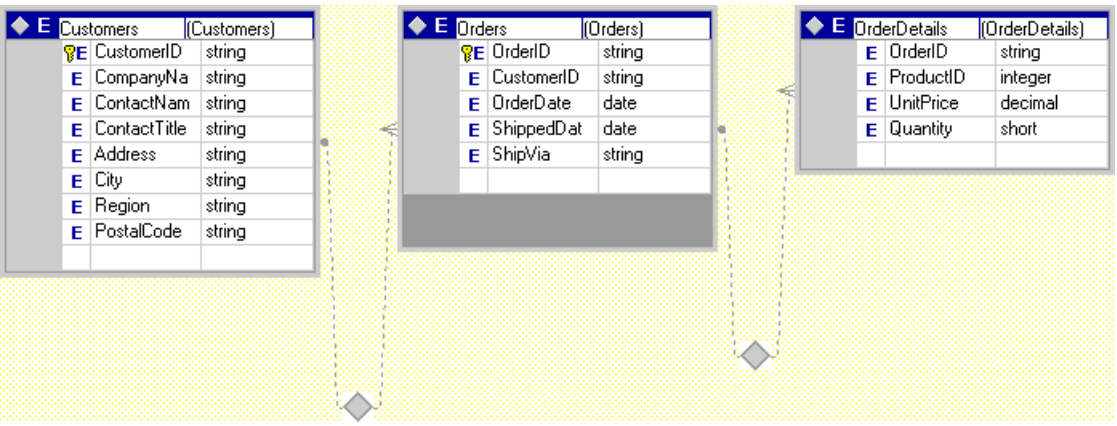
2. Change the default name of **CustomersOrderDetails** to **OrdersOrderDetails** to reflect the proper tables in the relationship.
3. The **Parent Element** should be set to **Orders**.
4. The **Child Element** should be set to **OrderDetails**.
5. The **Key** should be set to **OrdersIDKey**, which was defined earlier.

   The key fields are read only and contain the key column or columns defined in the key definition selected in step 5.

6. The **Foreign Key Fields** area is were you select the field in the child table that is equivalent to the key in the Parent table. Verify that this field is set to **OrderID** to match the key field in the parent table.
7. Click **OK** to accept the remaining default settings.

   A **DataRelation** object is created and a visual representation appears on the design surface.

This is how your dataset should look in the XML Designer:



You have just created three related tables in a Dataset. If you swtich to XML view, your code should be equivalent to the following. What is important to note is how native XML is structured in a one-to-many (or keyed) relational model. As you can see the `Customers`, `Orders`, and `OrderDetails` all exist hierarchically at the same level, as direct children of the `CustomerOrders` element. It is this `CustomerOrders` element that represents the dataset.

Note Notice how the keys are defined hierarchically as children of the CustomerOrders element. It is very important to understand why the key definitions are not located as nested children within the element that represents the table the key is defined in. If that were the case, the key would only be unique for each record, not for all records in the table. By having the key defined within the dataset itself (the CustomerOrders element) the key controls uniqueness throughout the entire dataset.

```
<xs:schema id="CustomerOrders" targetNamespace="http://tempuri.org/CustomerOrders.xsd" elemen
tFormDefault="qualified" xmlns="http://tempuri.org/CustomerOrders.xsd" xmlns:mstns="http://te
mpuri.org/CustomersOrders.xsd" xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:msdata="urn:
```

```
schemas-microsoft-com:xml-msdata">
  <xs:element name="CustomerOrders" msdata:IsDataSet="true">
   <xs:complexType>
     <xs:choice maxOccurs="unbounded">
      <xs:element name="Customers">
       <xs:complexType>
        <xs:sequence>
         <xs:element name="CustomerID" type="xs:string" minOccurs="0" />
        <xs:element name="CompanyName" type="xs:string" minOccurs="0" />
        <xs:element name="ContactName" type="xs:string" minOccurs="0" />
       <xs:element name="ContactTitle" type="xs:string" minOccurs="0" />
        <xs:element name="Address" type="xs:string" minOccurs="0" />
         <xs:element name="City" type="xs:string" minOccurs="0" />
          <xs:element name="Region" type="xs:string" minOccurs="0" />
         <xs:element name="PostalCode" type="xs:string" minOccurs="0" />
       </xs:sequence>
      </xs:complexType>
     </xs:element>
    <xs:element name="Orders">
     <xs:complexType>
       <xs:sequence>
        <xs:element name="OrderID" type="xs:string" minOccurs="0" />
        <xs:element name="CustomerID" type="xs:string" minOccurs="0" />
        <xs:element name="OrderDate" type="xs:date" minOccurs="0" />
        <xs:element name="ShippedDate" type="xs:date" minOccurs="0" />
        <xs:element name="ShipVia" type="xs:string" minOccurs="0" />
       </xs:sequence>
      </xs:complexType>
     </xs:element>
    <xs:element name="OrderDetails">
      <xs:complexType>
       <xs:sequence>
         <xs:element name="OrderID" type="xs:string" minOccurs="0" />
         <xs:element name="ProductID" type="xs:integer" minOccurs="0" />
         <xs:element name="UnitPrice" type="xs:decimal" minOccurs="0" />
         <xs:element name="Quantity" type="xs:short" minOccurs="0" />
        </xs:sequence>
       </xs:complexType>
   </xs:element>
  </xs:choice>
 </xs:complexType>
      <xs:key name="CustomersIDKey" msdata:PrimaryKey="true">
         <xs:selector xpath=".//mstns:Customers" />
         <xs:field xpath="mstns:CustomerID" />
      </xs:key>
      <xs:key name="OrdersIDKey" msdata:PrimaryKey="true">
         <xs:selector xpath=".//mstns:Orders" />
         <xs:field xpath="mstns:OrderID" />
      </xs:key>
      <xs:keyref name="CustomersOrders" refer="CustomersIDKey">
         <xs:selector xpath=".//mstns:Orders" />
         <xs:field xpath="mstns:CustomerID" />
      </xs:keyref>
      <xs:keyref name="OrdersOrderDetails" refer="OrdersIDKey">
         <xs:selector xpath=".//mstns:OrderDetails" />
         <xs:field xpath="mstns:OrderID" />
      </xs:keyref>  </xs:element>
</xs:schema>
```

See Also

Creating One-to-Many Relationships in XML Schemas and Datasets | Relational Data in XML Schemas | Creating Tables in XML Schemas | Creating Primary and Unique Keys in XML Schemas | Deleting Primary Keys and Unique Keys | Creating DataRelation Objects with the XML Designer | Introduction to Datasets | Relationships in ADO.NET Datasets

XML Files

XML files contain both the actual data (or content) and the elements that identify the data. The XML Designer provides two ways to view and work with XML documents, XML view and Data view. XML view, as its name implies, is actual XML code (or XML markup). Data view interprets the XML source and displays the contents in a data grid. Using the XML designer in XML view provides both color-coding, which makes the code easy to read, and IntelliSense, for helping to locate syntax errors. Additionally, if a schema is associated with the XML file the statement completion feature is enabled, as well as an option to validate the XML data against the schema.

> **Note** There are some instances of XML documents where Data view is disabled. This occurs when the XML data cannot be mapped into the dataset. If your XML document is structured in this way, changing to Data view will present a message stating; "Although this XML document is well formed, it contains structure that Data View cannot display."

In This Section

Creating New XML Files
Describes how to add new XML files to a project.
Assigning Schemas to XML Files
Describes how to set the **targetSchema** property, in order to associate a schema to an XML file.
Checking for Well-Formed and Valid XML
Describes how to confirm XML files have correct syntax and content.
Creating XML Schemas from XML Files
Describes how to use the XML Designer to infer a schema based on existing XML data.
Walkthrough: Creating an XML File with an Associated XML Schema
Provides step-by-step instructions for how to create an XML Schema and associated XML file, including element, type definitions, and setting the **targetSchema** property.

Related Sections

XML Schemas and Data
Introduces topics and sections that describe how to use the XML Designer to work with XML Schemas and datasets.
XML Schemas and the XML Designer
Introduces topics that describe the different elements in XML Schemas and how they are created using the XML Designer.
Creating XML Schemas and Datasets
Introduces topics that describe the different ways that schemas and datasets can be created using the XML Designer, as well as describing the connection between datasets and schemas.
Relational Data in XML Schemas
Introduces topics that describe the different options in XML Schemas for representing relational data.
XML in Visual Studio
Provides links to a variety of topics about XML.
World Wide Web Consortium (W3C) Web site (http://www.w3.org/)
Links to the XML Schema primer and other XML info posted by the W3C.
XML Developer Center on MSDN Online (http://msdn.microsoft.com/xml/default.asp)
Provides various resources about XML.
XML Schema Reference (XSD)
Provides reference topics for schema elements and data types.
XML Designer
Provides an overview of the XML Design tool in Visual Studio.

Creating New XML Files

Because XML is an applicable format for so many different applications, there is no one specific reason why someone would create a new XML document. If you need to create an XML file from scratch, there are a couple of XML Designer specifics to take into account.

- There are some instances of XML documents where Data view is disabled. This occurs when the XML source cannot be logically interpreted into structured data. If your XML document is structured in this way, changing to Data view will present a message stating, "Although this XML document is well formed, it contains structure that Data view cannot display."
- When creating new XML documents, you always begin in XML view, working with XML source code. After you have added the initial tags and content, if your XML document can be interpreted into structured data, you can then change to Data view and use the data grid to add data to your XML file.

To create an XML file in Visual Studio

1. From the **Project** menu, select **Add New Item.**
2. In the **Add New Item** dialog box, select **XML File**.
3. Type a name for your XML file.

   Visual Studio adds a new, blank XML file (.xml) to the project, automatically loads it into the XML Designer, and presents it in XML view.

4. Add XML content to your file.

Next Steps

After you have created your XML file, you may want to:

- Create a schema and dataset based on the structure of your XML. For more information, see Creating XML Schemas from XML Files.
- Associate a schema with it. For more information, see Assigning Schemas to XML Files.
- Verify that it is valid against a schema. For more information, see Checking for Well-Formed and Valid XML.

See Also

XML Files | Assigning Schemas to XML Files | Checking for Well-Formed and Valid XML | Creating XML Schemas from XML Files | Relational Data in XML Schemas | Creating XML Schemas and Datasets

Assigning Schemas to XML Files

You assign a schema to an XML file when you want to verify that the content of an XML file is made of the proper data types and is presented in the proper structure. In order to validate an XML file with a schema, you must create a link between the schema file and the XML file. This link is defined in the **targetSchema** property of the .xml file and when set is added as a namespace reference within the contents of the XML document.

To create the association between schemas and XML documents

1. Load an XML file into the XML Designer.
2. Change to XML view by clicking the **XML** tab on the bottom of the designer.
3. In the Properties window select **DOCUMENT** from the drop-down list.
4. Set the **targetSchema** property to the schema to be associated with the XML document.

> **Note** If the XML Schema is a file in your project it will be available in the **targetSchema** property drop-down list. If it is not in your project, type the Uniform Resource Identifier (URI) of the schema that will be used to validate your XML document.

To remove the association between an XML Schema and an XML document

1. Load the XML file into the XML Designer.
2. Change to XML view by clicking the **XML** tab at the bottom of the designer.
3. In the Properties window, select **DOCUMENT** from the drop-down list.
4. Clear the contents of the **targetSchema** property.

Next Steps

Now that you have associated a schema to an XML file, you may want to:

- Validate your XML file. For more information, see Checking for Well-Formed and Valid XML.
- Create a dataset from the schema that is associated with the XML file. For more information, see Creating ADO.NET Typed DataSets from Schemas.

See Also

XML Files | Creating New XML Files | Checking for Well-Formed and Valid XML | Creating XML Schemas from XML Files | Relational Data in XML Schemas | Creating XML Schemas and Datasets | Introduction to Data Access with ADO.NET | Introduction to Datasets

Checking for Well-Formed and Valid XML

There are two inherently different ways to verify whether XML files are coded correctly. All XML files must be well formed, and if an XML file has an associated schema, then the structure and content must also be deemed valid.

- **Well-formed** – The XML code must be syntactically correct or the XML parser will raise an error.
- **Valid** – If the XML file has an associated XML Schema, the elements must appear in the defined structure and the content of the individual elements must conform to the declared data types specified in the schema.

The XML Designer can be used to verify that XML files are both well formed and valid.

To check for well-formed or valid XML

> **Note** In order to check that XML files are valid, they must have an associated XML Schema. For more information on associating a schema with your XML file, see Assigning Schemas to XML Files.

1. Load the XML file you want to check into the XML Designer.
2. From the **XML** menu select **Validate XML Data**.
3. The status bar will indicate if errors were found in your XML file.
4. If errors were found, view the Task List For more information.

See Also

XML Files | Creating New XML Files | Assigning Schemas to XML Files | Creating XML Schemas from XML Files | Relational Data in XML Schemas | Creating XML Schemas and Datasets | Introduction to Data Access with ADO.NET | Introduction to Datasets

Creating XML Schemas from XML Files

You create schemas based on existing XML files when you want to validate that other XML files conform to the same data structure and data type restrictions as the original XML file.

> **Tip** To create an ADO.NET dataset from an XML file, you must first create a schema and then generate a dataset from that schema. For more information, see Creating ADO.NET Typed Datasets from Schemas.

> **Note** There are some instances of XML documents that schemas and datasets cannot be created from. Although the XML would be considered well formed, trying to create a schema will generate a **Failed to Create Schema** error if the contents cannot be logically interpreted into structured data.

To create a new XML Schema based on an existing XML Document

1. Load an XML document (.xml file) into the XML Designer.
2. From the **XML** menu click **Create Schema**.

   An XML Schema (.xsd file) is added to the current project with the same name as the original XML file.

3. Load the newly created XML Schema (.xsd file) into the XML Designer.
4. Verify and edit the data types that were assigned when creating the schema.

   > **Note** When a schema is inferred from an existing XML document, all the data types are initially set to **string**, so you must edit the data types according to the content requirements of your XML data.

5. If you need to make changes to the schema, you can use the XML Designer to add, edit, and remove elements.

Next Steps

- If the schema you just created requires primary or unique keys defined then you should create those now. For more information, see Creating Primary and Unique Keys in XML Schemas.
- If the schema you just created requires relationships defined between tables then you should create those now. For more information, see Creating DataRelation Objects with the XML Designer.
- Since the XML Designer can create datasets based on the structure of existing XML Schemas, you can now generate a new dataset based on the schema you created in the previous steps. For more information, see Creating ADO.NET Typed Datasets from Schemas.

See Also

XML Files | Creating New XML Files | Assigning Schemas to XML Files | Checking for Well-Formed and Valid XML | Relational Data in XML Schemas | Creating XML Schemas and Datasets | Introduction to Data Access with ADO.NET | Introduction to Datasets

Walkthrough: Creating an XML File with an Associated XML Schema

This walkthrough will illustrate the capabilities of working with XML files that have associated schemas. It will take you through the process of creating an XML file and an XML Schema based on the XML Schema definition language (XSD). Then you will create an association between the XML Schema and the XML file. After this association is established, you will work on the XML file in XML view in order to see the advanced capabilities of the XML editor. For this walkthrough the XML file will represent a list of companies. You will see how the XML file contains the data whereas the XML Schema only defines that data.

> **Note** By associating an XML Schema with your XML file, statement completion is enabled in the XML editor.

Creating a New Windows Application Project

To begin this walkthrough you will first need to create a new Windows Application in Visual Basic or Visual C#.

To create a new project and name it "XMLCustomerList"

1. From the **File** menu, point to **New**, and then click **Project** to display the **New Project** dialog box.
2. Depending on what language you want to use, select **Visual Basic Projects** or **Visual C# Projects** in the **Project Types** pane, and then select **Windows Application**.
3. Name the project **XMLCustomerList**, then click **OK** to create the project.

   Visual Studio will add the XMLCustomerList project to Solution Explorer.

Add a New XML File item to your Project

Because this walkthrough requires an XML file, you need to add one to your project.

To add a new XML file item to the project

1. From the **Project** menu, select **Add New Item**.

   The **Add New Item** dialog box appears.
2. In the Templates area of the **Add New Item** dialog box, select **XML File**.
3. Name the XML file **CustomerList**, and then click **Open**.

   Visual Studio will add a file called **CustomerList.xml** to the project and automatically load it into the designer.

Add a New XML Schema item to your Project

Because this walkthrough requires an XML Schema, you need to add one to your project.

To add a new XML Schema item to the project

1. From the **Project** menu, select **Add New Item**.

   The **Add New Item** dialog appears.
2. In the Templates area of the **Add New Item** dialog box select **XML Schema**.
3. Name the schema **CustomerListSchema** and then click **Open**.

   Visual Studio will add a file called **CustomerListSchema.xsd** to the project and automatically load it into the designer.

Add a Simple Type Definition to your Schema

Now you need to define the elements that will contain the data in your XML file. Because an XML Schema defines the data in an associated XML file, you create the element definitions in the schema.

The first definition will be a simple type element that will be used to define a standard US postal code (ZIP code). For this walkthrough we will use 5-digit codes only.

To create a simple type element that represents a 5 digit postal code

1. From the **XML Schema** tab of the Toolbox, drag a **simpleType** onto the design surface.
2. Select the default name **simpleType1** and rename this type to **postalCode**.

3. Use the TAB key to navigate one cell to the right and select **positiveInteger** from the drop-down list.
4. Use the TAB key to navigate to the next row.
5. Click the drop-down box.

   The only choice is facet. This is because simple types cannot include elements or attributes as part of their content models. Only facets can be used to build simple types.

6. Use the TAB key to navigate one cell to the right and select **pattern** from the drop-down list.
7. TAB over one cell to the right again and type **\d{5}**.

   The pattern facet allows you to enter regular expressions. The regular expression \d{5} says that the contents of the postalCode type is restricted to 5 digits. Regular expressions are beyond the scope of this walkthrough, but you can see how you can use the pattern facet in conjunction with the chosen data type to only allow specific data in a simple type.

   If you switch your schema to XML view you should see the following code within the root level schema tags (that means the code sample does not include the actual declaration part of the schema, nor does it include the actual schema tags which are called the root or document level tags):

```
<xs:simpleType name="postalCode">
   <xs:restriction base="xs:positiveInteger">
      <xs:pattern value="\d{5}" />
   </xs:restriction>
</xs:simpleType>
```

8. From the **File** menu, select **Save All**.

Add a Complex Type Definition to your Schema

The next definition will be a complex type element that will be used to define an address. As part of this complex type definition, we will use the simple type created in the previous steps.

To create a complex type element that represents a standard US address

1. Switch back to **Schema** view.
2. From the **XML Schema** tab of the Toolbox, drag a **complexType** onto the design surface.
3. Select the default name **complexType1** and rename this type to **usAddress**. Do not select a data type for this element.
4. Using the TAB key, navigate to the next row.
5. Click the drop-down list box to see the many choices of elements you can add to a complex type. You can select element, but for the rest of the walkthrough you will just TAB over this cell because element is the default.
6. Using the TAB key, navigate one cell to the right and type **Name**.
7. TAB one cell to the right and set the data type to **string**.
8. Repeat Steps 4 – 7 and create new rows in the `usAddress` element for the following:

Element name	Data type
Street	string
City	string
State	string
Zip	postalCode

   Notice the data type that is assigned to the Zip element. It is the `postalCode` simple type you created previously.

   If you switch to XML view, you should see the following code within the root level schema tags (that means the code sample does not include the actual declaration part of the schema, nor does it include the actual schema tags which are called the root or document level tags):

```
<xs:simpleType name="postalCode">
   <xs:restriction base="xs:positiveInteger">
      <xs:pattern value="\d{5}" />
   </xs:restriction>
</xs:simpleType>
<xs:complexType name="usAddress">
```

```
      <xs:sequence>
         <xs:element name="Name" type="xs:string" />
         <xs:element name="Street" type="xs:string" />
         <xs:element name="City" type="xs:string" />
         <xs:element name="State" type="xs:string" />
         <xs:element name="Zip" type="postalCode" />
      </xs:sequence>
   </xs:complexType>
```

You have now defined two separate types that can be used in element definitions as well as types.

9. From the **File** menu, select **Save All**.

Add the Main Elements to your Schema

Now that you have defined some data types lets construct the actual data definition for the XML file you will be creating. The XML file will contain the data for the customer list, so lets create the actual element that will define the data that will be valid in the XML file.

To create the Customer element

1. Switch to **Schema** view.
2. From the **XML Schema** tab of the Toolbox, drag an **element** onto the design surface.
3. Select the default name **element1** and rename this to **customer**. Do not select a data type for this element.
4. Using the TAB key navigate to the center cell of the next row, and type **CompanyName**.
5. TAB one cell to the right and set the data type to **string**.
6. Repeat Steps 4 – 5 and create new rows in the `Customer` element for the following:

Element name	Data type
ContactName	string
Email	string
Phone	string
BillToAddress	usAddress
ShipToAddress	usAddress

Notice the data type that is assigned to the BillToAddress as well as the ShipToAddress elements. It is the `usAddress` complex type created previously. We could have defined simple types for the Email, Phone elements and so on.

If you switch your schema to XML view, you should see the following code within the root-level schema tags (that means the code sample does not include the actual declaration part of the schema, nor does it include the actual schema tags which are called the root or document level tags):

```
   <xs:simpleType name="postalCode">
      <xs:restriction base="xs:positiveInteger">
         <xs:pattern value="\d{5}" />
      </xs:restriction>
   </xs:simpleType>
   <xs:complexType name="usAddress">
      <xs:sequence>
         <xs:element name="Name" type="xs:string" />
         <xs:element name="Street" type="xs:string" />
         <xs:element name="City" type="xs:string" />
         <xs:element name="State" type="xs:string" />
         <xs:element name="Zip" type="postalCode" />
      </xs:sequence>
   </xs:complexType>
   <xs:element name="Customer">
      <xs:complexType>
         <xs:sequence>
            <xs:element name="CompanyName" type="xs:string" />
            <xs:element name="ContactName" type="xs:string" />
```

```
            <xs:element name="Email" type="xs:string" />
            <xs:element name="Phone" type="xs:string" />
            <xs:element name="ShipToAddress" type="usAddress" />
            <xs:element name="BillToAddress" type="usAddress" />
         </xs:sequence>
      </xs:complexType>
   </xs:element>
```

7. From the **File** menu, select **Save All**.

In order to allow multiple instances of customer data within the XML document, we will create an element named `customerList` that will contain all the individual `customer` elements.

To create the customerList element

1. From the **XML Schema** tab of the Toolbox, drag an **element** onto the design surface.
2. Select the default name **element1** and rename this to **customerList**. Do not select a data type for this element.
3. Select the **customer** element (created previously) and drag it onto the **customerList** element.

   The individual design grids bind to represent the hierarchical structure of the data.

4. From the **File** menu, select **Save All**.

Associating the Schema and the XML file

In this part of the walkthrough, you will start working with the XML file. Within the XML file you will add a reference to the schema you just created.

To create an association between the XML file and the XML Schema

1. In Solution Explorer double-click the CustomerList.xml file.
2. The XML file opens in the designer in XML view.
3. In the Properties window, click the cell to the right of the **targetSchema** property and select **http://tempuri.org/CustomerListSchema.xsd**.

   Visual Studio adds a reference to the schema in the CustomerList.xml file and adds the `<customerList>` tags.

Adding Data to your XML file

The XML file is now ready for data. By associating a schema with your XML file, the XML editor now is aware of the valid elements that can be included in your XML file, as well as providing a fromatted grid in data view.

To add data to the customerList.xml file

1. In **XML** view of the **customerList.xml** file, position the cursor between the opening and closing `<customerList>` tags (Opening tag = `<customerList>` Closing tag = `</customerList>`).
2. Type **<**. Select the `Customer` element.
3. Type **>** to close the tag.
4. Type **<**, and select **CompanyName** from the list of valid elements.
5. Type **>** to close the tag.
6. Type **Blue Yonder Airlines** as the company name.
7. Switch to **Data** view.
8. Type **Nate Sun** in the **ContactName** field in the grid.
9. Complete the record by adding data into the other fields in the grid.
10. Switch back to **XML** view.

    The data from the grid is now properly formatted as XML.

Next Steps

Some common next steps include:

- Add some data in XML view and then switch to Data view. Explore the different ways to add data to your XML files using the

XML Designer.

- After adding some data, validate your document.
- Generate a dataset from your schema.

See Also

XML Schemas and Data | XML Files | Creating New XML Files | Assigning Schemas to XML Files | Checking for Well-Formed and Valid XML | XML Elements, Attributes, and Types | Creating ADO.NET Typed DataSets from Schemas

Programming with Components

Components shorten programming time and produce more robust applications, by allowing you to assemble applications from tested, standardized components. The designer architecture in Visual Studio .NET lets you assemble nonvisual component classes as easily as you assemble Visual Basic forms. The following sections explain how to create your own components, and how to assemble them from the sophisticated components in the .NET Framework classes.

In This Section

Component Programming Walkthroughs
Provides a list of the walkthroughs available for component creation and implementation.
Code for Component Programming
Provides a list of code examples for common tasks.
Troubleshooting Control and Component Authoring
Explains how to fix common problems.
Component Authoring
Describes component architecture, terminology, design, and implementation.
COM Interoperability in Visual Basic and Visual C#
Provides links to information about the concepts and implementation details for interoperating between COM and .NET objects.
Creating Messaging Components
Introduces components that interact with Microsoft Message Queuing system resources.
Creating File System and Timer Components
Introduces components that respond to changes to files and directories, raise events on a schedule, or monitor and interact with Windows processes.
Creating System Monitoring Components
Introduces components that interact with Windows performance counters and event logs.
Creating Active Directory Components
Introduces components that interact with the Active Directory, allowing you to perform administrative directory tasks within your application.
Creating Installation Components
Introduces installation components, which allow you to execute custom actions when your application is deployed.
Programming with Office
Provides information on using Microsoft Office and Visual Studio .NET as part of your business application.

Related Sections

Decision Chart
Provides a graphical, linked guide to decisions about designing your application.
Visual Basic Code Example Topics
Lists quickstart-like code examples that help you perform common tasks in Visual Basic .NET.
Inside the .NET Framework
Provides links to introductory topics about the common language runtime, managed code, and other essential concepts.
Component Model Namespaces in Visual Studio
Summarizes the primary namespaces in the .NET Framework that are for creating controls and components, and provides links to related reference and conceptual topics.
Framework Services Namespaces in Visual Studio
Summarizes the primary namespaces in the .NET Framework for interacting with a variety of back-end server resources (such as message queues and directory services) and provides links to related reference and conceptual topics.
Debugging Managed Code
Provides information about how to debug Visual Basic and C# code.
Deploying Applications and Components
Provides links to procedural, conceptual, and step-by-step topics about component deployment.

Component Programming Walkthroughs

Walkthroughs give step-by-step instructions for common scenarios, which makes them a good place to start learning about the product or a particular feature area.

The following walkthroughs will help orient you to the use of components, user controls, and threading in your Visual Basic and Visual C# applications:

Authoring a Component with Visual Basic .NET
: Shows the development of a simple component in Visual Basic, illustrating the interaction between client and component, object lifetime and circular references, debugging clients and components, and the use of shared methods and instance methods.

Authoring a Component with Visual C#
: Shows the development of a simple component in Visual C#, illustrating the interaction between client and component, object lifetime and circular references, debugging clients and components, and the use of shared methods and instance methods.

Creating Your Own Collection Class
: Shows how you can create your own collection classes by inheriting from one of the many .NET Framework collection classes and adding code to implement your own custom functionality.

Authoring a User Control with Visual Basic .NET
: Shows the development of a simple control in Visual Basic that inherits from the **UserControl** class, and how to inherit from an established user control.

Authoring a User Control with Visual C#
: Shows the development of a simple control in Visual C# that inherits from the **UserControl** class, and how to inherit from an established user control.

Inheriting from a Windows Forms Control with Visual Basic .NET
: Shows the development of a simple control in Visual Basic that inherits from a pre-existing Windows Forms control.

Inheriting from a Windows Forms Control with Visual C#
: Shows the development of a simple control in Visual C# that inherits from a pre-existing Windows Forms control.

Authoring a Simple Multithreaded Component with Visual Basic
: Demonstrates the development of a simple multithreaded component in Visual Basic, explaining how threads work and how to coordinate multiple threads in your component.

Authoring a Simple Multithreaded Component with Visual C#
: Demonstrates the development of a simple multithreaded component in Visual, explaining how threads work and how to coordinate multiple threads in your component.

Creating a Queue and Working with Messages
: Describes how to use a **MessageQueue** component instance to interact with Windows message queues.

Creating a Message Queue Transaction with COM+ Services
: Shows how you can use COM+ services to wrap your message in a transaction.

Exploring Event Logs, Event Sources, and Entries
: Describes how to use an **EventLog** component instance to interact with existing logs, create custom logs, and read and write values to logs.

Retrieving Categories and Counters
: Describes how to use a **PerformanceCounter** component instance to retrieve lists of Windows performance counter categories on a server.

Changing and Retrieving Performance Counter Values
: Describes how to use a **PerformanceCounter** component instance to retrieve and set raw and calculated values from a Windows performance counter.

Managing a Windows Process
: Describes how to use the **Process** component to monitor and interact with Windows processes on a server.

Reacting to File System Events
: Describes how to use a **FileSystemWatcher** component instance to monitor files and directories and react when changes occur.

Adding ActiveDirectory Objects
: Describes how to use the **DirectorySearcher** component to interact with Active Directory file system structures.

Installing an Event Log Component
: Describes how to use an installation component to configure server resources needed for your **EventLog** component instance.

See Also

Walkthroughs in Visual Basic and Visual C# | Visual Studio Walkthroughs | Samples and Walkthroughs |
Visual Basic Code Example Topics

Code for Component Programming

This topic is designed to help you find code that demonstrates how to perform common component programming tasks in Visual Basic .NET. The following categories are represented in the list below:

• Component Authoring • Multithreading • Containers and Sites • Exception Handling • Component Security • User Assistance • User and Custom Controls	• Messaging Components • File System and Timer Components • Performance Counter Components • Event Log Components • Services and Processes • Active Directory Components • Related Code Examples

You can also find sample code for component programming in the following locations:

- General Programming Example Topics
- General .NET Framework Example Topics
- Visual Basic Sample Applications
- Visual Basic QuickStarts and Other Samples
- .NET Framework Reference Information

Component Authoring

Example of...

Creating a simple component

Declaring a method in a component

Declaring an overloaded method in a component

Declaring and raising events in components

Creating named constants for components

Implementing properties in components

Implementing default properties in components

Exposing properties of contained objects

Saving and retrieving the state of a component

Creating abstract classes

Creating interfaces

Implementing existing interfaces

Displaying forms from components

Allowing for communication between components in an assembly

Containers and Sites

Example of...

Creating component containers

Extending component containers

Implementing nested classes

Linking dependent objects

Implementing collection properties

Multithreading

Example of...

Creating a simple multithreaded component

Creating a new thread of execution

Creating a lock on a thread

Writing to event logs from multithreaded components

Manipulating controls from threads

Exception Handling

Example of...

Throwing exceptions from components

Handling exceptions in components

Component Security

Example of...

Adding security attributes to your components

Adding imperative security checks to your components

User Assistance

Example of...

Displaying a help file for your component

Providing descriptive metadata for your component

Providing descriptive metadata for component properties, methods, and events

User and Custom Controls

Example of...

Creating a user control for Windows Forms applications

Inheriting from a Windows Forms control

Exposing Properties of Constituent Controls

Implementing Methods in Custom Controls

Painting and Rendering Custom Controls

Painting and Rendering Constituent Controls

Making Controls Invisible at Run Time

Docking and aligning controls

Adding your controls to the toolbox

Adding tooltips to your controls

Giving your control a transparent background

Messaging Components

Example of...

Creating a simple message queuing application

Creating public and private queues

Purging a queue's contents

Instantiating components to interact with a message queue

Sending simple messages

Sending complex messages

Sending messages to a journal queue

Specifying the format for received messages

Taking messages out of a queue

Peeking at queue contents

Receiving messages asynchronously

Sending transactional messages

Retrieving lists of available messages

Retrieving lists of available queues

File System and Timer Components

Example of...

Creating a simple application that monitors the file system

Instantiating components to monitor files or directories

Specifying what to monitor for

Handling file system events

Pausing a thread for continuation when a file system event occurs

Creating and configuring server-based timers

Monitoring server-based timers

Performance Counter Components

Example of...

Creating a simple application that retrieves existing categories and counters

Creating a simple application that writes to a counter

Instantiating components to interact with performance counters

Specifying the counter to monitor

Creating custom counters

Creating counter categories

Determining if a counter or category already exists

Adding and removing counter instances

Writing values to performance counters

Retrieving raw values from a performance counter

Retrieving calculated values from a performance counter

Sampling performance counter values

Retrieving lists of categories and counters

Event Log Components

Example of...

Determining if an event log exists

Creating custom event logs

Deleting event logs

Clearing event log entries

Instantiating a component to interact with an event log

Adding your application as a source of events

Writing entries to event logs

Reading entries from event logs

Handling event log events

Services and Processes

Example of...

Instantiating components to interact with services

Stopping, starting, and pausing services

Retrieving lists of services

Connecting to processes on another computer

Binding to existing processes

Starting processes

Stopping processes

Determining if a process is responding

Determining if a process has exited

Viewing running processes

Waiting for a process to complete an action

Investigating memory usage for a process

Investigating thread usage for a process

Creating a simple application that monitors a process

Active Directory Components

Example of...

Instantiating a component to interact with an Active Directory heirarchy

Creating nodes in a heirarchy

Removing heirarchy nodes

Searching a heirarchy

Creating a simple application that creates Active Directory objects

Related Code Examples

Code for Windows Forms Controls

Code Examples for Visual Basic Programmers

Troubleshooting Control and Component Authoring

This topic lists common problems that arise when developing components and controls. For more information, see Authoring Components.

Cannot Add Control to Toolbox

If you want to add a custom control that you created in another project or a third-party control to the Toolbox, you must do so manually.

To add a control to the Toolbox

1. Right-click the **Toolbox** and choose **Add/Remove Items** from the shortcut menu.

   The **Customize Toolbox** dialog box opens.

2. Add the component:

   - If you want to add a .NET component or control, click the **.NET Framework Components** tab.

     – or –

   - If you want to add a COM component or ActiveX control, click the **COM Components** tab.

3. If your control is listed in the dialog box, make sure it is selected, then click OK.

   The control is added to the Toolbox.

4. If your control is not listed in the dialog box, do the following:
   a. Click the **Browse** button.
   b. Browse to the folder that contains the .dll file that contains your control.
   c. Select the control and click **Open**.

      Your control appears in the dialog box.

   d. Make sure your control is selected, then click OK.

      Your control is added to the Toolbox.

Cannot Debug My Windows Forms User Control or Component

Windows Forms user controls, custom controls, and components are not stand-alone projects. They must be hosted by an application such as a Windows Forms project. To debug a control or component, you must add it to a Windows Forms project.

To debug a control or component

1. From the **Build** menu, click **Build Solution** to build your solution.
2. From the **File** menu, choose **Add Project** then **New Project** to add a test project to your application.
3. In the **Add New Project** dialog box choose **Windows Application** for the type of project.
4. Right-click the References node and click **Add Reference** on the shortcut menu to add a reference to the project containing the control or component.
5. Create an instance of your control or component in the test project. If your component is in the toolbox, you can drag it to your designer surface, or you can create the instance programmatically as shown in the following example:

   ```
   ' Visual Basic
   Dim Component1 As New MyNeatComponent()

   // C#
   MyNeatComponent Component1 = new MyNeatComponent();
   ```

   You can now debug your control or component as normal.

For more information on debugging, see Debugging.

Event Fires Twice in Inherited Control or Component

This is likely due to a duplicated **Handles** clause. For more information, see Troubleshooting Inherited Event Handlers in Visual Basic .NET.

See Also

Authoring Components | Control Authoring for Windows Forms

Component Authoring

The designers in Visual Studio .NET make it easy to author components and you can extend their functionality to provide a better programming experience for users of your components.

> **Visual Basic Note** Visual Basic programmers should be familiar with object-based programming concepts. It is not necessary to have a thorough understanding of inheritance, but you may find it useful to refer to Inheritance.

In This Section

Component Authoring Walkthroughs
Links to topics that provide step-by-step instruction for component programming.
Authoring Components
Gives the general procedure for creating your own component.
Component Architecture
Explains how the .NET Framework supports component programming: Naming recommendations, assemblies, the common language specification, and how garbage collection affects object lifetime.
Component Classes
Describes what makes a class a component, ways to expose component functionality, controlling access to components, and controlling how component instances are created.
Implementing Properties, Methods, Members, and Events in Components
Provides links to topics that explain the implementation of properties and methods in your component.
Interfaces and Abstract Classes
Describes how to implement polymorphic behavior in your components through interfaces and abstract classes.
Multithreading in Components
Discusses how to implement multithreading in your components.
Displaying Forms from Components
Demonstrates how to display a form from a nonvisual component.
Code Libraries
Discusses issues involved with creating and implementing code libraries.
Creating Code Libraries
Demonstrates how to create a code library from multiple projects.
Components and Containers
Describes the relationship between a component and its container, and how this relationship affects the architecture of your applications.
Components That Contain Other Components
Discusses having an object model in your component, and how to implement complex components.
Exceptions in Components
Provides an overview of structured exception handling and its use in component programming.
Code Security in Components
Demonstrates how to implement code security in your components.
User Assistance for Components
Describes how to implement help files for your components.
Control Authoring for Windows Forms
Provides information on how to author user controls and inherited controls for use in windows forms.
Private Communication Between Components in Assemblies
Describes how to allow components to communicate without exposing that communication to a client application.

Related Sections

Troubleshooting Control and Component Authoring
Explains how to fix common problems.
Visual Basic Code Example Topics
Lists quickstart-like code examples that help you perform common tasks in Visual Basic .NET.
What is the Common Language Specification?
Introduces the common language runtime, which is designed to simplify the creation and use of components. An important aspect of this simplification is enhanced interoperability between components written using different programming languages. The common language specification (CLS) makes it possible to create tools and components that are guaranteed to work with multiple languages.
Component Model Namespaces in Visual Studio
Describes the namespaces used in component authoring and the classes contained in those namespaces.

Decision Chart

Provides a graphical, linked guide to decisions about designing your application.

Enhancing Design-Time Support

Provides links to a variety of topics related to creating design-time support for your components, such as implementing UI type editors, type converters, and custom designers.

Design-Time Support for Windows Forms

Provides samples of design-time implementations in Windows Forms, such as UI type editors, extender providers, and designers.

Component Authoring Walkthroughs

The following step-by-step topics will help you learn how to create a component or a control for Windows Forms. For more information about creating custom controls for Web Forms pages, see Developing ASP.NET Server Controls.

In This Section

Authoring a Component with Visual Basic .NET
Shows the development in Visual Basic of a simple component, illustrating the interaction between client and component, object lifetime and circular references, debugging clients and components, and the use of shared methods and instance methods.

Authoring a Component with Visual C#
Shows in C# the development of a simple component, illustrating the interaction between client and component, object lifetime and circular references, debugging clients and components, and the use of shared methods and instance methods.

Related Sections

Authoring a Simple Multithreaded Component with Visual Basic
Demonstrates in Visual Basic the development of a simple multithreaded component, explaining how threads work and how to coordinate multiple threads in your component.

Authoring a Simple Multithreaded Component with Visual C#
Demonstrates in C# the development of a simple multithreaded component, explaining how threads work and how to coordinate multiple threads in your component.

Inheriting from a Windows Forms Control with Visual Basic .NET
Shows in Visual Basic the development of a simple control that inherits from a pre-existing Windows Forms control.

Inheriting From a Windows Forms Control with Visual C#
Shows in C# the development of a simple control that inherits from a pre-existing Windows Forms control.

Visual Basic Code Example Topics
Shows quickstart-like code examples that help you perform common tasks in Visual Basic .NET.

Authoring a User Control with Visual Basic .NET
Shows in Visual Basic the development of a simple control that inherits from the **UserControl** class, and how to inherit from an established user control.

Authoring a User Control with Visual C#
Shows in C# the development of a simple control that inherits from the UserControl class, and how to inherit from an established user control.

Component Programming Walkthroughs
Provides a list of the walkthroughs available for component creation and implementation.

Troubleshooting Control and Component Authoring
Explains how to fix common problems.

Programming with Components
Provides high-level information and links to topics about programming with components, including components that interact with Microsoft Message Queuing system resources, the file system, and Active Directory.

Authoring Components
Describes the process of creating your own custom control for Windows Forms or your own component.

Walkthroughs in Visual Basic and Visual C#
Lists all the step-by-step topics that use Visual Basic and Visual C#.

Walkthrough: Authoring a Component with Visual Basic .NET

Components provide reusable code in the form of objects. An application that uses a component's code, by creating objects and calling their properties and methods, is referred to as a *client*. A client may or may not be in the same assembly as a component it uses.

The following procedures build on each other, so the order in which you perform them is important.

Creating the Project

To create the CDemoLib class library and the CDemo component

1. From the **File** menu, select **New** and then **Project** to open the **New Project** dialog. Select the **Class Library** project template from the list of **Visual Basic Projects**, and enter **CDemoLib** in the **Name** box.

   Note The Class Library template is not available in the Standard Edition of Visual Basic .NET. For more information, see Visual Basic Standard Edition Features.

   Tip Always specify the name of a new project when you create it. Doing so sets the root namespace, assembly name, and project name, and also ensures that the default component will be in the correct namespace.

2. In Solution Explorer, right-click **CDemoLib** and select **Properties** from the shortcut menu. Notice that the **Root Namespace** box contains **CDemoLib**.

   The root namespace is used to qualify the names of components in the assembly. For example, if two assemblies provide components named CDemo, you can specify your CDemo component using `CDemoLib.CDemo`.

   Click **Cancel** to close the dialog box.

3. From the **Project** menu, choose **Add Component**. In the **Add New Item** Dialog Box, select **Component Class** and type **CDemo.vb**. in the **Name** box. A component named CDemo.vb is added to your class library.
4. In the Solution Explorer, right-click **CDemo.vb** and choose View code. The code editor opens.

   Notice the `Inherits System.ComponentModel.Component` immediately below `Public Class CDemo`. This section designates the class from which your class inherits. By default, a component inherits from the **Component** class provided by the system. The **Component** class provides many features for your component, including the ability to use designers.

5. In Solution Explorer, right-click **Class1.vb** and choose **Delete**. This deletes the default class that is provided with the class library, as it will not be used in this walkthrough.
6. From the **File** menu, choose **Save All** to save the project.

Adding Constructors and Finalizers

Constructors control the way your component is initialized; the **Finalize** method controls the way it tears down. Code in the constructor and the **Finalize** method of the CDemo class maintains a running count of the number of CDemo objects in existence.

To add code for the constructor and finalizer of the CDemo class

1. In the Code Editor, add member variables to keep a running total of instances of the CDemo class, and an ID number for each instance.

   ```
   Public ReadOnly InstanceID As Integer
   Private Shared NextInstanceID As Integer = 0
   Private Shared ClassInstanceCount As Long = 0
   ```

   Because the `InstanceCount` and `NextInstanceID` member variables are declared **Shared**, they exist only at the class level. All instances of CDemo that access these members will use the same memory locations. Shared members will be initialized the first time the CDemo class is referred to in code. This could be the first time a CDemo object is created, or the first time one of the class' shared members is accessed.

2. Locate **Public Sub New()** and **Public Sub New(Container As System.ComponentModel.IContainer)**, the default constructors for the CDemo class. In Visual Basic .NET, all constructors are named **New**. Your component can have several constructors, with different parameters, but they must all have the name **New**.

Note The access level of a class' constructors determines what clients will be able to create instances of the class. In other versions of Visual Basic, object creation was controlled by the Instancing property; if you've used the **Instancing** property, you may find it useful to read Component Instancing Changes in Visual Basic.

3. Add the following code to **Sub New()**, to increment the instance count when a new CDemo is created, and to set the instance ID number. You may have to expand the **Component Designer Generated Code** box in the code editor.

   Note Always add your code after the call to **InitializeComponent**. At that point, any constituent components have been initialized.

```
InstanceID = NextInstanceID
NextInstanceID += 1
ClassInstanceCount += 1
```

   As a **ReadOnly** member, `InstanceID` can be set only in the constructor.

   Note Users familiar with multithreading will point out quite rightly that assigning
   InstanceID
   and incrementing
   NextInstanceID
   should be an atomic operation. This and other issues related to threading are illustrated in
   Walkthrough: Authoring a Simple Multithreaded Component with Visual Basic

4. Add the following method after the end of the constructor:

```
Protected Overrides Sub Finalize()
   ClassInstanceCount -= 1
   End Sub
```

   The memory manager calls **Finalize** just before it finally reclaims memory occupied by the CDemo object. The **Finalize** method originates with **Object**, the root of all reference types in the .NET class hierarchy. By overriding **Finalize**, you can perform cleanup just before your component is removed from memory. However, as you'll see later in this walkthrough, there are good reasons to release resources earlier.

Adding a Property to the Class

The CDemo class has only one property, a shared property that allows the client to find out how many CDemo objects there are in memory at any given time. Methods can be created in a similar way.

To create a property for the CDemo class

- Add the following property declaration to the CDemo class, to allow clients to retrieve the number of instances of CDemo.

```
Public Shared ReadOnly Property InstanceCount() As Long
   Get
      Return ClassInstanceCount
   End Get
End Property
```

   Note Property declaration syntax is different from that employed in earlier versions of Visual Basic. For more information of the syntax change, see Property Changes in Visual Basic.

Testing the Component

To test the component, you need a project that uses it. This project must be the first project that starts when you press the Run button.

To add the CDemoTest client project as the startup project for the solution

1. From the **File** menu, point to **Add Project** and choose **New Project** to open the **Add New Project** dialog box.
2. Select the **Windows Application** project template from the list of Visual Basic Projects, and type **CDemoTest** in the **Name** box and then click **OK**.

3. In Solution Explorer, right-click **CDemoTest** and click **Set as Startup Project** on the shortcut menu.

In order to use the CDemo component, the client test project must have a reference to the class library project. After adding the reference, it is a good idea to add an **Imports** statement to the test application to simplify use of the component.

To add a reference to the class library project

1. In Solution Explorer, right-click the **References** node immediately beneath **CDemoTest**, and select **Add Reference** from the shortcut menu.
2. In the **Add Reference** dialog box, select the **Projects** tab.
3. Double-click the **CDemoLib** class library project, to add it to the **Selected Components** list. CDemoLib will appear under the References node for the CDemoTest project. Click **OK** to accept the selection.
4. In Solution Explorer, right-click **Form1.vb** and select **View Code** from the shortcut menu.

Adding the reference to CDemoLib allows you to use the fully qualified name of the CDemo component — that is, CDemoLib.CDemo.

To add an Imports statement

- Add the following **Imports** statement to the top of the Code Editor for Form1, above the Class declaration:

  ```
  Imports CDemoLib
  ```

  Adding the **Imports** statement allows you to omit the library name, and refer to the component type as CDemo. For more information on the **Imports** statement, see Namespaces.

  You will now create and use a test program to test your component.

Understanding Object Lifetime

The CDemoTest program will illustrate object lifetime in the .NET Framework by creating and releasing large numbers of CDemo objects.

To add code to create and release CDemo objects

1. Click **Form1.vb[Design]** to return to the designer.
2. Drag a **Button** and a **Timer** from the **Windows Forms** tab of the Toolbox onto the Form1 design surface.

   The nonvisual **Timer** component appears on a separate design surface below the form.

3. Double-click the icon for **Timer1** to create an event-handling method for the Timer1 component's **Tick** event. Place the following code in the event-handling method.

   ```
   Me.Text = "CDemo instances: " & CDemo.InstanceCount
   ```

   On every tick of the timer, the form's caption will display the current instance count for the CDemo class. The class name is used as a qualifier for the shared InstanceCount property — there is no need to create an instance of CDemo to access a shared member.

4. Locate the constructor for Form1 (**Sub New**), and add the following code after the call to **InitializeComponent**.

   ```
   Timer1.Enabled = True
   ```

   This will start the timer as soon as the form is created.

5. Click the **Form1.vb [Design]** tab to return to the designer.
6. Double-click the **Button** on Form1, to create an event-handling method for the button's **Click** event. Place the following code in the event-handling method.

   ```
   Dim cd As CDemo
   Dim ct As Integer
   For ct = 1 To 1000
      cd = New CDemo
   Next
   ```

This code may look strange to you. As each instance of CDemo is created, the previous instance is released. When the **For** loop is done, there will be only one instance of CDemo left. When the event-handling method exits, even that instance will be released, because the variable `cd` will go out of scope.

As you may have guessed already, things won't happen quite that way.

To run and debug the CDemoTest and CDemo projects

1. Press **F5** to start the solution.

   The client project will start, and Form1 will be displayed. Notice that the caption of the form displays "CDemo instances: 0".

2. Click the button. The caption of the form should display "CDemo instances: 1000".

   The instances of CDemo were all released by the time the button's **Click** event-handling procedure finished. Why haven't they been finalized? In brief, the memory manager finalizes objects in the background, at low priority. The priority is only bumped up if the system gets low on memory. This "lazy" garbage collection scheme allows for very fast object allocation.

3. Click the button several more times, watching the caption. At some point, the number of instances will suddenly drop. This means that the memory manager has reclaimed the memory of some of the objects.

   > **Note** If you've clicked more than 10 times, and the number of CDemo instances has not decreased, you may need to adjust the code so that it uses more memory. Close the form to return to the development environment, and increase the number of iterations in the **For** loop to 10000. Then run the project again.

4. Repeat step 3. You will get farther this time before the memory manager finalizes more objects.

   In fact, each time you repeat step 3, you will probably be able to allocate more CDemo objects before the memory manager steps in. This is because more and more of Visual Studio is swapped out, leaving more space for instances of CDemo.

5. Close the form to return to the development environment.

See Also

Programming with Components | Component Authoring Walkthroughs

Walkthrough: Authoring a Component with Visual C#

Components provide reusable code in the form of objects. An application that uses a component's code, by creating objects and calling their properties and methods, is referred to as a *client*. A client may or may not be in the same assembly as a component it uses.

The following procedures build on each other, so the order in which you perform them is important.

Creating the Project

To create the CDemoLib class library and the CDemo component

1. From the **File** menu, select **New** and then **Project** to open the **New Project** dialog. Select the **Class Library** project template from the list of **C# Projects**, and enter **CDemoLib** in the **Name** box.

   Tip Always specify the name of a new project when you create it. Doing so sets the root namespace, assembly name, and project name, and also ensures that the default component will be in the correct namespace.

2. In Solution Explorer, right-click **CDemoLib** and select **Properties** from the shortcut menu. Notice that the **Root Namespace** box contains **CDemoLib**.

   The root namespace is used to qualify the names of components in the assembly. For example, if two assemblies provide components named CDemo, you can specify your CDemo component using `CDemoLib.CDemo`.

   Click **Cancel** to close the dialog box.

3. From the **Project** menu, choose **Add Component**.
4. In the **Add New Item** dialog box, select **Component Class** and type **CDemo.cs**. in the **Name** box.

   A component named CDemo.cs is added to your class library.

5. In Solution Explorer, right-click **CDemo.cs** and choose View code. The code editor opens.

   Notice the `: System.ComponentModel.Component` immediately after `public class CDemo`. This section designates the class from which your class inherits. By default, a component inherits from the **Component** class provided by the system. The **Component** class provides many features for your component, including the ability to use designers.

6. In Solution Explorer, right-click **Class1.cs** and choose **Delete**. This deletes the default class that is provided with the class library, as it will not be used in this walkthrough.
7. From the **File** menu, choose **Save All** to save the project.

Adding Constructors and Destructors

Constructors control the way your component is initialized; the **Finalize** method controls the way it tears down. Code in the constructor and the **Finalize** method of the CDemo class maintains a running count of the number of CDemo objects in existence.

To add code for the constructor and destructor of the CDemo class

1. In the Code Editor, add member variables to keep a running total of instances of the CDemo class, and an ID number for each instance.

   ```
   public readonly int InstanceID;
   private static int NextInstanceID = 0;
   private static long ClassInstanceCount = 0;
   ```

   Because the `InstanceCount` and `NextInstanceID` member variables are declared **static**, they exist only at the class level. All instances of CDemo that access these members will use the same memory locations. Shared members will be initialized the first time the CDemo class is referred to in code. This could be the first time a CDemo object is created, or the first time one of the class' shared members is accessed.

2. Locate **public CDemo()** and **public CDemo(System.ComponentModel.IContainer container)**, the default constructors for the CDemo class. In Visual C# .NET, all constructors have the same name as the class. Your component can have several constructors, with different parameters, but they must all have the same name as your component.

   Note The access level of a class' constructors determines what clients will be able to create instances of the

class.

3. Add the following code to `public CDemo()`, to increment the instance count when a new CDemo is created, and to set the instance ID number. You may have to expand the **Component Designer Generated Code** box in the Code Editor.

   Note Always add your code after the call to **InitializeComponent**. At that point, any constituent components have been initialized.

```
InstanceID = NextInstanceID ++;
ClassInstanceCount ++;
```

   As a `readonly` member, `InstanceID` can be set only in the constructor.

   Note Users familiar with multithreading will point out quite rightly that assigning
   InstanceID
   and incrementing
   NextInstanceID
   should be an atomic operation. This and other issues related to threading are illustrated in Walkthrough: Authoring a Simple Multithreaded Component with Visual C#.

4. Add the following method after the end of the constructor:

```
~CDemo()
{
   ClassInstanceCount --;
}
```

   This method is called a *destructor*, and is signified by the tilde character (~) in front of the class name. The memory manager calls the destructor just before it finally reclaims memory occupied by the CDemo object. By implementing a destructor you can perform cleanup just before your component is removed from memory. However, as you'll see later in this walkthrough, there are good reasons to release resources earlier.

Adding a Property to the Class

The CDemo class has only one property, a shared property that allows the client to find out how many CDemo objects there are in memory at any given time. Methods can be created in a similar way.

To create a property for the CDemo class

- Add the following property declaration to the CDemo class, to allow clients to retrieve the number of instances of CDemo.

```
public static long InstanceCount
   {
      get
      {
         return ClassInstanceCount;
      }
   }
```

Testing the Component

To test the component, you need a project that uses it. This project must be the first project that starts when you press the Run button.

To add the CDemoTest client project as the startup project for the solution

1. From the **File** menu, point to **Add Project** and choose **New Project** to open the **Add New Project** dialog box.
2. Select the **Windows Application** project template from the list of Visual C# Projects, and type **CDemoTest** in the **Name** box and then click **OK**.
3. In Solution Explorer, right-click **CDemoTest** and click **Set as Startup Project** from the shortcut menu.

In order to use the CDemo component, the client test project must have a reference to the class library project. After adding the

reference, it is a good idea to add a **using** statement to the test application to simplify use of the component.

To add a reference to the class library project

1. In Solution Explorer, right-click the **References** node immediately beneath **CDemoTest**, and select **Add Reference** from the shortcut menu.
2. In the **Add Reference** dialog box, select the **Projects** tab.
3. Double-click the **CDemoLib** class library project, to add it to the **Selected Components** list. CDemoLib will appear under the References node for the CDemoTest project. Click **OK** to accept the selection.
4. In Solution Explorer, right-click **Form1.cs** and select **View Code** from the shortcut menu.

Adding the reference to CDemoLib allows you to use the fully qualified name of the CDemo component — that is, CDemoLib.CDemo.

To add a using statement

- Add the following **using** statement to the list of **using** statements at the top of the Code Editor for Form1.

  ```
  using CDemoLib;
  ```

  Adding the **using** statement allows you to omit the library name, and refer to the component type as CDemo.

  You will now create and use a test program to test your component.

Understanding Object Lifetime

The CDemoTest program will illustrate object lifetime in the .NET Framework by creating and releasing large numbers of CDemo objects.

To add code to create and release CDemo objects

1. Click **Form1.cs[Design]** to return to the designer.
2. Drag a **Button** and a **Timer** from the **Windows Forms** tab of the Toolbox onto the Form1 design surface.

   The nonvisual **Timer** component appears on a separate design surface below the form.

3. Double-click the icon for **timer1** to create an event-handling method for the timer1 component's **Tick** event. Place the following code in the event-handling method.

   ```
   this.Text = "CDemo instances: " + CDemo.InstanceCount;
   ```

   On every tick of the timer, the form's caption will display the current instance count for the CDemo class. The class name is used as a qualifier for the shared InstanceCount property — there is no need to create an instance of CDemo to access a shared member.

4. Locate the constructor for Form1 (**public Form1()**), and add the following code after the call to **InitializeComponent()**.

   ```
   timer1.Enabled = true;
   ```

   This will start the timer as soon as the form is created.

5. Click the **Form1.cs [Design]** tab to return to the designer.
6. Double-click the **Button** on Form1, to create an event-handling method for the button's **Click** event. Place the following code in the event-handling method.

   ```
   CDemo cd;
   int ct;
   for (ct = 0; ct < 1000; ct++)
      cd = new CDemo();
   ```

   This code may look strange to you. As each instance of CDemo is created, the previous instance is released. When the **for** loop is done, there will be only one instance of CDemo left. When the event-handling method exits, even that instance will be released, because the variable `cd` will go out of scope.

As you may have guessed already, things won't happen quite that way.

To run and debug the CDemoTest and CDemo projects

1. Press **F5** to start the solution.

   The client project will start, and Form1 will be displayed. Notice that the caption of the form displays "CDemo instances: 0".

2. Click the button. The caption of the form should display "CDemo instances: 1000".

   The instances of CDemo were all released by the time the button's **Click** event-handling procedure finished. Why haven't they been finalized? In brief, the memory manager finalizes objects in the background, at low priority. The priority is only bumped up if the system gets low on memory. This "lazy" garbage collection scheme allows for very fast object allocation.

3. Click the button several more times, watching the caption. At some point, the number of instances will suddenly drop. This means that the memory manager has reclaimed the memory of some of the objects.

   > **Note** If you've clicked more than 10 times, and the number of CDemo instances has not decreased, you may need to adjust the code so that it uses more memory. Close the form to return to the development environment, and increase the number of iterations in the **for** loop to 10000. Then run the project again.

4. Repeat Step 3. You will get farther this time before the memory manager finalizes more objects.

   In fact, each time you repeat step 3, you will probably be able to allocate more CDemo objects before the memory manager steps in. This is because more and more of Visual Studio is swapped out, leaving more space for instances of CDemo.

5. Close the form to return to the development environment.

See Also

Programming with Components | Component Authoring Walkthroughs

Authoring Components

There are as many ways to approach a programming problem as there are programmers, and creating your own components is just another programming task. With that in mind, the following steps represent a fairly comprehensive approach.

To author a component

1. Determine what you want your component to accomplish, or what part it will play in your application.
2. If you have a more complex component and need an object model, outline the model. For more information, see Component Object-Model Recommendations
3. If necessary, divide up functionality between the component and any subobjects or structures in its object model. For more information, see Nested Classes in Components.
4. Determine the best class or component to use as a base class for your component. Generally speaking, your base class will already implement many of the properties and methods that your class will need.
5. If you have subobjects, determine the best class or component to use as a base class for them.
6. Determine what functionality your component can provide by incorporating components from the .NET Framework classes.
7. If you need to factor out subsets of your component's functionality as interfaces, determine which classes need to implement them.
8. Express functionality as properties, methods, and events of the component and its subobjects or subsidiary structures. For more information, see Implementing Properties, Methods, Members, and Events in Components.
9. Assign appropriate access levels (**Public**, **Private**, and so on). For more information, see Ways to Implement Component Functionality.
10. Test and debug your component. As you add each feature, add features to your test project to exercise the new functionality.
11. Repeat, refining the design.

See Also

Walkthrough: Authoring a Component with Visual Basic .NET | Walkthrough: Authoring a Component with Visual C# | Troubleshooting Control and Component Authoring

Component Architecture

The topics in this section explain the overall architecture of components for the common language runtime. The .NET Framework is a library of components, plus supporting classes and structures, designed to make component development easy and robust. Components are defined more strictly than in COM: A component either implements the **System.ComponentModel.IComponent** interface, or has the **System.ComponentModel.Component** class somewhere in its inheritance tree.

The common language runtime is a requirement for any application developed with the framework. For information on the organization of classes in the libraries, see Introduction to the .NET Framework Class Library.

In This Section

Ways to Implement Component Functionality
  Explains the differences between standard modules, structures, and classes.
Components and Assemblies
  Explains how components are packaged into executables.
Reference Counting, Garbage Collection, and Object Lifetime
  Describes the advantages of traced garbage collection.
Component Naming Recommendations
  Provides recommendations for naming your components.

Related Sections

Walkthrough: Authoring a Component with Visual Basic .NET
  Provides step-by-step instructions for creating your own component.
Component Classes
  Explains component design and usability.
System.ComponentModel Namespace
  Provides reference on the **System.ComponentModel** namespace and its members.
Inside the .NET Framework
  Lists links to topics about the common language runtime, managed execution, the Common Type System, cross-language interoperability, and .NET Framework security.

Ways to Implement Component Functionality

You will frequently find it useful to create utility functions and small "helper" objects for your components. You can implement such utility objects and functions using classes and structures. Additionally, in Visual Basic you can implement modules.

> **Visual Basic Note** In previous versions of Visual Basic, classes were the only way to expose component functionality. Standard modules were useful receptacles for internal data and utility functions, but could not be exposed outside the project. User-defined types could be exposed, but had no functionality other than providing complex data types. These restrictions have been removed.

Classes have the advantage of being reference types — passing a reference is more efficient than passing a structure variable with all its data. On the other hand, structures do not require allocation of memory on the global heap.

The following table summarizes the differences between the ways of implementing component functionality. In C# the access modifiers (**Public**, for example) are lowercase (**public**).

	Class	**Structure (struct)**	**Module (Visual Basic only)**
Default access level for the type	**Friend/internal**	**Friend/internal**	**Friend**
Default access level for data members	**Private**	**Public** (Visual Basic .NET) **private** (Visual C#)	**Private**
Default access level for properties and methods	**Public** (Visual Basic) **private** (Visual C#)	**Public** (Visual Basic .NET) **Private** (Visual C#)	**Public**
Value or reference type?	Reference	Value	Neither
Can be a base for new types?	Yes	No	No
Can implement interfaces?	Yes	Yes	No
Can raise and handle events?	Yes	Yes	Yes
Scope of members	Class	Structure	Module and containing Namespace
Instance creation?	**New** operator (Visual Basic) constructor (C#)	**New** operator (Visual Basic) constructor (C#)	Not instanced
Type initialization	Shared constructor, shared initializers	Shared constructor, shared initializers	Shared initializer
Instance initialization	Parameterized constructor, parameterless constructor	Parameterized constructor, parameterless constructor	Not instanced
Can contain nested types?	Yes	Yes	Yes
Can be a nested type?	Yes	Yes	No
Destructor?	Yes	No	No

Structures

Structures are value types — that is, a variable of a structure type contains the structure's data, rather than containing a reference to the data as a class type does. Structures can have properties and methods, can raise and handle events, and can implement interfaces.

> **Visual Basic Note** Previous versions of Visual Basic provided a **Type** statement for defining structures. The **Structure** statement adds many new capabilities. For details, see Structure Statement.

- A structure has an implicit parameterless constructor, which initializes each data member of the structure to the default value for its type.

  > **Note** This means that you cannot declare your own parameterless constructor for a structure.

- In addition to its defined members, a structure includes the members of **System.ValueType**.
- Because you cannot inherit from a structure, **Protected** access is prohibited on structure members.
- Unlike a class, a structure does not have a finalizer (**Sub Finalize** in Visual Basic).
- You can use **Public Shared** (**public static** in C#) members to provide utility functions related to a type. **Shared** (**static**)

members can be called using the type name; no instance of the type is required.

> **Visual Basic Note** Data members in structures must be declared with the **Dim** statement, or with **Public**, **Private**, or **Friend**. This is a change from the user-defined types in earlier versions of Visual Basic.

Although a structure can have properties, methods, and shared data, these do not affect the space occupied by a variable of the structure type. The space allocated for a structure is only as large as the structure's instance data.

When Should You Use a Structure Instead of a Class?

Because you cannot inherit from a structure, structures should be used only for objects that do not need to be extended. Use structures when the object you wish to create has a small instance size, and take into account the performance characteristics of classes versus structures. In general, objects requiring a larger instance size should be created as a class. Structures with large instance sizes cause degradation in performance as the instance data is passed from method to method. Depending on how the structure is allocated and used, however, a structure may be more efficient even with a larger instance size. For example, a structure allocated and passed around inside an array can be efficient.

Classes

In addition to providing all the capabilities of structures, classes can be extended and they have finalizers.

Classes Can Be Extended

By default, your component can be a base class for new components. If you do not want to invest the additional development and testing time to ensure that your component can function as a robust base class, you can mark your class **NotInheritable** (Visual Basic) or **sealed** (C#) to prevent users from extending its functionality.

You can also author a component whose sole purpose is to be a base class for other components, by marking it **MustInherit** (Visual Basic) or **abstract** (C#). Users of your assembly cannot create instances of an **abstract** (**MustInherit)** class; they must derive new component classes from it. For details, see Abstract Classes.

> **Note** When you include an abstract component in your assembly, or any component you intend users to inherit from, you should provide documentation for your component.

Destructors

Before garbage collection frees the memory used by an object, the class finalizer is called. This gives you the opportunity to do any necessary cleanup. However, because of the indeterminate nature of garbage collection, you have no way of knowing when your finalizer will be called. If you want to be able to control when your component terminates, you should implement a **Dispose** method for it. For details, see Initialization and Termination of Components.

Importing Shared Class Members (Visual Basic Only)

Users can import the **Public Shared** members of a class so that they can be accessed without qualification. The following example makes the public shared members of the Mortgage class available without qualification.

```
' Visual Basic
Imports MyCorp.Financials.Mortgage
```

Modules (Visual Basic Only)

Modules provide a simple way to organize utility functions and global data used by the components in your assembly.

- A member of a module can be accessed without qualification within the namespace that contains the definition of the module and within any file that has imported that namespace. If members with duplicate names exist in two modules, you can qualify them with the module names to resolve the ambiguity.
- A module must be a member of a namespace. It cannot be a member of another type, such as a class.
- Modules are not instanced and are functionally equivalent to classes containing only shared members.
- A module cannot have an instance constructor.

If a module is declared **Public**, its public members are visible outside your assembly. In order to access these members, a user of your assembly must qualify them with the name of the module.

See Also

Walkthrough: Authoring a Component with Visual Basic .NET | Component Architecture

Components and Assemblies

Components are packaged in *assemblies*. Assemblies are the reusable, versionable, self-describing building blocks of .NET applications. The simplest assembly is a single executable that contains all the information necessary for deployment and versioning.

The metadata describing an assembly is called the *manifest*. In the most general case, an assembly can consist of a number of executable files and resource files. In that case, the manifest may be a separate file, or it may be included in one of the executable files. Assemblies enable the common language runtime to locate and securely execute code. In particular, the runtime uses assemblies for the following purposes:

- Establishing security and version identity for code.
- Providing a scope for class and type resolution.
- Locating classes to load.

Assemblies and Versioning

One of the primary goals of assemblies is versioning. Specifically, assemblies provide a means for developers to specify version rules between different software components, and to have those rules enforced at run time.

Because assemblies are the building blocks of applications, they are the logical point for specifying and enforcing version information. Each assembly has a specific version number as part of its identity.

The version number is stored in the assembly's manifest along with other identity information, including the friendly name and supported locales. The fact that a version number is part of an assembly's identity is the key to versioning: AssemblyA, version 1 and AssemblyA, version 2 are two completely different entities, and are treated as such at run time.

For more general information on assemblies, see Assemblies in the .NET Framework documentation.

See Also

Component Architecture | Assemblies

Reference Counting, Garbage Collection, and Object Lifetime

Unlike COM, the common language runtime does not use reference counting to govern object lifetime. Instead, the garbage collector traces object references and identifies objects that can no longer be accessed by running code.

This simplifies component programming a great deal, because you do not have to worry about circular references. If a group of objects contain references to each other, but none of these object are referenced directly or indirectly from stack or shared variables, then garbage collection will automatically reclaim the memory.

An additional advantage of traced garbage collection is that allocation of new objects in memory is extremely fast. Performance is further improved by the elimination of the COM **AddRef** and **Release** mechanism, and objects require less memory.

The only potential disadvantage of traced garbage collection is the interval between the release of the last reference that can be reached by running code, and the moment when the garbage collector detects this condition. On a lightly loaded system with a great deal of memory, considerable time may elapse before your component's destructor is called.

The technique used in the .NET Framework is to implement a **Dispose** method on any component that uses limited or expensive system resources. When you are done using an instance of your component, you can call **Dispose** to free all resources except the memory the component is occupying, and to disable the component. For details, see Initialization and Termination of Components.

See Also

Component Architecture | Component.Dispose Method | IDisposable Interface

Component Naming Recommendations

The names you select for your classes — and for their properties, methods, and events — are one of the most important factors in how easy it is to use your component. When naming these elements and their formal parameters, you can help the user of your component by following a few simple rules.

- Use complete words whenever possible — for example, SpellCheck. Abbreviations can take many forms, and hence can be confusing. If you plan to localize your component for other cultures, be aware that abbreviations can be hard for a non-native speaker to understand. If whole words are too long, use complete first syllables or carefully chosen abbreviations.
- Use mixed case for your class, method, and property names, capitalizing each word or syllable — for example, ShortcutMenus, or AsyncReadComplete.
- Use the same word or phrase your users would use to describe a concept. For example, you might have a SavingsAccount component that represented a savings account.
- Append "Collection" onto the correct name of the object contained by your collection — for example WorksheetCollection, FormCollection, or WidgetCollection.
- Use either the verb/object or object/verb order consistently for your method names. For example, placing the verb first results in names like InsertWidget and InsertSprocket, whereas placing the object first yields names like WidgetInsert and SprocketInsert.
- Do not repeat the name of the class in the method. For example, if you have a class named Book, do not have a method called Book.CloseBook. Instead, name your method Book.Close.

See Also

Coding Techniques

Component Classes

A class is a unit of reusable code. So is a component. To author a component, you start with a class. What is the difference, then?

A class becomes a component when it conforms to a standard for component interaction. This standard is provided through the **IComponent** interface. Any class that implements the **IComponent** interface is a component. For details about interfaces, see Polymorphism in Components. Such standards allow developers to plug components together quickly to create larger programs. The common language specification (CLS) describes the way components authored for the common language runtime can interact. For details, see What Is the Common Language Specification?

The .NET Framework provides the **IComponent** interface and the **Component** base class to make it easy to author components that work well in the design environment, and **UserControl** and **Control** classes that make it easy to author visual components.

In This Section

Component Class Characteristics
Introduces class attributes and what they mean for a component class.
Initialization and Termination of Components
Explains where to write the code to initialize your component, dispose of system resources, and destroy your component.
Component Instancing and Constructors
Describes controlling the way your components are created, by setting the appropriate access on their constructors.
Component Instancing Changes in Visual Basic
Lists class and constructor access levels for the Visual Basic 6.0 programmer.

Related Sections

Walkthrough: Authoring a Component with Visual Basic .NET
Provides step-by-step instructions for creating your own component.
Ways to Implement Component Functionality
Recommends when users should use a class, a structure, or (for Visual Basic users) a module to create a utility function or "helper" object.
Authoring Components
Describes the steps for creating your own component.
Component Authoring
Lists topics about creating your own components.
Implementing Properties, Methods, Members, and Events in Components
Details the use of properties, methods, and member variables in your component.

Component Class Characteristics

The characteristics of a class acquire additional significance when the class becomes a component.

Component Name

Choose component class names carefully. A class name should be short but descriptive, formed from whole words, with individual words capitalized — for example, BusinessRule. This provides a convenient way to organize your components and to produce shorthand clues to component functions. For more information, see Component Naming Recommendations.

Access Modifier

A class defined with private access is not visible to users of your assembly. This is the access level to use for helper classes internal to the assembly.

Setting the **Access** modifier to public makes a component class available to users of your assembly. To control whether users of your assembly can create instances of the component, specify the appropriate access level for its constructor. For details, see Component Instancing and Constructors.

Base Class

Unless you intend to implement the **IComponent** interface yourself, your base class should be **System.ComponentModel.Component** or a class that derives from **System.ComponentModel.Component**. You can find this information in the reference topic for any class in the .NET Framework. In Visual Basic, the **Inherits** statement specifies the base class. In C#, the base class follows the colon in the class declaration. Examples are shown below:

```
' Visual Basic
Imports System.ComponentModel
Public Class MyComponent
   Inherits System.ComponentModel.Component
End Class

//C#
using System.ComponentModel
public class MyComponent:System.ComponentModel.Component;
{}
```

Namespace Statement

Every component is contained within a *namespace*. By default, the namespace is the name of your project. Users of your component assembly will add **Imports** (Visual Basic) or **using** (C#) statements for the namespaces containing components they want to access.

> **Note** You can add additional levels of structure by enclosing components within additional **Namespace...End Namespace** blocks.

Generally speaking, the structure of namespaces in your component assembly should reflect its internal organization. If your assembly contains a large number of components, it makes sense to group related components in separate namespaces.

See Also

Walkthrough: Authoring a Component with Visual Basic .NET | Initialization and Termination of Components | Component Instancing and Constructors | Component Instancing Changes in Visual Basic | Components and Assemblies

Initialization and Termination of Components

Your component is initialized by its constructor (**Sub New** in Visual Basic) and destroyed by its destructor (**Sub Finalize** in Visual Basic). Your component's constructor is called when an instance of your component is created; the constructor cannot be called thereafter. The destructor is called just before your component is destroyed by garbage collection and its memory is reclaimed.

> **Visual Basic Note** In previous versions of Visual Basic, the **Initialize** and **Terminate** events served the same purpose as the constructor and destructor.

Waiting for Garbage Collection

The common language runtime calls your component's destructor after the garbage collection determines that the component can no longer be reached by any executing code. This happens if all references to the component have been released, or if the only references to your component are held by objects that are similarly isolated from all executing code — for example, in the case of circular references.

Because there can be a delay between the moment when a user is finished with your component and the moment when its destructor is called, an additional step is introduced into the lifetime of .NET components: If your component acquires system resources, such as database connections or handles to Windows system objects, you should implement the **IDisposable** interface, and provide a **Dispose** method so that users of your component can choose when to release those resources.

Life Cycle of a Component

1. **Type initialization**: When the first instance of your component is created, the first code that executes is any shared initialization code. A reference to any shared member will also cause the shared constructor to execute. This includes any shared fields (member variables) that are initialized, and the shared constructor (**Shared Sub New**) if it exists. In the following code, a reference font is created for the entire class.

   > **Note** The C# keyword that corresponds to **Shared** is **static**, which is not to be confused with the **Static** keyword in Visual Basic.

   ```
   ' Visual Basic
   Public Class ADrawing
   Shared ReadOnly ReferenceFont As New Font("TimesNewRoman", 14)
   ' Shared constructor does not overload other constructors.
   Shared Sub New()
      ' There is no call to the base class's shared constructor.
      ' Insert code to initialize the shared data.
   End Sub
   End Class

   // C#
   class ADrawing
   {
      static Font m_referenceFont = new Font("TimesNewRoman", 14);
      // Simply add the static keyword to create a static constructor.
      // Static constructors do not have any parameters.
      static ADrawing()
      {
         // There is no call to the base class's static constructor.
         // Code to initialize the font here.
      }
   }
   ```

   > **Note** Class initialization can occur even if no instances of your component are created. For example, an **abstract** (**MustInherit**) class with shared member functions will be initialized and those functions will be available for use by the application, even though no instance of the class is created.

2. **Instance initialization**: When an instance of your component is created, data members that have initialization code are initialized, and the appropriate constructor overload is executed. The following code initializes a private field and defines two

constructors, one to be called if there are no parameters, and the other to be called if the user specifies parameters.

```
' Visual Basic
Class AShape
   Private answer As Integer = 42
   Public Sub New()
      ' Call another constructor with default initialization values.
      MyClass.New(System.Drawing.Color.Red, 5280, DefinedSizes.Large)
   End Sub
   Public Overloads Sub New(myColor As Color, myLength As Integer, _
            Size As DefinedSizes)
      ' Insert code to initialize the class.
   End Sub
   ' Defines the DefinedSizes enum
   Public Enum DefinedSizes
      Large
      Small
   End Enum
End Class

// C#
class AShape
{
   private int m_answer = 42;
   // Forward to another constructor.
   public AShape()
   : this(System.Drawing.Color.Red, 5280, DefinedSizes.Large)
   {
      // Additional initialization goes here.
   }

   public AShape(Color color, int length, DefinedSizes size)
   {
      // Code to initialize the class goes here.
   }
   // Defines the DefinedSizes enum
   public enum DefinedSizes
   {
      Large,
      Small
   }
}
```

3. **Disposing of resources**: If your component implements the **IDisposable** interface, it must provide a **Dispose** method, which the client should call when it is finished using the component. Note that any component that inherits from **System.ComponentModel.Component** already has a default implementation of **Dispose**, which can be overridden to provide additional cleanup code. In the **Dispose** method, your component frees all system resources it may have allocated, releases references to other objects, and renders itself unusable. There may also be instances where it is appropriate for your component to call its own **Dispose** method. The following code disposes of a dependent object that has a **Dispose** method.

```
' Visual Basic
' Assumes that the class implements IDisposable
Public Sub Dispose() Implements IDisposable.Dispose
   myWidget.Dispose
   myWidget = Nothing
   ' Insert additional code.
End Sub
```

```
// C#
// Assumes that the class implements IDisposable
public void IDisposable.Dispose()
{
   mywidget.Dispose();
   mywidget = null;
   // Dispose of remaining objects.
}
```

After you call **Dispose**, your client should release any remaining references to the component, so that garbage collection can reclaim the component's memory.

4. **Instance destruction**: When garbage collection detects that there are no remaining references to the component, the runtime calls your component's destructor (**Finalize** in Visual Basic .NET) and frees the memory. You should override your base class's **Finalize** method (for Visual Basic .NET) or implement a destructor (for Visual C#) to implement your own clean-up code, but always include a call to the destructor or **Finalize** method of the base class.

```
' Visual Basic
Protected Overrides Sub Finalize()
   m_Gadget = Nothing
   m_Gear = Nothing
   MyBase.Finalize()
End Sub

// C#
// In C#, a destructor is used instead of a Finalize method.
~ThisClass()
{
   m_gadget = null;
   m_gear = null;
   // The base class finalizer is called automatically
}
```

When Should You Implement a Dispose Method?

If your component inherits from **System.ComponentModel.Component**, a default implementation of **Dispose** is provided. This implementation can be overridden to provide custom cleanup code. If you are building your component by creating a custom implementation of **IComponent**, you should implement **IDisposable** to provide a **Dispose** method for your component.

Your component needs a **Dispose** method if it allocates system objects, database connections, or other scarce resources that should be released as soon as a user is finished with the component.

You should also implement a **Dispose** method if your component holds references to other objects that have **Dispose** methods.

Why Implement Dispose?

Depending on system activity, an unpredictable interval could elapse between the time a user finishes using your component and the time garbage collection detects that the component's code is unreachable. If you do not provide a **Dispose** method, your component will continue to hold its resources during this interval.

A Worst-Case Scenario

Imagine a server component that uses a database connection and does not have a **Dispose** method. On a server with a large amount of memory, you might create and release many instances of the component without having much impact on free memory. In this case, garbage collection might not destroy the components for some time after the references to them are released.

Eventually, all of the available database connections could be tied up by components that had been released but not destroyed. Even though the server had no shortage of memory, it might be unable to respond to user requests.

See Also

Walkthrough: Authoring a Component with Visual Basic .NET | Component Class Characteristics | Component Instancing and Constructors | Component Instancing Changes in Visual Basic | Component.Dispose Method | Object.Finalize Method | Polymorphism in Components

Component Instancing and Constructors

You can control the way programmers create instances of your component classes. For example, you may want to ensure that every **Book** object is added to the **Library** collection as soon as the user creates it. Or you may want to limit a component to a single instance.

By giving different access levels to your component's constructors (overloads of **Sub New** in Visual Basic), you can control the ways your component is created.

> **Note** If you have used Visual Basic 5.0 or 6.0, you may find it helpful to read Component Instancing Changes in Visual Basic.

Public Constructors

Make a constructor public (**Public Sub New** or **Overloads Public Sub New** in Visual Basic) so that developers can use it to create instances of your component. If you have multiple public constructors, the one that will be invoked is the one that matches the argument list the user specifies.

Friend (Internal) Constructors

If you have a constructor you want to use within your assembly, but do not want to expose to users of your component, use the access level **Friend** (**Friend Sub New** or **Overloads Friend Sub New**) in Visual Basic, or **internal** in C#.

For example, you might want a component to appear only as a member of a collection. If you give its only constructor **Friend** access, users cannot create instances themselves. However, the **Add** method of your collection can use the **New** operator to create an instance of the component, because the component and collection are part of the same assembly. The following sample code illustrates this.

```
' Visual Basic
' Constructor for the Book class.
Class Book
   Friend Sub New(ByVal Ident as Integer, Author as String, Publisher as _
      String)
      ' Insert code to initialize Book object.
   End Sub
End Class
Class Library
   ' Add method for the Books collection class of the Library class.
   Public Function Add(Ident, Author, Publisher) As Book
      Dim b As New Book(Ident, Author, Publisher)
      ' Insert code to insert b into the collection.
      ' Return a reference to the new book.
      Return b
   End Function
End Class

// C#
class Book
{
   // Book class constructor.
   internal Book(int ident, string author, string publisher)
   {
      // Add code to inititialize.
   }
}

class Library
{
   // Add method for the Books collection of the Library class.
   public Book Add(int ident, string author, string publisher)
   {
      Book newBook = new Book(ident, author, publisher);
      // Add code to add newBook to the Books collection.
      // Return a reference to the new book.
      return(newBook);
```

```
    }
}
```

Users of your assembly cannot create the component with the **New** operator because there is no public constructor.

Private Constructors

Make a constructor private if:

- You want it to be available only to the class itself. For example, you might have a special constructor used only in the implementation of your class' **Clone** method.
- You do not want instances of your component to be created. For example, you may have a class containing nothing but **Shared** utility functions, and no instance data. Creating instances of the class would waste memory. In Visual Basic, you would use a module for this kind of construct.

  > **Note** The C# keyword that corresponds to **Shared** is **static**, which is not to be confused with **Static** keyword in Visual Basic.

- You want to limit your component to one instance.

See Also

Component Instancing Changes in Visual Basic | Walkthrough: Authoring a Component with Visual Basic .NET | Component Class Characteristics | Initialization and Termination of Components

Component Instancing Changes in Visual Basic .NET

In Visual Basic 6.0, instance creation was controlled by the **Instancing** property of a class, which set both the access level (**Public**, **Private**, **Friend**) of a class and the way an object could be created.

In Visual Basic .NET, you set the access modifier for a class to **Public**, **Private**, or **Friend**, and then specify the user's ability to create instances of the class by setting the access level of the constructor (**Sub New**), as shown in the following table.

Visual Basic 6.0 Instancing	Visual Basic .NET
Private	Class **Access** attribute: **Private**
PublicNotCreatable	Class **Access** attribute: **Public** Declare the constructor **Friend** (**Friend Sub New**)
SingleUse	Not supported*
GlobalSingleUse	Not supported*
MultiUse	Class **Access** attribute: **Public** Declare the constructor **Public** (**Public Sub New**)
GlobalMultiUse	Not supported*

* There is no combination of class access and constructor access that duplicates **SingleUse**, **GlobalSingleUse**, or **GlobalMultiUse**. You can get an effect similar to **GlobalMultiUse** by importing a class into a client program, although this only makes shared members accessible without qualification. Public modules also allow a similar effect.

See Also

Component Instancing and Constructors | Component Class Characteristics | Initialization and Termination of Components

Implementing Properties, Methods, Members, and Events in Components

Your component provides functionality through its properties, methods, and events. Methods are the actions a user can tell your component to execute. Properties and member variables are data a user of your component can get and/or set. Events are notifications that something interesting has happened to your component.

The following topics examine the issues you should be aware of when declaring properties, methods, events, and other members in a component.

In This Section

Method Implementation in Components
Provides conceptual information on implementing methods in your component and overloading methods.
Event Implementation for Components and Custom Controls with Visual Basic
Introduces declaring and raising events.
Troubleshooting Inherited Event Handlers in Visual Basic .NET
Describes how to correct common problems that arise with event handlers in inherited components.
Named Constants for Your Components
Discusses enumerations, read-only constants, and reference-value constants.
Property Implementation in Components
Details custom properties in components.
Default Properties for Your Components
Discusses how default properties and indexers can be used in components.
Exposing Properties of Contained Objects
Describes how to make available a property of a privately contained object in your component.
Saving the Properties of Components
Describes how to serialize and deserialize the properties of your component.

Related Sections

Component Authoring
Provides an overview of issues associated with component authoring.
Walkthrough: Authoring a Component with Visual Basic .NET
Details how to create a basic component step-by-step.
Walkthrough: Authoring a User Control With Visual Basic .NET
Provides step-by-step instructions for creating a simple user control.

Method Implementation in Components

Methods are blocks of code that are implemented in your component to provide functionality to the user. They provide the functionality that lets your component accomplish specific tasks. Methods are of two types: value-returning and non–value-returning.

In Visual Basic, if a method is required to return a value, it is implemented as a **Public Function** procedure. If no value is returned, it is implemented as a **Public Sub** procedure. Methods are declared using the following syntax:

```
' Visual Basic
Public Function ConvertMatterToEnergy(matter as Integer) As Integer
   Dim energy As Integer
   ' Insert code to convert matter to energy.
   Return energy
End Function
```

Because functions return a value, they must specify a return type, such as **Integer**, **String**, **Object**, and so on The arguments **Function** or **Sub** procedures take, if any, must also be specified.

C# makes no distinction between functions and sub procedures, as Visual Basic does. The method declaration indicates the type of the return value. The **void** keyword is used if the method does not return a value. The syntax for declaring a C# public method is:

```
// C#
public int ConvertMatterToEnergy(int matter)
{
   int energy;
   // Insert code to convert matter to energy.
   return energy;
}
```

When you declare a method, declare all of its arguments as explicit data types whenever possible. Arguments that take object references should be declared as specific class types — for example, **As Widget** instead of **As Object**. For C#, use **widget w** instead of **object w**.

Typed arguments allow many errors to be caught when compiled. The compiler always catches errors, whereas run-time testing is only as good as the coverage of the test suite.

Overloaded Methods

If you want to allow users of your component to supply different combinations of parameters to a method, provide multiple overloads of the method, using explicit data types. Do not create parameters declared **As Object** that can contain any data type.

> **Visual Basic Note** The universal data type in the common language specification is **Object** rather than **Variant**. **Variant** has been removed from the language. For more information, see Universal Data Type Changes in Visual Basic.

For example, the Spin method of a hypothetical Widget object might allow either direct specification of spin direction and speed, or specification of another Widget object from which angular momentum is to be absorbed.

```
' Visual Basic
Overloads Public Sub Spin( _
   ByVal spinDirection As SpinDirectionsEnum, _
   ByVal revolutionsPerSecond As Double)
   ' Insert code to implement the method.
End Sub
Overloads Public Sub Spin(ByVal driver As Widget) _
   ' Insert code to implement the method.
End Sub

// C#
public void Spin(SpinDirectionsEnum spinDirection,
double revolutionsPerSecond)
{
   // Insert code to implement the method.
```

```
}

public void Spin(Widget driver)
{
   // Insert code to implement the method.
}
```

Iterative Design

As you implement functionality in your component, it is important to test it in the scenarios where you expect it to be used. Testing often reveals additional overloaded methods that will be needed by users of your component. For example, consider the hypothetical Widget object from the previous section. After testing your Widget object, you might realize that there are situations where you need it to spin, but cannot provide parameters to either of the overloads of the Spin method. You might solve this problem by adding an overload of the Spin method that takes no parameters to be invoked in these cases. Such an example is shown below:

```
' Visual Basic
Overloads Public Sub Spin()
   ' Insert code to overload the method.
End Sub

// C#
public void Spin()
{
   // Insert code to overload the method.
}
```

See Also

Implementing Properties, Methods, Members, and Events in Components | Overloaded Properties and Methods

Event Implementation for Components and Custom Controls with Visual Basic

Events are ways for your control to communicate with the rest of the program. You *raise* an event in code after predetermined conditions are met. For example, if you were authoring a control that provided a password-protected login, you might raise an event after three unsuccessful login attempts. Once an event is raised, it may be *handled* by the hosting form or control, or it may be ignored altogether. An event is handled by a special method called an *event handler*, which is invoked when the event is raised.

Implementation of events in components is essentially the same as implementation of events in custom controls. The difference, however, is that components do not have a user interface. Therefore, they will usually not respond to events that occur through interaction with the user (for example, mouse clicks). More often, components will respond to events raised by other components. Because event implementation is somewhat different between Visual Basic and C#, there are separate topics for each language. If you are using C#, see Events Tutorial

Declaring an Event in Visual Basic

For an event to be raised, it must first be declared. The following example shows how to declare an event:

```
Public Event TooManyUsersLoggedOn(Users as Integer)
```

There now exists an event called TooManyUsersLoggedOn. It passes a single parameter, Users, when it is raised. An event can have multiple parameters, or none, depending on what is required by the author.

Raising an Event in Visual Basic

Once an event has been declared, it can be raised in code. Raising the event causes the event and its parameters to be shuttled to its event handler, whereupon the code in the event handler executes. An event can be raised in the following manner.

```
RaiseEvent TooManyUsersLoggedOn(6)
```

When an event is raised, it must provide the same number and type of parameters as are in its declaration. Parameters can be used by events to carry information between parts of a program that might not be able to communicate otherwise. This is particularly useful in multi-threaded applications where parameters are the best means of passing methods between threads. For more information, see Walkthrough: Authoring a Simple Multithreaded Component with Visual Basic.

Handling an Event in Visual Basic

After an event is raised, one of two things can occur:

- If there is no handler for the event, it is ignored, and program execution proceeds normally.
- If there is a handler for the event, the code within that method executes and the program proceeds from there.

There are two ways to add an event handler:

- Using the **Handles** keyword and **WithEvents**
- Using the **AddHandler** keyword

The Handles Keyword

The first way to add an event handler is by using the **Handles** keyword as shown:

```
Public Sub RemoveAUser(ByVal Users as Integer) Handles _
   Mycontrol.TooManyUsersLoggedOn
' Insert implementation here
End Sub
```

Using this syntax, the method RemoveAUser will be invoked whenever the TooManyUsersLoggedOn event of MyControl is raised. After this method completes, the program execution will return to the line after where the event was raised. Note that in order to add an event handler in this manner, the control where the event originates must be declared using the keyword **WithEvents**, as shown in the following example:

```
Private WithEvents MyControl as MyUserControl
```

Dynamic Event Handlers with the AddHandler Keyword

A second method of adding an event handler, which does not require the **WithEvents** keyword, is to add the event handler dynamically using the **AddHandler** keyword. Using this method, handlers may be created or removed at run time in accordance with the needs of the program. The **AddHandler** keyword is used as follows:

```
AddHandler MyControl.TooManyUsersLoggedOn, AddressOf RemoveAUser
```

This line dynamically creates a handler for the TooManyUsersLoggedOn event of MyControl. The **AddressOf** keyword creates a delegate that points to the method RemoveAUser, and directs the event to that method. Note that a method added as a handler in this way must take the same number and type of parameters as the event passes.

A handler may also be removed dynamically, using the **RemoveHandler** keyword and similar syntax, as shown:

```
RemoveHandler MyControl.TooManyUsersLoggedOn, AddressOf RemoveAUser
```

See Also

AddHandler and RemoveHandler | WithEvents | Walkthrough: Declaring and Raising Events | Walkthrough: Authoring a Simple Multithreaded Component with Visual Basic | Events Tutorial

Troubleshooting Inherited Event Handlers in Visual Basic .NET

This topic lists common issues that arise with event handlers in inherited components. For more information, see Overriding Event Handlers with Visual Basic .NET.

Code in Event Handler Executes Twice for Every Call

- An inherited event handler must not include a Handles clause. The method in the base class is already associated with the event and will fire accordingly. Remove the **Handles** clause from the inherited method.

```
' INCORRECT
Protected Overrides Sub Button1_Click(ByVal sender As System.Object, _
   ByVal e as System.EventArgs) Handles Button1.Click
   ' The Handles clause will cause all code
   ' in this block to be executed twice.
End Sub

' Visual Basic
Protected Overrides Sub Button1_Click(ByVal sender As System.Object, _
   ByVal e as System.EventArgs)
   ' Insert code to do something here.
End Sub
```

- If the inherited method does not have a **Handles** keyword, verify that your code does not contain an extra AddHandler Statement or any additional methods that handle the same event.

See Also

Writing Event Handlers | Events and Event Handlers | Event Handler Example | Events and Delegates

Named Constants for Your Components

Constants provide a convenient way to set and refer to values that are not expected to change. Individual constants may be declared and referred to in code, or sets of related constants may be declared as enumerations.

A constant is declared by using the **Const** keyword (**const** in C#). A constant may represent any intrinsic or enumeration data type (for example, **Integer**, **String**, **Decimal**), but may not be a structure or an instance of a class. Constants use the following syntax:

```
' Visual Basic
Public Const SpeedOfLightInKMPerSecond as Integer = 300000
Private Const ColorOfTheSky as String = "Blue"
Const DensityOfWater as Integer = 1

//C#
public const int SpeedOfLightInKMPerSecond = 300000;
private const string ColorOfTheSky = "Blue";
const int DensityOfWater = 1;
```

Constants and enumerations follow the same access rules as variables. A class-level constant or enumeration may be modified using the same access modifiers (for example, **Public**) as a variable. A procedure-level constant is visible only within the context of the procedure. Enumerations cannot be declared at the procedure level, and must be declared at the class or namespace level. Public constants are not recommended due to the possibility of name confusion. If you wish to have global constants, it is best to use an enum.

Enumerations

Enumerations allow you to define a set of related constants. For instance, you might create an enumeration for the days of the week, as in the following example:

```
' Visual Basic
Public Enum DaysOfTheWeek
   mydaySaturday = 1
   mydaySunday = 2
   mydayMonday = 3
   mydayTuesday = 4
   mydayWednesday = 5
   mydayThursday = 6
   mydayFriday = 7
End Enum

 //C#
public enum DaysOfTheWeek
{
   mydaySaturday = 1,
   mydaySunday = 2,
   mydayMonday = 3,
   mydayTuesday = 4,
   mydayWednesday = 5,
   mydayThursday = 6,
   mydayFriday = 7
}
```

The members of an enumeration need not be sequential. Thus, if you have some general-purpose numeric constants you wish to define for your component, you can put them into a general enum.

```
' Visual Basic
Private Enum General
   genFeetInAMile = 5280
   genIgnitionTemp = 451
End Enum

// C#
private enum General
```

```
{
   genFeetInAMile = 5280,
   genIgnitionTemp = 451,
}
```

Data Types for Enumerations

The default data type for enumerations in Visual Basic is **Integer**, but they may be either **Byte**, **Short**, **Integer**, or **Long**. In C#, it can be any of the integral types except **char**. The default is **int**. If you wish to provide constants of a data type other than those supported in enumerations (such as strings, decimals, and so on), you should do so by using the **Const** (**const** in C#) keyword.

To declare an enumeration as a variable type other than integer, use the following syntax:

```
' Visual Basic
Enum AccountType as Byte
   myAccountSavings = 1
   myAccountChecking = 2
End Enum

// C#
enum AccountType : byte
{
   myAccountSavings = 1,
   myAccountChecking = 2
}
```

See Also

Constants and Enumerations | Enum Statement | const | enum

Property Implementation in Components

Properties allow your component to store, manipulate, and retain data that it requires to run. They are similar to member variables, in that they provide storage for data, but properties are much more robust. Properties are accessed by special methods; the **Get** and **Set** accessors, which allow property values to be validated, additional code to be executed, or other tasks to be performed when the property is set or retrieved. The following example shows a hypothetical property that includes validation code:

```
' Visual Basic
' This is the private member variable that stores the value for the
' property.
Private pMyProperty as Integer
' This is the property block.
Public Property MyProperty() as Integer
   Get
      'Returns the value if pMyProperty.
      Return pMyProperty
   End Get
   Set (Value As Integer)
      ' Checks if a valid value is being assigned to the property.
      If Value > 5 Then
         Messagebox.Show( _
            "Invalid value for the property. Please try again")
      Else
         pMyProperty = Value
      End If
   End Set
End Property

// C#
// This is the private member variable that stores the value for the
// property.
int m_MyProperty;

// This is the property block.
public int MyProperty
{
   // Returns the value of m_MyProperty.
   get
   {
      return m_MyProperty;
   }

   set
   {
      // Checks if a valid value is being
      // assigned to the property.
      if (value > 5)
      {
         MessageBox.Show(
         "Invalid value for the property. Please try again");
      }
      else
      {
         m_MyProperty = value;
      }
   }
}
```

Read-Only Properties

Properties are by default read-write, meaning the values stored in them can be both retrieved and set. Occasionally, you may wish to create a property that cannot be set, but can only be retrieved. These are called *read-only properties*. Read-only properties do not have a **Set** accessor, and their values cannot be changed during run time. An example of a read-only property is shown below: (In C#, a read-only property is created by defining only the **get** method.)

```
' Visual Basic
' This member variable will hold the value of the property.
Private ReadOnly pLength as Integer
Public ReadOnly Property Length() as Integer
   Get
      Return pLength
   End Get
End Property

// C#
// This member variable will hold the value of the property.
private int pLength;

public int Length
{
   get
   {
      return pLength;
   }
}
```

The value of a read-only variable (`pLength` in the above example) can only be set in the constructor of the component or when the member variable is initialized.

Write-Only Properties

Under rare circumstances, you may wish to create a property that can only be set, but the value cannot be retrieved. You might do this in a component that alters global properties in response to an internationalization value, for example. In such a component, different methods would be executed based on the value to which the property is set. These *write-only properties*, as they are called, do not contain a **Get** accessor, and cannot be read during execution of the program. An example is shown below:

```
Private pLanguage as myenumLanguage
Public WriteOnly Property Language() as myenumLanguage
   Set (Value As myenumLanguage)
      pLanguage = Value
      Select Case Value
         Case myenumLanguage.English
            ' Code to implement English.
         Case myenumLanguage.Icelandic
            ' Code to implement Icelandic.
         Case myenumLanguage.Tagalog
            ' Code to implement Tagalog.
      End Select
   End Set
End Property

// C#
myenumLanguage m_language;
public myenumLanguage Language
{
   set
   {
      m_language = value;
      switch (value)
      {
         case myenumLanguage.English :
            // Code to implement English.
            break;
         case myenumLanguage.Icelandic :
            // Code to implement Icelandic.
            break;
         case myenumLanguage.Tagalog :
            // Code to implement Tagalog.
            break;
         default :
            break;
```

```
        }
    }
}
```

The preceding example creates a write-only property that executes different sets of code based upon the value of a property when it is written. There is no way to retrieve the property once it is set, and unless you have a specific reason for making the value of the property inaccessible, it is recommended that you use a read-write property for these constructions.

See Also

Method Implementation in Components | Event Implementation for Components and Custom Controls with Visual Basic | Exposing Properties of Contained Objects

Default Properties for Your Components

You might want to implement a default property for your component. If you mark a property as the default property of your component, the property name can be omitted in both Visual Basic and C# code. That is, the following lines are equivalent.

```
' Visual Basic
System.Diagnostics.Debug.WriteLine(myWidgets(3).Status)
System.Diagnostics.Debug.WriteLine(myWidgets.Item(3).Status)
```

You can create default properties that accept parameters (such as the **Item** property of a collection), but cannot create a default property that does not accept parameters.

> **Visual Basic Note** In previous versions of Visual Basic, you were allowed to implement parameterless default properties. This is no longer allowed. For details, see Default Property Changes in Visual Basic

The C# equivalent of a default **Item** property is an *indexer*. A C# indexer appears to a Visual Basic programmer as a default Item property, and a Visual Basic default property appears to a C# programmer as an indexer. The following line shows the usage of the indexer for the myWidgets object.

```
// C#
System.Diagnostics.Debug.WriteLine(myWidgets[1].Status);
```

Enabling this shortcut makes sense only if the primary purpose of a component is to contain other objects. Examples of a default property (Visual Basic) and an indexer (C#) are shown below.

```
' Visual Basic
' A class that contains many Widgets
Public Class Widgets
   'Default property implementation
   Default Public Property Widget(ByVal I As Integer) As Widget
      Get
      ' Implementation code to return a widget goes here.
      End Get
      Set(ByVal Value As Widget)
      ' Implementation code to set a widget goes here.
      End Set
   End Property
End Class

// C#
// Class that contains many Widgets.
public class Widgets
{
   // Indexer implementation.
   public Widget this[int index]
   {
      get
      {
         // Insert code to return a Widget.
      }
      set
      {
         // Insert code to set a Widget.
      }
   }
}
// Insert client code.
```

See Also

Default Property Changes in Visual Basic | Implementing Properties, Methods, Members, and Events in Components

Exposing Properties of Contained Objects

Your component can act as a container for other objects, such as other components or controls. This means that the container logically encloses a contained object, and the container serves as an additional layer of functionality between the contained object and the client code. An example of this relationship might be a form and a control, where the control is contained in the form. In Visual Studio designers, a contained object is by default private, and thus cannot be accessed by the developer. If you want to make properties of these contained objects available to users of your component, you must expose the properties. A property of a contained object is exposed by creating a property in the container, and using the **Get** and **Set** accessors of that property to effect the change in the private property of the contained object.

Consider a hypothetical component with a contained object named MyComponent. In this example, when the user requests the **ContainedText** property, the value stored in the **Text** property of MyComponent is delivered. When the user assigns a value to this property, that value is automatically passed to the **Text** property of MyComponent and the **Set** code will execute, changing the **Text** property of MyComponent.

The following example shows how to expose the **Text** property of the contained object:

```
' Visual Basic
Public Property ContainedText() as String
   Get
      Return MyComponent.Text
   End Get
   Set(Value as String)
      MyComponent.Text = Value
   End Set
End Property

// C#
public string ContainedText
{
   get
   {
      return MyComponent.Text;
   }
   set
   {
      MyComponent.Text = value;
   }
}
```

To expose a property of a constituent control

1. In the Code Editor for your user control, create a public property. For details, see Property Implementation in Components.
2. In the **Get** section of the property, write code that retrieves the value of the property you want to expose.
3. In the **Set** section of the property, write code that passes the value of the property to the exposed property of the constituent control.

See Also

Implementing Properties, Methods, Members, and Events in Components | Exposing Properties of Constituent Controls

Saving the Properties of Components

Occasionally, you will find it necessary to save the state of your component. You might do this to save personal information about users of your component, or to change the default configuration of a custom control, for example.

You can persist the state of a component to a file through a process called *serialization*. Serialization allows an object to be converted to a stream of data, which can then be saved to a file. The stream can be converted to an object again through deserialization.

The .NET Framework provides two formatters for serialization: The BinaryFormatter Class and SoapFormatter Class. These classes walk an *object graph* (an in-memory representation of your object) currently in memory and convert it to a stream of data. As their respective names imply, the **BinaryFormatter** converts the object graph to a binary stream (which is most useful for desktop applications), and the **SOAPFormatter** converts the object graph to Simple Object Access Protocol (SOAP) format (which is most useful for Internet applications). These streams can then be saved as files that can be deserialized and converted back to objects when needed.

Serializing the Stream

To save the state of your component

1. Apply the SerializableAttribute attribute to the component you want to serialize.

```
' Visual Basic
<Serializable()> Public Class Account
   ' Insert code to implement class.
End Class

// C#
[Serializable()]
public class Account
{
   // ' Insert code to implement class.
}
```

2. Create an instance of this class as the object you want to save.

```
' Visual Basic
Dim myAccount As New Account()

// C#
Account myAccount = new Account();
```

3. Create a **Stream** object as a file to provide a repository for your serialized data.

```
' Visual Basic
Imports System.IO
' This creates a new file with the name SavedAccount.txt, and creates
' a Stream object to write data to that file. The GetFolderPath method
' is used to get the path to the current user's Personal folder.
Dim myFileStream As Stream = File.Create _
   (System.Environment.GetFolderPath _
      (System.Environment.SpecialFolder.Personal))

// C#
using System.IO;
// This creates a new file with the name SavedAccount.txt, and creates
// a Stream object to write data to that file. The GetFolderPath method
// is used to get the path to the current user's Personal folder.
Stream myFileStream = File.Create
   (System.Environment.GetFolderPath
      (System.Environment.SpecialFolder.Personal));
```

4. Create an instance of a formatter class.

   - If you want to convert the object graph to a binary stream, create an instance of the **BinaryFormatter** class.

```
' Visual Basic
Imports System.Runtime.Serialization.Formatters.Binary
Dim serializer As New BinaryFormatter()

// C#
using System.Runtime.Serialization.Formatters.Binary;
BinaryFormatter serializer = new BinaryFormatter();
```

   - If you want to convert the object graph to a SOAP stream, create an instance of the **SoapFormatter** class.

     Note You must have a reference in your project to the **System.Runtime.Serialization.Formatters.Soap** namespace.

```
' Visual Basic
Imports System.Runtime.Serialization.Formatters.Soap
Dim serializer as New SoapFormatter

// C#
using System.Runtime.Serialization.Formatters.Soap;
SoapFormatter serializer = new SoapFormatter();
```

5. Call the **Serialize** method of your formatter to write your object to your **Stream**.

```
' Visual Basic
serializer.Serialize(myFileStream, myAccount)
myFileStream.Close()

// C#
serializer.Serialize(myFileStream, myAccount);
myFileStream.Close();
```

6. Close the file stream to complete serialization.

   You now have a saved instance of your object in binary form in the file SavedAccount.txt.

Deserializing the Stream

After you have saved you object, you will want to retrieve it at some point. The same formatter you used to serialize your component can be used to deserialize it.

To restore a saved component

1. Declare a variable of the same type as the class you want to restore.

```
' Visual Basic
Dim restoredAccount As Account

// C#
Account restoredAccount;
```

2. Create a **Stream** object and open the file that contains the serialized object.

```
' Visual Basic
Imports System.IO
Dim myFileStream As Stream = File.OpenRead _
   (System.Environment.GetFolderPath _
```

```
      (System.Environment.SpecialFolder.Personal))

// C#
using System.IO;
Stream myFileStream = File.OpenRead
   (System.Environment.GetFolderPath
      (System.Environment.SpecialFolder.Personal));
```

3. Create an instance of the same type of formatter class that was used to serialize the data.

   - The following example shows how to use the **BinaryFormatter** class.

```
' Visual Basic
Imports System.Runtime.Serialization.Formatters.Binary
Dim deserializer As New BinaryFormatter()

// C#
using System.Runtime.Serialization.Formatters.Binary;
BinaryFormatter deserializer = new BinaryFormatter();
```

   - The following example shows how to use the **SoapFormatter** class.

     Note You must have a reference to the **System.Runtime.Serialization.Formatters.Soap** namespace already in your project.

```
' Visual Basic
Imports System.Runtime.Serialization.Formatters.Soap
Dim deserializer As New SoapFormatter()

// C#
using System.Runtime.Serialization.Formatters.Soap;
SoapFormatter deserializer = new SoapFormatter();
```

4. Call the **Deserialize** method of your formatter to convert the file stream into an object graph, and use the **CType** function (Visual Basic) or an explicit cast (C#) to convert the object graph into an object of the desired type.

```
' Visual Basic
' The CType function converts the file stream specified in the first
' argument to an object of the type specified in the second argument.
' If the object in the first argument cannot be converted to the
' specified type, an error will result.
restoredAccount = CType(deserializer.Deserialize(myFileStream), _
   Account)
myFileStream.Close()

// C#
// The explicit cast converts the file stream to an object of the type // specified. If t
he object in the file stream cannot be converted to // the specified type, an error will
result.
restoredAccount = (Account)(deserializer.Deserialize(myFileStream));
myFileStream.Close();
```

Security Note You should not use this technique to store any sensitive or secret data, as it is not an acceptable way of securing data. See the System.Security.Cryptography Namespace for information on creating secure encryption schemes for your data.

See Also

BinaryFormatter Class | SoapFormatter Class | SerializableAttribute Class | System.Security.Cryptography Namespace

Interfaces and Abstract Classes

Interfaces and abstract classes allow you to create definitions for component interaction. Through interfaces, you can specify methods that a component must implement without actually specifying how the method is implemented. Abstract classes allow you to create definitions for behavior while providing some common implementation for inheriting classes. Both are valuable tools for implementing polymorphic behavior in your components.

In This Section

Polymorphism in Components
: Provides an overview of polymorphism in components, and different ways you can implement polymorphic behavior in your components.

Abstract Classes
: Describes what abstract classes are and how to use them.

Interface Definition and Factoring Design Considerations
: Provides recommendations for designing your own interfaces.

Implementing Existing Interfaces
: Describes how to implement an existing interface.

Recommendations for Abstract Classes vs. Interfaces
: Provides recommendations for when to create an abstract class and when to create an interface.

Design Recommendations for Interfaces and Components
: Makes recommendations for best practices when designing your interfaces and components.

Interface Implementation Examples
: Provides examples of interface implementation.

Related Sections

Component Authoring
: Provides links to topics on all aspects of component authoring.

Component Authoring Walkthroughs
: Links to walkthroughs for component authoring.

Interfaces Overview
: Provides an overview of interfaces in Visual Basic.

Polymorphism in Components

Polymorphism is the ability for classes to provide different implementations of methods that are called by the same name. Polymorphism allows a method of a class to be called without regard to what specific implementation it provides. For example, you might have a class named Road which calls the Drive method of an additional class. This Car class may be SportsCar, or SmallCar, but both would provide the Drive method. Though the implementation of the Drive method would be different between the classes, the Road class would still be able to call it, and it would provide results that would be usable and interpretable by the Road class.

Polymorphism in components can be implemented in a variety of ways:

- **Interface polymorphism** - Multiple classes may implement the same *interface*, and a single class may implement one or more interfaces. Interfaces are essentially definitions of how a class needs to respond. An interface describes the methods, properties, and events that a class needs to implement, and the type of parameters each member needs to receive and return, but leaves the specific implementation of these members up to the implementing class.
- **Inheritance polymorphism** - Multiple classes may *inherit* from a single base class. By inheriting, a class receives all of the methods, properties, and events of the base class in the same implementation as the base class. Additional members can then be implemented as needed, and base members can be overridden to provide different implementations. Note that an inherited class may also implement interfaces — the techniques are not mutually exclusive.
- **Polymorphism through abstract classes** - Abstract classes provide elements of both inheritance and interfaces. An abstract class is a class that cannot be instantiated itself; it must be inherited. Some or all members of the class might be unimplemented, and it is up to the inheriting class to provide that implementation. Members that are implemented might still be overridden, and the inheriting class can still implement addition interfaces or other functionality.

These three types of polymorphism are discussed in greater detail below.

Interface Polymorphism

A powerful technique in component programming is being able to implement multiple interfaces on an object. Each interface is composed of a small group of closely related methods, properties, and events. By implementing interfaces, your component can provide functionality to any other component requiring that interface, without regard to the particular functionality contained within. This allows successive versions of components to incorporate different capability without disturbing core functionality.

The features of your component most commonly used by other developers should naturally be members of the component class itself. However, components with a large number of members can be difficult to use. Consider factoring out some features of your component as separate interfaces that are implemented privately.

Another benefit of defining features in terms of interfaces is that you can add features to your component incrementally by defining and implementing additional interfaces. Advantages include:

- Design effort is simplified, because components can begin small, with minimal functionality, and continue to provide that functionality while acquiring additional features over time, as it becomes clear from actual use what those features should be.
- Maintaining compatibility is simplified, because new versions of a component can continue to provide existing interfaces, while adding new interfaces. Subsequent versions of client applications can take advantage of these when it makes sense for them to do so.

Polymorphism Through Inheritance

Visual Basic and C# also provide polymorphism through inheritance. This is a powerful mechanism for small-scale development tasks, but has generally proven to be problematic for large-scale systems. An over-emphasis on inheritance-driven polymorphism typically results in a massive shift of resources from coding to designing, which does nothing to shorten overall development time.

Given that the real test of software is whether it works for end users, tools for rapid prototyping and rapid application development (RAD) have gained wider acceptance than tools for object-oriented programming.

When to Use Inheritance-Driven Polymorphism

The foremost use of inheritance is to add functionality to an existing base class. Programmer productivity is much greater when you start with a framework of fully debugged base classes, and methods can be incrementally added to base classes without breaking versioning.

You may also wish to use inheritance when your application design includes several related classes that must share identical implementation for certain common functions. The overlapping functionality can be implemented in a base class, from which the classes used in the application can be derived. An abstract class combines features of both inheritance and implementation, and may be useful when elements of each are needed.

Polymorphism Through Abstract Classes

Abstract classes provide features of both inheritance and interface implementation. An abstract (**MustInherit** in Visual Basic) class is a class that cannot be instantiated, and must be implemented in an inheriting class. It may contain methods and properties that are already implemented, but it may also incorporate unimplemented procedures which must be implemented in the inheriting class. This allows you to provide an invariant level of functionality in some methods of your class, while leaving flexibility options open for other procedures. An additional benefit of abstract classes is that when new versions of your component are required, additional methods may be added to the base class as needed, whereas interfaces must remain invariant.

When to Use Abstract Classes

An abstract class is useful when you need a group of related components to incorporate a set of methods with identical functionality, but also require flexibility in the implementation of other methods. Abstract classes are also valuable when versioning issues are anticipated, because the base class remains flexible and easily modified. For details, see Recommendations for Abstract Classes vs. Interfaces.

See Also

Interface Implementation Examples | Abstract Classes | Recommendations for Abstract Classes vs. Interfaces | Implementing Existing Interfaces | Design Recommendations for Interfaces and Components

Abstract Classes

Abstract classes are closely related to interfaces. They are classes that cannot be instantiated, and are frequently either partially implemented, or not at all implemented. One key difference between abstract classes and interfaces is that a class may implement an unlimited number of interfaces, but may inherit from only one abstract (or any other kind of) class. A class that is derived from an abstract class may still implement interfaces. Abstract classes are useful when creating components because they allow you specify an invariant level of functionality in some methods, but leave the implementation of other methods until a specific implementation of that class is needed. They also version well, because if additional functionality is needed in derived classes, it can be added to the base class without breaking code.

An abstract class is denoted in Visual Basic by the keyword **MustInherit**. In C#, the **abstract** modifier is used. Any methods that are meant to be invariant may be coded into the base class, but any methods that are to be implemented are marked in Visual Basic with the **MustOverride** modifier. In C#, the methods are marked **abstract**. The following example shows an abstract class:

```
' Visual Basic
Public MustInherit Class WashingMachine
   Sub New()
      ' Code to instantiate the class goes here.
   End sub
   Public MustOverride Sub Wash
   Public MustOverride Sub Rinse (loadSize as Integer)
   Public MustOverride Function Spin (speed as Integer) as Long
End Class

// C#
abstract class WashingMachine
{
   public WashingMachine()
   {
      // Code to initialize the class goes here.
   }

   abstract public void Wash();
   abstract public void Rinse(int loadSize);
   abstract public long Spin(int speed);
}
```

In the above example, an abstract class is declared with one implemented method and three unimplemented methods. A class inheriting from this class would have to implement the Wash, Rinse, and Spin methods. The following example shows what the implementation of this class might look like:

```
' Visual Basic
Public Class MyWashingMachine
   Inherits WashingMachine
   Public Overrides Sub Wash()
      ' Wash code goes here
   End Sub
   Public Overrides Sub Rinse (loadSize as Integer)
      ' Rinse code goes here
   End Sub
   Public Overrides Function Spin (speed as Integer) as Long
      ' Spin code goes here
   End Sub
End Class

// C#
class MyWashingMachine : WashingMachine
{
   public MyWashingMachine()
   {
      // Initialization code goes here.
   }

   override public void Wash()
   {
```

```
        // Wash code goes here.
    }

    override public void Rinse(int loadSize)
    {
        // Rinse code goes here.
    }

    override public long Spin(int speed)
    {
        // Spin code goes here.
    }
}
```

When implementing an abstract class, you must implement each abstract (**MustOverride**) method in that class, and each implemented method must receive the same number and type of arguments, and have the same return value, as the method specified in the abstract class.

See Also

Interface Implementation Examples | Implementing Existing Interfaces | abstract | MustInherit | MustOverride

Interface Definition and Factoring Design Considerations

Interfaces can be useful tools when used properly. But when used improperly, they can be cumbersome, or even become hindrances to efficient programming. The following section will provide recommendations for the best design and use of interfaces.

> **Caution** An interface once defined and accepted must remain invariant, to protect applications written to use it. Do not change interfaces once you have published them.

Interface Invariance

When you create an interface, you are creating a definition that can never change after the interface definition has been released. This *interface invariance* is an important principle of component design, because it protects existing systems that have been written to use the interface.

When an interface is clearly in need of enhancement, a new interface should be created. This interface might be named by appending a "2" onto the original interface name, to show its relationship to the existing interface.

Generating new interfaces too frequently can bulk up your components with unused interface implementations. Well-designed interfaces tend to be small and independent of each other, reducing the potential for performance problems.

Factoring Interfaces

The process of determining what properties and methods belong on an interface is called *factoring*.

In general, you should group a few closely related functions in an interface. Too many functions make the interface unwieldy, while dividing the parts of a feature too finely results in extra overhead and diminished ease of use. For example, the following code calls methods on two different interfaces of the BusinessAccount class:

```
' Visual Basic
Public Sub MergeAccounts (ByVal act1 As _
      BusinessAccount, ByVal act2 As BusinessAccount)
   Dim pac As IAccount = act1
   Dim company As ICustomer = act1
   act1.Merge(act2)
   company.ChangeAddress()
   pac.PostInterest
End Sub

// C#
public void MergeAccounts(BusinessAccount account1,
   BusinessAccount account2)
{
   IAccount account = account1;
   ICustomer customer = account1;
   account1.Merge(account2);
   customer.ChangeAddress();
   account.PostInterest();
}
```

It is much harder to go wrong in designing interfaces than in creating large inheritance trees. If you start small, you can have parts of a system running relatively quickly. The ability to evolve the system by adding interfaces allows you to gain the advantages object-oriented programming was intended to provide.

> **Note** The capital "I" in front of interface names is a convention. It is not strictly necessary to follow this convention. However, it provides an easy way to distinguish between interfaces and the classes that implement them.

See Also

Polymorphism in Components

Implementing Existing Interfaces

An interface is a contract. It defines a standard set of properties, methods, and events that are to be found in any class that implements the interface. This insures that if your class for example implements IWidget, any other class that expects to interact with IWidget can interact with your class, regardless of what other functions your class may perform.

Interfaces and their member functions and properties are implemented in Visual Basic using the **Implements** keyword. In C#, interfaces to be implemented follow a colon (:) in the class declaration and are separated on a single line by commas.

The property and method definitions must take the exact form defined by the interface. For example, say you were implementing the following interface:

```
' Visual Basic
Public Interface IAccount
   Sub DeductFees(Fees as IFeeSchedule)
   Sub PostInterest()
   Property Balance() As Decimal
End Interface

// C#
public interface IAccount
{
   void DeductFees(IFeeSchedule feeSchedule);
   void PostInterest();
   decimal Balance
   {
      get;
      set;
   }
}
```

In the code that you would write to implement these properties and methods, you would have to follow the construction of these procedures exactly. For instance, the following line would be illegal:

```
' Visual Basic
Public Property MyBalance() As Double Implements IAccount.Balance

// C#
double IAccount.MyBalance
```

This line is not a legal implementation of the interface because it declares the property as a double, instead of an integer. Likewise, a method must take the same number and type of arguments as are defined in the interface. It also has to have the same name.

To implement an existing interface

1. Declare a class using the standard syntax.
2. Within the class declaration, define the interface you are implementing. A single class may implement more than one interface.

   The following examples show how to specify that the IAccount, ICustomer, and IPortfolio interfaces be implemented:

   ```
   ' Visual Basic
   Public Class BusinessAccount
      Implements IAccount
      Implements ICustomer
      Implements IPortfolio
      ' Code to implement class here.
   End Class

   // C#
   public class BusinessAccount : IAccount, ICustomer, IPortfolio
   {
   ```

```
    // Code to implement class here.
}
```

3. Implement each property and method required by the interface. Examples are shown below:

```
' Visual Basic
Public Sub BuyEquity() Implements IPortfolio.PurchaseEquity
   ' Code to carry out ths procedure goes here.
End Sub

Public Property Balance() as Decimal Implements IAccount.Balance
   ' Appropriate code goes here.
End Property

Public Function ChangeAddress(NewAddress as IAddress) as Boolean _
   Implements ICustomer.ChangeAddress
   ' Appropriate code is here.
End Function

// C#
bool ICustomer.ChangeAddress(IAddress newAddress)
{
   // Appropriate code goes here.
}

void IPortfolio.BuyEquity()
{
   // Code to carry out this procedure goes here.
}

public decimal Balance
{
   // Appropriate code goes here.
}
```

Note In Visual C#, you cannot implement an interface member with a method of a different name, such as you can in Visual Basic .NET using the **Implements** keyword.

4. Write any other code you require your class to contain.

See Also

Interface Definition and Factoring Design Considerations | Design Recommendations for Interfaces and Components | Recommendations for Abstract Classes vs. Interfaces

Recommendations for Abstract Classes vs. Interfaces

The choice of whether to design your functionality as an interface or an abstract class (a **MustInherit** class in Visual Basic) can sometimes be a difficult one. An *abstract class* is a class that cannot be instantiated, but must be inherited from. An abstract class may be fully implemented, but is more usually partially implemented or not implemented at all, thereby encapsulating common functionality for inherited classes. For details, see Abstract Classes.

An *interface*, by contrast, is a totally abstract set of members that can be thought of as defining a contract for conduct. The implementation of an interface is left completely to the developer.

Both interfaces and abstract classes are useful for component interaction. If a method requires an interface as an argument, then any object that implements that interface can be used in the argument. For example:

```
' Visual Basic
Public Sub Spin (ByVal widget As IWidget)
End Sub
// C#
public void Spin (IWidget widget)
{}
```

This method could accept any object that implemented IWidget as the widget argument, even though the implementations of IWidget might be quite different. Abstract classes also allow for this kind of polymorphism, but with a few caveats:

- Classes may inherit from only one base class, so if you want to use abstract classes to provide polymorphism to a group of classes, they must all inherit from that class.
- Abstract classes may also provide members that have already been implemented. Therefore, you can ensure a certain amount of identical functionality with an abstract class, but cannot with an interface.

Here are some recommendations to help you to decide whether to use an interface or an abstract class to provide polymorphism for your components.

- If you anticipate creating multiple versions of your component, create an abstract class. Abstract classes provide a simple and easy way to version your components. By updating the base class, all inheriting classes are automatically updated with the change. Interfaces, on the other hand, cannot be changed once created. If a new version of an interface is required, you must create a whole new interface.
- If the functionality you are creating will be useful across a wide range of disparate objects, use an interface. Abstract classes should be used primarily for objects that are closely related, whereas interfaces are best suited for providing common functionality to unrelated classes.
- If you are designing small, concise bits of functionality, use interfaces. If you are designing large functional units, use an abstract class.
- If you want to provide common, implemented functionality among all implementations of your component, use an abstract class. Abstract classes allow you to partially implement your class, whereas interfaces contain no implementation for any members.

See Also

Abstract Classes | Polymorphism in Components | MustInherit (Visual Basic) | abstract (C#)

Design Recommendations for Interfaces and Components

A key advantage of component software development using multiple interfaces is that your components can evolve by implementing additional interfaces as the need for new functionality arises. For example: As the BankManager system evolves into, say, the BrokerageManagement system, components that provide savings and checking account objects can be replaced by components that provide MoneyMarketAccount and MultiUseAccount objects.

A legacy application compiled to use personal accounts may be still be able to function quite nicely using the new account classes, as long as it doesn't include code specific to personal accounts.

- The key points to remember when using multiple interfaces in this fashion are: After an interface is defined and in use, it must never change. For details about *interface invariance, see* Interface Definition and Factoring Design Considerations.
- If an interface needs to be expanded, create a new interface. For details, see Interface Definition and Factoring Design Considerations.
- If versioning is an anticipated issue, consider creating an *abstract class*. Abstract classes are classes that must be implemented and cannot be instantiated on their own. An abstract class can be versioned by adding additional methods, properties and events as the need arises. By making the required change to the base class, the inheriting classes can incorporate new functionality without the need for changing all of the references to that class, as you would have to do with an interface. For details, see Recommendations for Abstract Classes vs. Interfaces.
- New versions of components can provide new features by implementing new and expanded interfaces.
- New versions of components can support legacy code by continuing to provide old interfaces.
- New versions of applications can take advantage of new features (that is, new and expanded interfaces), and if necessary can be written so as to degrade gracefully when only older interfaces are available.

See Also

Polymorphism in Components | Abstract Classes

Interface Implementation Examples

After you have identified an existing interface that would be useful to your component, or defined interfaces of your own, you can implement that interface in classes your components provide.

Suppose you had a BankManager system that included a number of components, each of which provided objects representing accounts, transactions, and business rules. For example, one component might provide a BusinessAccount class, while another provided a PersonalAccount class.

Defining a Standard Interface

You might create a standard interface named IAccount, which included PostInterest and DeductFees methods:

```
' Visual Basic code for the abstract IAccount interface module.
Public Interface IAccount
   Sub PostInterest()
   Sub DeductFees(ByVal FeeSchedule As IFeeSchedule)
End Interface

// C#
// Code for the IAccount interface module.
public interface IAccount
{
   void PostInterest();
   void DeductFees(IFeeSchedule feeSchedule);
}
```

Notice that the argument of the DeductFees method uses another interface, IFeeSchedule. One would expect this interface to contain methods describing general customer attributes, such as calculating fees for various account transactions, and that it would be implemented by many classes — BusinessAccount, PersonalAccount, PreferredAccount, and so on.

Notice also that there is no code in these methods. IAccount simply defines the interface. Implementation details will vary according to the object that implements the interface.

Implementing the Standard Interface

For example, the BusinessAccount class might implement IAccount as follows:

```
' Visual Basic
Public Class BusinessAccount
Implements IAccount

Private Sub PostInterest() Implements Iaccount.PostInterest
   ' Code to post interest using the most favorable rate.
End Sub

Private Sub DeductFees(ByVal FeeSchedule As IFeeSchedule) Implements _
   Iaccount.PostInterest
   ' Code to charge a preferred rate for various services.
End Sub
End Class

// C#
class BusinessAccount : IAccount
{
   void IAccount.PostInterest()
   {
      // Code to post interest using the most favorable rate.
   }

   void IAccount.DeductFees(IFeeSchedule feeSchedule)
   {
      // Code to change a preferred rate for various services.
   }
}
```

Note An interface is a contract. You must implement all of the properties and methods in the interface. For details, see Implementing Existing Interfaces.

Implementing the Interface Differently in Another Class

By contrast, the PersonalAccount class might implement IAccount, as shown in the following examples:

```
' Visual Basic
Public Class PersonalAccount
Implements IAccount

Private Sub PostInterest() Implements IAccount.PostInterest
   ' Calculate interest for a small account and post it.
End Sub

Private Sub DeductFees(ByVal FeeSchedule As IFeeSchedule) Implements _
   IAccount.DeductFees
   ' Calculate how many service interactions the customer has
   ' had and charge accordingly.
End Sub
End Class

// C#
class PersonalAccount : IAccount
{
   void IAccount.PostInterest()
   {
      // Calculate interest for a small account and post it.
   }

   void IAccount.DeductFees(IFeeSchedule feeSchedule)
   {
      // Calculate how many service interactions the customer
      // has had and charge accordingly.
   }
}
```

Upgrading Applications to Use a New Interface

Once you have classes that implement IAccount, you can upgrade your existing applications one by one to use the new, more competitive interface. You can access the PostInterest and DeductFees methods by assigning a BusinessAccount or PersonalAccount object to a variable of type IAccount, as shown in the following examples:

```
' Visual Basic
Dim bus As New BusinessAccount
Dim act As IAccount
act = bus
act.PostInterest()

// C#
BusinessAccount business = new BusinessAccount();
IAccount account = business;
account.PostInterest();
```

You can also declare procedure arguments as IAccount, and pass the procedure any object that implements the IAccount interface, as shown in the following examples:

```
' Visual Basic
Public Sub AccountMaintenence(ByVal Account As _
      IAccount, ByVal FeeSchedule As IFeeSchedule)
   Account.PostInterest
   Account.DeductFees(FeeSchedule)
End Sub
```

```
// C#
public void AccountMaintenance(IAccount account, IFeeSchedule feeSchedule)
{
   account.PostInterest();
   account.DeductFees(feeSchedule);
}
```

The procedure shown above could be called with any account as the first argument, and any fee schedule as the second. The caller of the procedure can use whatever account is most appropriate for the occasion.

See Also

Implementing Existing Interfaces | Abstract Classes | Polymorphism in Components

Multithreading in Components

Multithreading, or *free threading*, refers to the ability of a program to execute multiple threads of operation simultaneously. An example of a multithreaded application might be a program that receives user input on one thread, performs a variety of complex calculations on a second thread, and updates a database on a third thread. In a single-threaded application, a user might spend idle time waiting for the calculations or database updates to finish. In a multithreaded application, these processes can proceed in the background so user time is not wasted.

Multithreading can be a powerful tool to use in component programming. By writing multithreaded components, you can create components that perform complex calculations in the background while leaving the user interface free to respond to user input.

Although multithreading can be a powerful tool, it can also be difficult to apply correctly. Improperly implemented multithreaded code can degrade application performance, or even cause frozen applications. The following topics will introduce you to some of the considerations and best practices for multithreaded programming.

In This Section

Walkthrough: Authoring a Simple Multithreaded Component with Visual Basic
Demonstrates creating a multithreaded component using Visual Basic.
Walkthrough: Authoring a Simple Multithreaded Component with Visual C#
Demonstrates creating a multithreaded component using Visual C#.
Creating Threads
Explains how to create a new thread of execution.
Coordinating Multiple Threads of Execution
Describes how to use locks to coordinate threads and ensure atomic execution of code.
Thread-Safe Components
Discusses what makes a component thread-safe and how to implement thread-safety in your components.
Event Logs and Multithreaded Components
Explains issues involved with using event logs with mutliple threads of execution.
Logging Events for Multithreaded Components
Describes how to coordinate using an **EventLog** component with multiple threads of operation.
Manipulating Controls From Threads
Explains the special considerations when manipulating controls with multiple threads.

Related Sections

Multithreading in Visual Basic .NET
Provides links to a variety of topics about using multithreading with Visual Basic.
Threading Sample
Demonstrates multithreading in Visual C#.
Thread Class
Provides information about syntax and members from the .NET Framework reference.

Walkthrough: Authoring a Simple Multithreaded Component with Visual Basic

You can write applications that are able to perform multiple tasks simultaneously. This ability, called *multithreading*, or *free threading*, is a powerful way to design components that are processor-intensive and require user input. An example of a component that might make use of multithreading would be a component that calculates payroll information. The component could process data entered into a database by a user on one thread while processor-intensive payroll calculations were performed on another. By running these processes on separate threads, users do not need to wait for the computer to complete calculations before entering additional data. In this walkthrough, you will create a simple multithreaded component that performs multiple complex calculations simultaneously.

Creating the Project

Your application will consist of a single form and a component. The user will input values and signal to the component to begin calculations. The form will then receive values from your component and display them in label controls. The component will perform the processor-intensive calculations and signal the form when complete. You will create public variables in your component to hold values received from your user interface. You will also implement methods in your component to perform the calculations based on the values of these variables.

> **Note** While a function is usually preferable for a method that calculates a value, arguments cannot be passed between threads, nor can values be returned. There are many simple ways to supply values to threads and to receive values from them. In this demonstration, you will return values to your user interface by updating public variables, and events will be used to notify the main program when a thread has completed execution.

To create the form

1. Create a new **Windows Application** project.
2. Name the application **Calculations** and rename **Form1.vb** as **frmCalculations.vb**.

   This form will serve as the primary user interface for your application.

3. Add five **Label** controls, four **Button** controls, and one **TextBox** control to your form. Set properties for these controls as follows.

Control	Name	Text
Label1	**lblFactorial1**	(blank)
Label2	**lblFactorial2**	(blank)
Label3	**lblAddTwo**	(blank)
Label4	**lblRunLoops**	(blank)
Label5	**lblTotalCalculations**	(blank)
Button1	**btnFactorial1**	**Factorial**
Button2	**btnFactorial2**	**Factorial - 1**
Button3	**btnAddTwo**	**Add Two**
Button4	**btnRunLoops**	**Run a Loop**
TextBox1	**txtValue**	(blank)

To create the Calculator component

1. From the **Project** menu, select **Add Component**.
2. Name this component **Calculator**.

To add public variables to the Calculator component

1. Open the Code Editor for **Calculator**.
2. Add statements to create public variables that you will use to pass values from **frmCalculations** to each thread.

   The variable `varTotalCalculations` will keep a running total of the total number of calculations performed by the component, and the other variables will receive values from the form.

```
Public varAddTwo As Integer
Public varFact1 As Integer
```

```
Public varFact2 As Integer
Public varLoopValue As Integer
Public varTotalCalculations As Double = 0
```

To add methods and events to the Calculator component

1. Declare the events that your component will use to communicate values to your form. Immediately beneath the variable declarations entered in the previous step, type the following code:

```
Public Event FactorialComplete(ByVal Factorial As Double, ByVal _
   TotalCalculations As Double)
Public Event FactorialMinusComplete(ByVal Factorial As Double, ByVal _
   TotalCalculations As Double)
Public Event AddTwoComplete(ByVal Result As Integer, ByVal _
   TotalCalculations As Double)
Public Event LoopComplete(ByVal TotalCalculations As Double, ByVal _
   Counter As Integer)
```

2. Immediately beneath the variable declarations entered in step 1, type the following code:

```
' This sub will calculate the value of a number minus 1 factorial
' (varFact2-1!).
Public Sub FactorialMinusOne()
   Dim varX As Integer = 1
   Dim varTotalAsOfNow As Double
   Dim varResult As Double = 1
   ' Performs a factorial calculation on varFact2 - 1.
   For varX = 1 to varFact2 - 1
      varResult *= varX
      ' Increments varTotalCalculations and keeps track of the current
      ' total as of this instant.
      varTotalCalculations += 1
      varTotalAsOfNow = varTotalCalculations
   Next varX
   ' Signals that the method has completed, and communicates the
   ' result and a value of total calculations performed up to this
   ' point
   RaiseEvent FactorialMinusComplete(varResult, varTotalAsOfNow)
End Sub

' This sub will calculate the value of a number factorial (varFact1!).
Public Sub Factorial()
   Dim varX As Integer = 1
   Dim varResult As Double = 1
   Dim varTotalAsOfNow As Double = 0
   For varX = 1 to varFact1
       varResult *= varX
       varTotalCalculations += 1
       varTotalAsOfNow = varTotalCalculations
   Next varX
   RaiseEvent FactorialComplete(varResult, varTotalAsOfNow)
End Sub

' This sub will add two to a number (varAddTwo + 2).
Public Sub AddTwo()
   Dim varResult As Integer
   Dim varTotalAsOfNow As Double
   varResult = varAddTwo + 2
   varTotalCalculations += 1
```

```
    varTotalAsOfNow = varTotalCalculations
    RaiseEvent AddTwoComplete(varResult, varTotalAsOfNow)
End Sub

' This method will run a loop with a nested loop varLoopValue times.
Public Sub RunALoop()
    Dim varX As Integer
    Dim varY As Integer
    Dim varTotalAsOfNow As Double
    For varX = 1 To varLoopValue
        ' This nested loop is added solely for the purpose of slowing
        ' down the program and creating a processor-intensive
        ' application.
        For varY = 1 To 500
            varTotalCalculations += 1
            varTotalAsOfNow = varTotalCalculations
        Next
    Next
    RaiseEvent LoopComplete(varTotalAsOfNow, varX - 1)
End Sub
```

Transferring User Input to the Component

The next step is to add code to **frmCalculations** to receive input from the user and to transfer and receive values to and from the Calculator component.

To implement front-end functionality to frmCalculations

1. Open **frmCalculations** in the Code Editor.
2. Locate the `Inherits System.Windows.Forms.Form` statement. Immediately beneath it type:

```
Dim WithEvents Calculator1 As Calculator
```

3. Locate `Public Sub New()`. Immediately before the `End Sub` statement, add the following line:

```
' Creates a new instance of Calculator.
Calculator1 = New Calculator
```

4. Locate the `End Class` statement at the bottom of the Code Editor. Immediately above it, add the following code to handle button clicks:

```
Protected Sub btnFactorial1_Click(ByVal sender As Object, ByVal e As _
    System.EventArgs) Handles btnFactorial1.Click
    ' Passes the value typed in the txtValue to Calculator.varFact1.
    Calculator1.varFact1 = CInt(txtValue.Text)
    ' Disables the btnFactorial1 until this calculation is complete.
    btnFactorial1.Enabled = False
    Calculator1.Factorial()
End Sub

Protected Sub btnFactorial2_Click(ByVal sender As Object, ByVal e _
    As System.EventArgs) Handles btnFactorial2.Click
    Calculator1.varFact2 = CInt(txtValue.Text)
    btnFactorial2.Enabled = False
    Calculator1.FactorialMinusOne()
End Sub

Protected Sub btnAddTwo_Click(ByVal sender As Object, ByVal e As _
    System.EventArgs) Handles btnAddTwo.Click
    Calculator1.varAddTwo = CInt(txtValue.Text)
```

```
    btnAddTwo.Enabled = False
    Calculator1.AddTwo()
End Sub

Protected Sub btnRunLoops_Click(ByVal sender As Object, ByVal e As _
    System.EventArgs) Handles btnRunLoops.Click
    Calculator1.varLoopValue = CInt(txtValue.Text)
    btnRunLoops.Enabled = False
    ' Lets the user know that a loop is running.
    lblRunLoops.Text = "Looping"
    Calculator1.RunALoop()
End Sub
```

5. Beneath the code you added in the previous step, type the following to handle the events your form will receive from **Calculator1**:

```
Protected Sub FactorialHandler(ByVal Value As Double, ByVal _
    Calculations As Double)
    ' Displays the returned value in the appropriate label.
    lblFactorial1.Text = Value.ToString
    ' Re-enables the button so it can be used again.
    btnFactorial1.Enabled = True
    ' Updates the label that displays the total calculations performed
    lblTotalCalculations.Text = "TotalCalculations are " & _
       Calculations.ToString
End Sub

Protected Sub Factorial1Handler(ByVal Value As Double, ByVal _
    Calculations As Double)
    lblFactorial2.Text = Value.ToString
    btnFactorial2.Enabled = True
    lblTotalCalculations.Text = "TotalCalculations are " & _
        Calculations.ToString
End Sub

Protected Sub AddTwoHandler(ByVal Value As Integer, ByVal _
    Calculations As Double)
    lblAddTwo.Text = Value.ToString
    btnAddTwo.Enabled = True
    lblTotalCalculations.Text = "TotalCalculations are " & _
        Calculations.ToString
End Sub

Protected Sub LoopDoneHandler(ByVal Calculations As Double, ByVal _
    Count As Integer)
    btnRunLoops.Enabled = True
    lblRunLoops.Text = Count.ToString
    lblTotalCalculations.Text = "TotalCalculations are " & _
       Calculations.ToString
End Sub
```

6. In the constructor of **frmCalculations**, add the following code immediately before the **End Sub** line to handle the custom events your form will receive from **Calculator1**:

```
AddHandler Calculator1.FactorialComplete, AddressOf FactorialHandler
AddHandler Calculator1.AddTwoComplete, AddressOf AddTwoHandler
AddHandler Calculator1.FactorialMinusComplete, AddressOf _
    Factorial1Handler
AddHandler Calculator1.LoopComplete, AddressOf LoopDoneHandler
```

These statements add event handlers to handle the custom events your form will receive from **Calculator1**. The AddressOf Operator creates a delegate to the procedure that is specified, and the AddHandler Statement designates that delegate to handle the event specified.

Testing Your Application

You have now created a project that incorporates a form and a component capable of performing several complex calculations. Though you have not implemented multithreading capability yet, you will test your project to verify its functionality before proceeding.

To test your project

1. From the **Debug** menu, choose **Start**. The application begins and **frmCalculations** appears.
2. In the text box, type **4**, then click the button labeled **Add two**.

   The numeral "6" should be displayed in the label beneath it, and "Total Calculations are 1" should be displayed in **lblTotalCalculations**.

3. Now click the button labeled **Factorial - 1**.

   The numeral "6" should be displayed beneath the button, and **lblTotalCalculations** now reads "Total Calculations are 4."

4. Change the value in the text box to **20**, then click the button labeled **Factorial**.

   The number "2.43290200817664E+18" is displayed beneath it, and **lblTotalCalculations** now reads "Total Calculations are 24."

5. Change the value in the text box to **50000**, and then click the button labeled **RunALoop**.

   Note that there is a small but noticeable interval before this button is reenabled. The label under this button should read "50000" and the total calculations displayed are "25000024."

6. Change the value in the text box to **5000000** and click the button labeled **Run A Loop**, then immediately click the button labeled **Add Two**. Click it again.

   The button does not respond, nor will any control on the form respond until the loops have been completed.

   If your program runs only a single thread of execution, processor-intensive calculations such as the above example have the tendency to tie up the program until the calculations have been completed. In the next section, you will add multithreading capability to your application so that multiple threads may be run at once.

Adding Multithreading Capability

The previous example demonstrated the limitations of applications that run only a single thread of execution. In the next section you will use the Thread Class to add multiple threads of execution to your component.

To add the Threads subroutine

1. Open **Calculator.vb** in the Code Editor. Near the top of the code, locate the `Inherits System.ComponentModel.Component` line. Immediately beneath it, type the following:

```
' Declares the variables you will use to hold your thread objects.
Public FactorialThread As System.Threading.Thread
Public FactorialMinusOneThread As System.Threading.Thread
Public AddTwoThread As System.Threading.Thread
Public LoopThread As System.Threading.Thread
```

2. Immediately before the `End Class` statement at the bottom of the code, add the following method:

```
Public Sub ChooseThreads(ByVal threadNumber As Integer)
' Determines which thread to start based on the value it receives.
   Select Case threadNumber
      Case 1
         ' Sets the thread using the AddressOf the subroutine where
         ' the thread will start.
         FactorialThread = New System.Threading.Thread(AddressOf _
```

```
            Factorial)
         ' Starts the thread.
         FactorialThread.Start()
      Case 2
         FactorialMinusOneThread = New _
            System.Threading.Thread(AddressOf FactorialMinusOne)
         FactorialMinusOneThread.Start()
      Case 3
         AddTwoThread = New System.Threading.Thread(AddressOf AddTwo)
         AddTwoThread.Start()
      Case 4
         LoopThread = New System.Threading.Thread(AddressOf RunALoop)
         LoopThread.Start()
   End Select
End Sub
```

When a **Thread** object is instantiated, it requires an argument in the form of a **ThreadStart** object. The **ThreadStart** object is a delegate that points to the address of the subroutine where the thread is to begin. A **ThreadStart** object cannot take parameters or pass values, and thus cannot indicate a function. The **AddressOf** operator returns a delegate that serves as the **ThreadStart** object. The **ChooseThreads** sub you just implemented will receive a value from the program calling it and use that value to determine the appropriate thread to start.

To add the thread-starting code to frmCalculations

1. Open the **frmCalculations.vb** file in the Code Editor. Locate `Sub btnFactorial1_Click`.
   a. Comment out the line that calls the **Calculator1.Factorial** method directly as shown:

   ```
   ' Calculator1.Factorial
   ```

   b. Add the following line to call the **Calculator1.ChooseThreads** method:

   ```
   ' Passes the value 1 to Calculator1, thus directing it to start the ' correct thread
   .
   Calculator1.ChooseThreads(1)
   ```

2. Make similar modifications to the other button_click subroutines.

   Note Be sure to include the appropriate value for the threads argument.

   When you have finished, your code should look similar to the following:

```
Protected Sub btnFactorial1_Click(ByVal sender As Object, ByVal e As _
   System.EventArgs) Handles btnFactorial1.Click
   ' Passes the value typed in the txtValue to Calculator.varFact1.
   Calculator1.varFact1 = CInt(txtValue.Text)
   ' Disables the btnFactorial1 until this calculation is complete.
   btnFactorial1.Enabled = False
   ' Calculator1.Factorial()
   ' Passes the value 1 to Calculator1, thus directing it to start the
   ' Correct thread.
   Calculator1.ChooseThreads(1)
End Sub

Protected Sub btnFactorial2_Click(ByVal sender As Object, ByVal e As _
   System.EventArgs) Handles btnFactorial2.Click
   Calculator1.varFact2 = CInt(txtValue.Text)
   btnFactorial2.Enabled = False
   ' Calculator1.FactorialMinusOne()
   Calculator1.ChooseThreads(2)
```

```
End Sub

Protected Sub btnAddTwo_Click(ByVal sender As Object, ByVal e As _
   System.EventArgs) Handles btnAddTwo.Click
   Calculator1.varAddTwo = CInt(txtValue.Text)
   btnAddTwo.Enabled = False
   ' Calculator1.AddTwo()
   Calculator1.ChooseThreads(3)
End Sub

Protected Sub btnRunLoops_Click(ByVal sender As Object, ByVal e As _
   System.EventArgs) Handles btnRunLoops.Click
   Calculator1.varLoopValue = CInt(txtValue.Text)
   btnRunLoops.Enabled = False
   ' Lets the user know that a loop is running.
   lblRunLoops.Text = "Looping"
   ' Calculator1.RunALoop()
   Calculator1.ChooseThreads(4)
End Sub
```

Marshalling Calls to Controls

You will now facilitate updating the display on the form. Because controls are always owned by the main thread of execution, any call to a control from a subordinate thread requires a *marshalling* call. Marshalling is the act of moving a call across thread boundaries, and is very expensive in terms of resources. To minimize the amount of marshalling that needs to occur, and to be sure that your calls are handled in a thread-safe manner, you will use the Control.BeginInvoke Method to invoke methods on the main thread of execution, thereby minimizing the amount of cross-thread-boundary marshalling that must occur. This kind of call is necessary when calling methods that manipulate controls. For further information, see Manipulating Controls From Threads.

To create the control-invoking procedures

1. Open the Code Editor for **frmCalculations**. In the declarations section, add the following code.

```
Public Delegate Sub FHandler(ByVal Value As Double, ByVal _
   Calculations As Double)
Public Delegate Sub A2Handler(ByVal Value As Integer, ByVal _
   Calculations As Double)
Public Delegate Sub LDhandler(ByVal Calculations As Double, ByVal _
   Count As Integer)
```

   Invoke and **BeginInvoke** require a delegate to the appropriate method as an argument. These lines declare the delegate signatures that will be used by **BeginInvoke** to invoke the appropriate methods.

2. Add the following empty methods to your code.

```
Public Sub FactHandler(ByVal Value As Double, ByVal Calculations As _
   Double)
End Sub
Public Sub Fact1Handler(ByVal Value As Double, ByVal Calculations As _
   Double)
End Sub
Public Sub Add2Handler(ByVal Value As Integer, ByVal Calculations As _
   Double)
End Sub
Public Sub LDoneHandler(ByVal Calculations As Double, ByVal Count As _
   Integer)
End Sub
```

3. From the **Edit** menu, use **Cut** and **Paste** to cut all the code from `Sub FactorialHandler` and paste it into `FactHandler`.
4. Repeat the previous step for Factorial1Handler and Fact1Handler, AddTwoHandler and Add2Handler, and

LoopDoneHandler and LDoneHandler.

When finished, there should be no code remaining in `FactorialHandler`, `Factorial1Handler`, `AddTwoHandler`, and `LoopDoneHandler`, and all the code these used to contain should have been moved to the appropriate new methods.

5. Call the **BeginInvoke** method to invoke the methods asynchronously. You can call **BeginInvoke** from either your form (**me**) or any of the controls on the form:

```
Protected Sub FactorialHandler(ByVal Value As Double, ByVal _
   Calculations As Double)
   ' BeginInvoke causes asynchronous execution to begin at the address
   ' specified by the delegate. Simply put, it transfers execution of
   ' this method back to the main thread. Any parameters required by
   ' the method contained at the delegate are wrapped in an object and
   ' passed.
   Me.BeginInvoke(New FHandler(AddressOf FactHandler), New Object() _
      {Value, Calculations})
End Sub

Protected Sub Factorial1Handler(ByVal Value As Double, ByVal _
   Calculations As Double)
   Me.BeginInvoke(New FHandler(AddressOf Fact1Handler), New Object() _
      {Value, Calculations})
End Sub

Protected Sub AddTwoHandler(ByVal Value As Integer, ByVal _
   Calculations As Double)
   Me.BeginInvoke(New A2Handler(AddressOf Add2Handler), New Object() _
      {Value, Calculations})
End Sub

Protected Sub LoopDoneHandler(ByVal Calculations As Double, ByVal _
   Count As Integer)
   Me.BeginInvoke(New LDHandler(AddressOf Ldonehandler), New Object() _
      {Calculations, Count})
End Sub
```

It may seem as though the event handler is simply making a call to the next method. The event handler is actually causing a method to be invoked on the main thread of operation. This approach saves on calls across thread boundaries and allows your multithreaded applications to run efficiently and without fear of causing lockup. For details on working with controls in a multithreaded environment, see Manipulating Controls From Threads

6. Save your work.
7. Test your solution by choosing **Start** from the **Debug** menu.
   a. Type **10000000** in the text box and click **Run A Loop**.

      "Looping" is displayed in the label beneath this button. This loop should take a significant amount of time to run. If it completes too early, adjust the size of the number accordingly.

   b. In rapid succession, click all three buttons that are still enabled. You will find that all buttons respond to your input. The label beneath **Add Two** should be the first to display a result. Results will later be displayed in the labels beneath the factorial buttons. These results evaluate to infinity, as the number returned by a 10,000,000 factorial is too large for a double-precision variable to contain. Finally, after an additional delay, results are returned beneath the **Run A Loop** button.

      As you just observed, four separate sets of calculations were performed simultaneously upon four separate threads. The user interface remained responsive to input, and results were returned after each thread completed.

Coordinating Your Threads

An experienced user of multithreaded applications may perceive a subtle flaw with the code as typed. Recall the lines of code from each calculation-performing subroutine in **Calculator.vb**:

```
varTotalCalculations += 1
varTotalAsOfNow = varTotalCalculations
```

These two lines of code increment the public variable **varTotalCalculations** and set the local variable **varTotalAsOfNow** to this value. This value is then returned to **frmCalculations** and displayed in a label control. But is the correct value being returned? If only a single thread of execution is running, the answer is clearly yes. But if multiple threads are running, the answer becomes more uncertain. Each thread has the ability to increment the variable **varTotalCalculations**. It is possible that after one thread increments this variable, but before it copies the value to **varTotalAsOfNow**, another thread could alter the value of this variable by incrementing it. This leads to the possibility that each thread is, in fact, reporting inaccurate results. Visual Basic provides the SyncLock Statement to allow synchronization of threads to ensure that each thread always returns an accurate result. The syntax for **SyncLock** is as follows:

```
SyncLock AnObject
   Insert code that affects the object
   Insert some more
   Insert even more
' Release the lock
End SyncLock
```

When the **SyncLock** block is entered, execution on the specified expression is blocked until the specified thread has an exclusive lock on the object in question. In the example shown above, execution is blocked on `AnObject`. **SyncLock** must be used with an object that returns a reference rather than a value. The execution may then proceed as a block without interference from other threads. A set of statements that execute as a unit are said to be *atomic*. When **End SyncLock** is encountered, the expression is freed and the threads are allowed to proceed normally.

To add the SyncLock statement to your application

1. Open **Calculator.vb** in the Code Editor.
2. Locate each instance of the following code:

   ```
   varTotalCalculations += 1
   varTotalAsOfNow = varTotalCalculations
   ```

   There should be four instances of this code, one in each calculation method.

3. Modify this code so that it reads as follows:

   ```
   SyncLock Me
      varTotalCalculations += 1
      varTotalAsOfNow = varTotalCalculations
   End SyncLock
   ```

4. Save your work and test it as in the previous example.

   You may notice a slight impact on the performance of your program. This is because execution of threads stops when an exclusive lock is obtained on your component. Although it ensures accuracy, this approach impedes some of the performance benefit of multiple threads. You should carefully consider the need for locking threads, and implement them only when absolutely necessary.

See Also

Programming with Components | Component Authoring Walkthroughs | Multithreading in Components | Coordinating Multiple Threads of Execution | Walkthrough: Authoring a Simple Multithreaded Component with Visual C#

Walkthrough: Authoring a Simple Multithreaded Component with Visual C#

You can write applications that are able to perform multiple tasks simultaneously. This ability, called *multithreading*, or *free threading*, is a powerful way to design components that are processor-intensive and require user input. An example of a component that might make use of multithreading would be a component that calculates payroll information. The component could process data entered into a database by a user on one thread while processor-intensive payroll calculations were performed on another. By running these processes on separate threads, users do not need to wait for the computer to complete calculations before entering additional data. In this walkthrough, you will create a simple multithreaded component that performs several complex calculations simultaneously.

Creating the Project

Your application will consist of a single form and a component. The user will input values and signal to the component to begin calculations. The form will then receive values from your component and display them in label controls. The component will perform the processor-intensive calculations and signal the form when complete. You will create public variables in your component to hold values received from your user interface. You will also implement methods in your component to perform the calculations based on the values of these variables.

> **Note** Although a function is usually preferable for a method that calculates a value, arguments cannot be passed between threads, nor can values be returned. There are many simple ways to supply values to threads and to receive values from them. In this demonstration, you will return values to your user interface by updating public variables, and events will be used to notify the main program when a thread has completed execution.

To create the form

1. Create a new **Windows Application** project.
2. Name the application **Calculations** and rename **Form1.cs** as **frmCalculations.cs**.

   This form will serve as the primary user interface for your application.

3. Double-click the form in the designer to open the **Code Editor**. From the **Edit** menu, choose **Find and Replace**, then **Replace**. Use **Replace All** to replace **Form1** with **frmCalculations**.
4. In the **Solution Explorer**, right-click **frmCalculations.cs** and select **View Designer**. The designer opens.
5. Add five **Label** controls, four **Button** controls, and one **TextBox** control to your form.
6. Set properties for these controls as follows:

Control	Name	Text
Label1	**lblFactorial1**	(blank)
Label2	**lblFactorial2**	(blank)
Label3	**lblAddTwo**	(blank)
Label4	**lblRunLoops**	(blank)
Label5	**lblTotalCalculations**	(blank)
Button1	**btnFactorial1**	**Factorial**
Button2	**btnFactorial2**	**Factorial - 1**
Button3	**btnAddTwo**	**Add Two**
Button4	**btnRunLoops**	**Run a Loop**
Textbox1	**txtValue**	(blank)

To create the Calculator component

1. From the **Project** menu, select **Add Component**.
2. Name the component **Calculator**.

To add public variables to the Calculator component

1. Open the Code Editor for **Calculator**.
2. Add statements to create public variables that you will use to pass values from **frmCalculations** to each thread.

   The variable `varTotalCalculations` will keep a running total of the total number of calculations performed by the component, and the other variables will receive values from the form.

```
public int varAddTwo;
public int varFact1;
public int varFact2;
public int varLoopValue;
public double varTotalCalculations = 0;
```

To add methods and events to the Calculator component

1. Declare the delegates for the events that your component will use to communicate values to your form.

   Note Although you will be declaring four events, you only need to create three delegates, because two events will have the same signature.

   Immediately beneath the variable declarations entered in the previous step, type the following code:

```
// This delegate will be invoked with two of your events.
public delegate void FactorialCompleteHandler(double Factorial, double TotalCalculations)
;
public delegate void AddTwoCompleteHandler(int Result, double TotalCalculations);
public delegate void LoopCompleteHandler(double TotalCalculations, int Counter);
```

2. Declare the events that your component will use to communicate with your application. Do this by adding the following code immediately beneath the code entered in the previous step.

```
public event FactorialCompleteHandler FactorialComplete;
public event FactorialCompleteHandler FactorialMinusOneComplete;
public event AddTwoCompleteHandler AddTwoComplete;
public event LoopCompleteHandler LoopComplete;
```

3. Immediately beneath the code you typed in the previous step, type the following code:

```
// This method will calculate the value of a number minus 1 factorial
// (varFact2-1!).
public void FactorialMinusOne()
{
   double varTotalAsOfNow = 0;
   double varResult = 1;
   // Performs a factorial calculation on varFact2 - 1.
   for (int varX = 1; varX <= varFact2 - 1; varX++)
   {
      varResult *= varX;
      // Increments varTotalCalculations and keeps track of the current
      // total as of this instant.
      varTotalCalculations += 1;
      varTotalAsOfNow = varTotalCalculations;
   }
   // Signals that the method has completed, and communicates the
   // result and a value of total calculations performed up to this
   // point.
   FactorialMinusOneComplete(varResult, varTotalAsOfNow);
}

// This method will calculate the value of a number factorial.
// (varFact1!)
public void Factorial()
{
   double varResult = 1;
   double varTotalAsOfNow = 0;
   for (int varX = 1; varX <= varFact1; varX++)
```

```
    {
       varResult *= varX;
       varTotalCalculations += 1;
       varTotalAsOfNow = varTotalCalculations;
    }
    FactorialComplete(varResult, varTotalAsOfNow);
 }

 // This method will add two to a number (varAddTwo+2).
 public void AddTwo()
 {
    double varTotalAsOfNow = 0;
    int varResult = varAddTwo + 2;
    varTotalCalculations += 1;
    varTotalAsOfNow = varTotalCalculations;
    AddTwoComplete(varResult, varTotalAsOfNow);
 }

 // This method will run a loop with a nested loop varLoopValue times.
 public void RunALoop()
 {
    int varX;
    double varTotalAsOfNow = 0;
    for (varX = 1; varX <= varLoopValue; varX++)
    {
     // This nested loop is added solely for the purpose of slowing down
     // the program and creating a processor-intensive application.
       for (int varY = 1; varY <= 500; varY++)
       {
          varTotalCalculations += 1;
          varTotalAsOfNow = varTotalCalculations;
       }
    }
    LoopComplete(varTotalAsOfNow, varLoopValue);
 }
```

Transferring User Input to the Component

The next step is to add code to **frmCalculations** to receive input from the user and to transfer and receive values to and from the Calculator component.

To implement front-end functionality to frmCalculations

1. Open **frmCalculations** in the Code Editor.
2. Locate the `public class frmCalculations` statement. Immediately beneath the `{` type:

   ```
   Calculator Calculator1;
   ```

3. Locate the constructor. Immediately before the `}`, add the following line:

   ```
   // Creates a new instance of Calculator.
   Calculator1 = new Calculator();
   ```

4. In the designer, click each button to generate the code outline for each control's click event handlers and add code to create the handlers.

   When complete, your click event handlers should resemble the following:

   ```
   private void btnFactorial1_Click(object sender, System.EventArgs e)
   // Passes the value typed in the txtValue to Calculator.varFact1.
   ```

```
{
   Calculator1.varFact1 = int.Parse(txtValue.Text);
   // Disables the btnFactorial1 until this calculation is complete.
   btnFactorial1.Enabled = false;
   Calculator1.Factorial();
}

private void btnFactorial2_Click(object sender, System.EventArgs e)
{
   Calculator1.varFact2 = int.Parse(txtValue.Text);
   btnFactorial2.Enabled = false;
   Calculator1.FactorialMinusOne();
}
private void btnAddTwo_Click(object sender, System.EventArgs e)
{
   Calculator1.varAddTwo = int.Parse(txtValue.Text);
   btnAddTwo.Enabled = false;
   Calculator1.AddTwo();
}
private void btnRunLoops_Click(object sender, System.EventArgs e)
{
   Calculator1.varLoopValue = int.Parse(txtValue.Text);
   btnRunLoops.Enabled = false;
   // Lets the user know that a loop is running
   lblRunLoops.Text = "Looping";
   Calculator1.RunALoop();
}
```

5. Beneath the code you added in the previous step, type the following to handle the events your form will receive from **Calculator1**:

```
protected void FactorialHandler(double Value, double Calculations)
// Displays the returned value in the appropriate label.
{
   lblFactorial1.Text = Value.ToString();
   // Re-enables the button so it can be used again.
   btnFactorial1.Enabled = true;
   // Updates the label that displays the total calculations performed
   lblTotalCalculations.Text = "TotalCalculations are " +
      Calculations.ToString();
}

protected void FactorialMinusHandler(double Value, double Calculations)
{
   lblFactorial2.Text = Value.ToString();
   btnFactorial2.Enabled = true;
   lblTotalCalculations.Text = "TotalCalculations are " +
      Calculations.ToString();
}

protected void AddTwoHandler(int Value, double Calculations)
{
   lblAddTwo.Text = Value.ToString();
   btnAddTwo.Enabled = true;
   lblTotalCalculations.Text = "TotalCalculations are " +
      Calculations.ToString();
}

protected void LoopDoneHandler(double Calculations, int Count)
```

```
{
   btnRunLoops.Enabled = true;
   lblRunLoops.Text = Count.ToString();
   lblTotalCalculations.Text = "TotalCalculations are " +
      Calculations.ToString();
}
```

6. In the constructor of **frmCalculations**, add the following code immediately before the **}** to handle the custom events your form will receive from **Calculator1**.

```
Calculator1.FactorialComplete += new
   Calculator.FactorialCompleteHandler(this.FactorialHandler);
Calculator1.FactorialMinusOneComplete += new
   Calculator.FactorialCompleteHandler(this.FactorialMinusHandler);
Calculator1.AddTwoComplete += new
   Calculator.AddTwoCompleteHandler(this.AddTwoHandler);
Calculator1.LoopComplete += new
   Calculator.LoopCompleteHandler(this.LoopDoneHandler);
```

Testing Your Application

You have now created a project that incorporates a form and a component capable of performing several complex calculations. Although you have not implemented multithreading capability yet, you will test your project to verify its functionality before proceeding.

To test your project

1. From the **Debug** menu, choose **Start**.

   The application begins and **frmCalculations** appears.

2. In the text box, type **4**, then click the button labeled **Add two**.

   The numeral "6" should be displayed in the label beneath it, and "Total Calculations are 1" should be displayed in **lblTotalCalculations**.

3. Now click the button labeled **Factorial - 1**.

   The numeral "6" should be displayed beneath the button, and **lblTotalCalculations** now reads "Total Calculations are 4."

4. Change the value in the text box to **20**, then click the button labeled **Factorial**.

   The number "2.43290200817664E+18" is displayed beneath it, and **lblTotalCalculations** now reads "Total Calculations are 24."

5. Change the value in the text box to **50000**, and then click the button labeled **RunALoop**.

   Note that there is a small but noticeable interval before this button is reenabled. The label under this button should read "50000" and the total calculations displayed are "25000024."

6. Change the value in the text box to **5000000** and click the button labeled **Run A Loop**, then immediately click the button labeled **Add Two**. Click it again.

   The button does not respond, nor will any control on the form respond until the loops have been completed.

   If your program runs only a single thread of execution, processor-intensive calculations such as the above example have the tendency to tie up the program until the calculations have been completed. In the next section, you will add multithreading capability to your application so that multiple threads may be run at once.

Adding Multithreading Capability

The previous example demonstrated the limitations of applications that run only a single thread of execution. In the next section you will use the Thread Class object to add multiple threads of execution to your component.

To add the Threads subroutine

1. Open **Calculator.cs** in the Code Editor.

2. Near the top of the code, locate the class declaration, and immediately beneath the `{`, type the following:

```
// Declares the variables you will use to hold your thread objects.
public System.Threading.Thread FactorialThread;
public System.Threading.Thread FactorialMinusOneThread;
public System.Threading.Thread AddTwoThread;
public System.Threading.Thread LoopThread;
```

3. Immediately before the end of the class declaration at the bottom of the code, add the following method:

```
public void ChooseThreads(int threadNumber)
{
// Determines which thread to start based on the value it receives.
switch(threadNumber)
   {
      case 1:
         // Sets the thread using the AddressOf the subroutine where
         // the thread will start.
         FactorialThread = new System.Threading.Thread(new
            System.Threading.ThreadStart(this.Factorial));
         // Starts the thread.
         FactorialThread.Start();
         break;
      case 2:
         FactorialMinusOneThread = new
            System.Threading.Thread(new
               System.Threading.ThreadStart(this.FactorialMinusOne));
         FactorialMinusOneThread.Start();
         break;
      case 3:
         AddTwoThread = new System.Threading.Thread(new
             System.Threading.ThreadStart(this.AddTwo));
         AddTwoThread.Start();
         break;
      case 4:
         LoopThread = new System.Threading.Thread(new
            System.Threading.ThreadStart(this.RunALoop));
         LoopThread.Start();
         break;
   }
}
```

When a **Thread** object is instantiated, it requires an argument in the form of a **ThreadStart** object. The **ThreadStart** object is a delegate that points to the address of the method where the thread is to begin. A **ThreadStart** object cannot take parameters or pass values, and therefore can only indicate a **void** method. The **ChooseThreads** method you just implemented will receive a value from the program calling it and use that value to determine the appropriate thread to start.

To add the appropriate code to frmCalculations

1. Open the **frmCalculations.cs** file in the Code Editor, then locate `protected void btnFactorial1_Click`.
   a. Comment out the line that calls the **Calculator1.Factorial1** method directly as shown:

```
// Calculator1.Factorial()
```

   b. Add the following line to call the **Calculator1.ChooseThreads** method:

```
// Passes the value 1 to Calculator1, thus directing it to start the
// correct thread.
Calculator1.ChooseThreads(1);
```

2. Make similar modifications to the other button_click subroutines.

   Note Be certain to include the appropriate value for the **Threads** argument.

   When you have finished, your code should look similar to the following:

```
protected void btnFactorial1_Click(object sender, System.EventArgs e)
// Passes the value typed in the txtValue to Calculator.varFact1
{
   Calculator1.varFact1 = int.Parse(txtValue.Text);
   // Disables the btnFactorial1 until this calculation is complete
   btnFactorial1.Enabled = false;
   // Calculator1.Factorial();
   Calculator1.ChooseThreads(1);
}

protected void btnFactorial2_Click(object sender, System.EventArgs e)
{
   Calculator1.varFact2 = int.Parse(txtValue.Text);
   btnFactorial2.Enabled = false;
   // Calculator1.FactorialMinusOne();
   Calculator1.ChooseThreads(2);
}
protected void btnAddTwo_Click(object sender, System.EventArgs e)
{
   Calculator1.varAddTwo = int.Parse(txtValue.Text);
   btnAddTwo.Enabled = false;
   // Calculator1.AddTwo();
   Calculator1.ChooseThreads(3);
}

protected void btnRunLoops_Click(object sender, System.EventArgs e)
{
   Calculator1.varLoopValue = int.Parse(txtValue.Text);
   btnRunLoops.Enabled = false;
   // Lets the user know that a loop is running
   lblRunLoops.Text = "Looping";
   // Calculator1.RunALoop();
   Calculator1.ChooseThreads(4);
}
```

Marshalling Calls to Controls

You will now facilitate updating the display on the form. Because controls are always owned by the main thread of execution, any call to a control from a subordinate thread requires a *marshalling* call. Marshalling is the act of moving a call across thread boundaries, and is very expensive in terms of resources. To minimize the amount of marshalling that needs to occur, and to be sure that your calls are handled in a thread-safe manner, you will use the Control.BeginInvoke Method to invoke methods on the main thread of execution, thereby minimizing the amount of cross-thread-boundary marshalling that must occur. This kind of call is necessary when calling methods that manipulate controls. For details, see Manipulating Controls From Threads.

To create the control-invoking procedures

1. Open the code editor for **frmCalculations**. In the declarations section, add the following code:

```
public delegate void FHandler(double Value, double Calculations);
public delegate void A2Handler(int Value, double Calculations);
public delegate void LDHandler(double Calculations, int Count);
```

   Invoke and **BeginInvoke** require a delegate to the appropriate method as an argument. These lines declare the delegate signatures that will be used by **BeginInvoke** to invoke the appropriate methods.

2. Add the following empty methods to your code.

```
public void FactHandler(double Value, double Calculations)
{
}
public void Fact1Handler(double Value, double Calculations)
{
}
public void Add2Handler(int Value, double Calculations)
{
}
public void LDoneHandler(double Calculations, int Count)
{
}
```

3. From the **Edit** menu, use **Cut** and **Paste** to cut all the code from the method `FactorialHandler` and paste it into `FactHandler`.
4. Repeat the previous step for FactorialMinusHandler and Fact1Handler, AddTwoHandler and Add2Handler, and LoopDoneHandler and LDoneHandler.

   When finished, there should be no code remaining in `FactorialHandler`, `Factorial1Handler`, `AddTwoHandler`, and `LoopDoneHandler`, and all the code these used to contain should have been moved to the appropriate new methods.

5. Call the **BeginInvoke** method to invoke the methods asynchronously. You can call **BeginInvoke** from either your form (**this**) or any of the controls on the form.

   When complete, your code should look similar to the following:

```
protected void FactorialHandler(double Value, double Calculations)
{
   // BeginInvoke causes asynchronous execution to begin at the address
   // specified by the delegate. Simply put, it transfers execution of
   // this method back to the main thread. Any parameters required by
   // the method contained at the delegate are wrapped in an object and
   // passed.
   this.BeginInvoke(new FHandler(FactHandler), new Object[]
      {Value, Calculations});
}
protected void FactorialMinusHandler(double Value, double Calculations)
{
   this.BeginInvoke(new FHandler(Fact1Handler), new Object []
      {Value, Calculations});
}

protected void AddTwoHandler(int Value, double Calculations)
{
   this.BeginInvoke(new A2Handler(Add2Handler), new Object[]
      {Value, Calculations});
}

protected void LoopDoneHandler(double Calculations, int Count)
{
   this.BeginInvoke(new LDHandler(LDoneHandler), new Object[]
      {Calculations, Count});
}
```

It may seem as though the event handler is simply making a call to the next method. The event handler is actually causing a method to be invoked on the main thread of operation. This approach saves on calls across thread boundaries and allows your multithreaded applications to run efficiently and without fear of causing lockup. For details on working with controls in

a multithreaded environment, see Manipulating Controls From Threads

6. Save your work.
7. Test your solution by choosing **Start** from the **Debug** menu.
   a. Type **10000000** in the text box and click **Run A Loop**.

      "Looping" is displayed in the label beneath this button. This loop should take a significant amount of time to run. If it completes too early, adjust the size of the number accordingly.

   b. In rapid succession, click all three buttons that are still enabled. You will find that all buttons respond to your input. The label beneath **Add Two** should be the first to display a result. Results will later be displayed in the labels beneath the factorial buttons. These results evaluate to infinity, as the number returned by a 10,000,000 factorial is too large for a double-precision variable to contain. Finally, after an additional delay, results are returned beneath the **Run A Loop** button.

      As you just observed, four separate sets of calculations were performed simultaneously upon four separate threads. The user interface remained responsive to input, and results were returned after each thread completed.

Coordinating Your Threads

An experienced user of multithreaded applications may perceive a subtle flaw with the code as typed. Recall the lines of code from each calculation-performing subroutine in **Calculator.cs**:

```
varTotalCalculations += 1;
varTotalAsOfNow = varTotalCalculations;
```

These two lines of code increment the public variable **varTotalCalculations** and set the local variable **varTotalAsOfNow** to this value. This value is then returned to **frmCalculations** and displayed in a label control. But is the correct value being returned? If only a single thread of execution is running, the answer is clearly yes. But if multiple threads are running, the answer becomes more uncertain. Each thread has the ability to increment the variable **varTotalCalculations**. It is possible that after one thread increments this variable, but before it copies the value to **varTotalAsOfNow**, another thread could alter the value of this variable by incrementing it. This leads to the possibility that each thread is, in fact, reporting inaccurate results. Visual C# provides the lock Statement to allow synchronization of threads to ensure that each thread always returns an accurate result. The syntax for **lock** is as follows:

```
lock(AnObject)
{
   // Insert code that affects the object.
   // Insert more code that affects the object.
   // Insert more code that affects the object.
// Release the lock.
}
```

When the **lock** block is entered, execution on the specified expression is blocked until the specified thread has an exclusive lock on the object in question. In the example shown above, execution is blocked on `AnObject`. **lock** must be used with an object that returns a reference rather than a value. The execution may then proceed as a block without interference from other threads. A set of statements that execute as a unit are said to be *atomic*. When the `}` is encountered, the expression is freed and the threads are allowed to proceed normally.

To add the lock statement to your application

1. Open **Calculator.cs** in the Code Editor.
2. Locate each instance of the following code:

   ```
   varTotalCalculations += 1;
   varTotalAsOfNow = varTotalCalculations;
   ```

   There should be four instances of this code, one in each calculation method.

3. Modify this code so that it reads as follows:

   ```
   lock(this)
   {
   ```

```
    varTotalCalculations += 1;
    varTotalAsOfNow = varTotalCalculations;
}
```

4. Save your work and test it as in the previous example.

   You may notice a slight impact on the performance of your program. This is because execution of threads stops when an exclusive lock is obtained on your component. Although it ensures accuracy, this approach impedes some of the performance benefits of multiple threads. You should carefully consider the need for locking threads, and implement them only when absolutely necessary.

See Also

Programming with Components | Component Authoring Walkthroughs | Multithreading in Components | Coordinating Multiple Threads of Execution | Walkthrough: Authoring a Simple Multithreaded Component with Visual Basic

Creating Threads

The .NET Framework allows you to create multithreaded applications quickly and easily by providing an object-oriented threading model. Creating a new thread is as easy as declaring it and supplying it with a delegate to the method where the thread is to start. When you are ready to begin execution on the thread, call the Thread.Start Method. There are special considerations involved when working with multiple threads of execution. For an overview of techniques involved in working with multiple threads, see Thread-Safe Components.

To create a new thread of execution

1. Declare the thread.

```
' Visual Basic
Dim myThread as System.Threading.Thread

// C#
System.Threading.Thread myThread;
```

2. Instantiate the thread with the appropriate delegate for the starting point of the thread. Use the **AddressOf** operator to create the delegate in Visual Basic, or create a new **ThreadStart** object in C#.

```
' Visual Basic
myThread = New System.Threading.Thread(AddressOf myStartingMethod)

// C#
myThread = new System.Threading.Thread(new
   System.Threading.ThreadStart(myStartingMethod));
```

3. When ready, call the **Thread.Start** method to start the thread.

```
' Visual Basic
myThread.Start()

// C#
myThread.Start();
```

See Also

Thread-Safe Components | Coordinating Multiple Threads of Execution | Multithreading in Components

Coordinating Multiple Threads of Execution

In order for your multithreaded components to be thread-safe, you must coordinate access to shared resources. If multiple threads attempt to access a shared resource at the same time, race conditions can result, causing corruption of data. You can avoid race conditions by using locks. For details on thread-safety and race conditions, see Thread-Safe Components.

To create a lock on an object

1. Identify the code that must be executed atomically and the object on which the code will be executed. For details, see Thread-Safe Components
2. Place a lock on that object and enclose your code inside that lock.

   This code will now be executed atomically on the locked object.

```
' Visual Basic
SyncLock MyObject   ' This represents the start of the lock on MyObject.
' Insert code to be executed atomically on MyObject here.
End SyncLock   ' This represents the end of the lock.

// C#
lock (MyObject)
   // All code inside the braces {} is executed with MyObject locked.
{
   // Insert code to be executed atomically on MyObject here.
}
```

See Also

Walkthrough: Authoring a Simple Multithreaded Component with Visual Basic | Walkthrough: Authoring a Simple Multithreaded Component with Visual C# | Multithreading in Components | Thread-Safe Components

Thread-Safe Components

Sharing resources between threads is a frequent necessity in multithreaded programming. Multiple threads may need to access a shared database, for instance, or make updates to a set of system variables. When more than one thread simultaneously competes for access to shared resources, the possibility of a *race condition* occurs. A race condition exists when a thread modifies a resource to an invalid state, and then another thread attempts to access that resource and use it in the invalid state. Consider the following example:

```
' Visual Basic
Public Class WidgetManipulator
Public TotalWidgets as Integer = 0
Public Sub AddWidget()
   TotalWidgets += 1
   Console.WriteLine("Total widgets = " & TotalWidgets.ToString)
End Sub
Public Sub RemoveWidgets()
   TotalWidgets -= 10
End Sub
End Class

// C#
public class WidgetManipulator
{
   public int TotalWidgets = 0;
   public void AddWidget()
   {
      TotalWidgets++;
      Console.WriteLine("Total widgets = " + TotalWidgets.ToString());
   }
   public void RemoveWidgets()
   {
      TotalWidgets -= 10;
   }
}
```

This class exposes two methods. One method, AddWidget, adds 1 to the TotalWidgets field and writes the value to the console. The second method subtracts 10 from the value of TotalWidgets. Consider what would happen if two threads simultaneously attempted to access the same instance of the WidgetManipulator class. One thread might call AddWidget at the same time that the second thread called RemoveWidgets. In that case, the value of `TotalWidgets` could be changed by the second thread before an accurate value could be reported by the first thread. This race condition can cause inaccurate results to be reported and can cause corruption of data.

Preventing Race Conditions by Using Locks

You can protect critical sections of your code from race conditions by employing *locks*. A lock, represented by the Visual Basic keyword SyncLock Statement, or the C# keyword lock Statement, allows a single thread of execution to obtain exclusive execution rights on an object. The following example demonstrates locks:

```
' Visual Basic
SyncLock MyObject
' Insert code that affects MyObject.
End SyncLock

// C#
lock(MyObject)
{
   // Insert code that affects MyObject.
}
```

When a lock is encountered, execution on the object specified (`MyObject` in the previous example) is blocked until the thread can gain exclusive access to the object. When the end of the lock is reached, the lock is freed and execution proceeds normally. You can only obtain a lock on an object that returns a reference. A value type cannot be locked in this fashion.

Disadvantages of Locks

Although using locks will guarantee that multiple threads do not simultaneously access an object, they can cause significant performance degradation. Imagine a program with many different threads running. If each thread needs to use a particular object and has to wait to obtain an exclusive lock on that object before executing, the threads will all cease executing and back up behind one another, causing poor performance. For these reasons, you should only use locks when you have code that must be executed as a unit. For example, you might update multiple resources that were interdependent. Such code is said to be *atomic*. Restricting your locks only to code that must be executed atomically will allow you to write multithreaded components that ensure the safety of your data while still maintaining good performance.

You must also be careful to avoid situations where *deadlocks* might occur. In this case, multiple threads wait for each other to release shared resources. For example, Thread 1 might hold a lock on resource A and is waiting for resource B. Thread 2, on the other hand, might have a lock on resource B and awaits resource A. In such a case, neither thread will be allowed to proceed. The only way to avoid deadlock situations is through careful programming.

See Also

Multithreading in Components | Coordinating Multiple Threads of Execution | Manipulating Controls from Threads |
Walkthrough: Authoring a Simple Multithreaded Component with Visual Basic |
Walkthrough: Authoring a Simple Multithreaded Component with Visual C#

Event Logs and Multithreaded Components

Event logs are for keeping track of significant events when an application is running. Using event logs, you can record information that might be useful for troubleshooting or performance analysis. There are additional considerations when logging events with multithreaded components. First, there must be a mechanism to record the identity of the thread logging the message. Second, thread safety must be taken into account when interacting with the event log. The thread attempting to write a message must obtain an exclusive lock on the log in order to avoid race conditions. For an overview of event logs, see Administering Event Logs. For more information on thread safety, see Thread-Safe Components.

To identify each thread, you should set the Thread.Name Property. This property takes and returns a **String** value, and can be used to set a unique identifier for each thread. This value can then be passed to the EventLog.CreateEventSource Method to designate your thread as an *event source* for each thread. When logging events, your thread can set the **Source** property of the event log to its name, thereby ensuring accurate logging of events.

When executing multiple lines of code in a multithreaded environment, it is important that the threads obtain exclusive locks on the event log before executing any code. For example, consider the following lines of code, executed in a multithreaded environment:

```
' Visual Basic
MyEventLog.Source = Threading.Thread.CurrentThread.Name.ToString
EventLog.WriteEntry("What thread did this come from?", "myApplication")

// C#
MyEventLog.Source = System.Threading.Thread.CurrentThread.Name.ToString();
EventLog.WriteEntry("What thread did this come from?", "myApplication");
```

If multiple threads are executing these lines simultaneously, if is possible for one thread to change the EventLog.Source Property of the event log, and for another thread to write a message, after that property had been changed. In order to avoid conditions such as these, you can use the SyncLock Statement (Visual Basic) or **lock** (C#) statement to obtain an exclusive lock on the object before executing code from multiple threads. Using locking, the previous example would now look like the following:

```
' Visual Basic
SyncLock MyEventLog
   MyEventLog.Source = System.Threading.Thread.CurrentThread.Name.ToString
   EventLog.WriteEntry("What thread did this come from?", "myApplication")
End SyncLock

// C#
lock(MyEventLog)
{
   MyEventLog.Source = Threading.Thread.CurrentThread.Name.ToString();
   EventLog.WriteEntry("What thread did this come from?",
      "myApplication");
}
```

You can also use the **Debug** and **Trace** classes to log events that occur in multithreaded applications. These classes are static classes that allow you to send output to the output window, the console window, a text file or event log, or any other of a number of objects. For details, see Tracing and Instrumenting Applications in Visual Basic and Visual C#.

See Also

Multithreading in Components | Logging Events for Multithreaded Components

Logging Events for Multithreaded Components

When logging events from multithreaded components, special considerations come into play. You must supply a means of identifying the thread from which the event came. You must also ensure that the threads do not interfere with one another when logging events. For details, see Event Logs and Multithreaded Components.

To use event logs in multithreaded applications

1. Declare and create the event log. For details, see Creating and Removing Custom Event Logs.
2. Set the Thread.Name Property property of each thread to a unique identifier.

```
' Visual Basic
Dim X as Integer
X = 0
Dim MyThread as New Threading.Thread(AddressOf MyMethod)
MyThread.Name = "Thread number " & X.ToString
' Increments X so no other threads will share the same name.
X += 1

// C#
int X;
X = 0;
Thread MyThread = new Thread(new
   ThreadStart(MyMethod));
MyThread.Name = "Thread Number " + X.ToString();
// Increments X so no other threads will share the same name.
X += 1;
```

3. Create a new **Source** for the event log, and set the **Name** to a unique string value that corresponds to the thread. For details on creating a **Source**, see Adding Your Application as a Source of Event Log Entries.

```
' Visual Basic
Imports System.Threading
' Checks to see if there already is a source with the name of the
' thread.
If Not EventLog.SourceExists(Thread.CurrentThread.Name.ToString, _
   "myApplication") Then
   ' Creates a source with the name of the thread
   EventLog.CreateEventSource(Thread.CurrentThread.Name.ToString, _
      "MyApplication")
End If

// C#
// These lines go at the top of the code
using System.Threading;
using System.Diagnostics;
// Checks to see if there already is a source with the name of the
// thread.
if (!EventLog.SourceExists(CurrentThread.Name.ToString()))
   // Creates a source with the name of the thread.
   EventLog.CreateEventSource(CurrentThread.Name.ToString(),
      "MyApplication");
```

4. Set the **Source** property of the event log to the name of the thread and call the **EventLog.WriteEntry** method.

   Note Be sure to obtain an exclusive lock on the event log before proceeding with these operations.

```
' Visual Basic
' Use the SyncLock keyword to obtain an exclusive lock on an object.
```

```
SyncLock MyEventLog
  MyEventLog.Source = Thread.CurrentThread.Name.ToString
  EventLog.WriteEntry(Thread.CurrentThread.Name.ToString, _
     "Error in Widget 42")
End SyncLock

// C#
// Use the lock keyword to obtain an exclusive lock on an object
lock(MyEventLog)
{
  MyEventLog.Source = Thread.CurrentThread.Name.ToString();
  EventLog.WriteEntry(Thread.CurrentThread.Name.ToString(),
     "Error in Widget 42");
}
```

See Also

Creating and Removing Custom Event Logs | Writing Entries to Event Logs | Walkthrough: Exploring Event Logs, Event Sources, and Entries | Multithreading in Components | Coordinating Multiple Threads of Execution | Walkthrough: Authoring a Simple Multithreaded Component with Visual Basic | Walkthrough: Authoring a Simple Multithreaded Component with Visual C# | SyncLock Statement

Manipulating Controls from Threads

Multithreading is best suited to running processor-intensive procedures of class modules. Unlike other components, there are issues with directly calling methods in controls from separate threads. Methods that affect controls should be executed only on the thread on which the control was created. Because marshalling calls from one thread and sending them across thread boundaries to another is very expensive in terms of system resources, you should avoid repeatedly making calls to controls on other threads. At best, direct calls from other threads are expensive, and the performance of your application will suffer. At worst, conditions could occur that cause a deadlock in your application, freezing execution.

However, there might be occasions when you want to call methods of controls from your threads. For example, you might call a method that disables a button or updates a display on a form in response to action taken by a thread. The .NET Framework provides methods that are safe to call from any thread for invoking methods that interact with controls owned by other threads. The **Control.Invoke** method allows for the synchronous execution of methods on controls, whereas the **Control.BeginInvoke** method initiates asynchronous execution. To use these methods, you must declare a delegate with the same signature as the method you will be invoking. Then you can call the **Invoke** or **BeginInvoke** method of any control on the form by supplying the appropriate delegate to the method you want to call. Any required parameters are wrapped in an **Object** and transmitted to the method.

To invoke methods involving controls owned by other threads

1. Declare a delegate with a signature identical to the method you want to invoke.

   The following example shows how to declare a delegate with **Integer** and **String** parameters.

   ```
   ' Visual Basic
   Public Delegate Sub myDelegate(ByVal anInteger as Integer, ByVal _
      aString as String)

   // C#
   public delegate void myDelegate(int anInteger, string aString);
   ```

2. Use any control to invoke methods that manipulate controls owned by other threads.

   Note Parameters (if any) required by the method can be supplied inside an **Object**.

   - If you want to invoke methods synchronously, call the Control.Invoke Method.

     ```
     ' Visual Basic
     Label1.Invoke(New myDelegate(AddressOf myMethod), New _
        Object() {1, "This is the string"})

     // C#
     Label1.Invoke(new myDelegate(myMethod), new Object[] {1,
        "This is the string"});
     ```

   - If you want to invoke methods asynchronously, call the Control.BeginInvoke Method.

     ```
     ' Visual Basic
     Label1.BeginInvoke(New myDelegate(AddressOf myMethod), _
        New Object() {1, "This is the string"})

     // C#
     Label1.BeginInvoke(new myDelegate(myMethod), new
     Object[] {1, "This is the string"});
     ```

See Also

Walkthrough: Authoring a Simple Multithreaded Component with Visual Basic | Walkthrough: Authoring a Simple Multithreaded Component with Visual C# | Multithreading in Components

Displaying Forms from Components

Although most components do not have visual interfaces, there are times when it may be useful for a user to interact visually with a component. For example, a component that represents a bank account might display a specialized alert form when the account is overdrawn. These functions should remain constant in the component and not depend on the client application. Because forms are classes, it is easy to create an instance of the form, and then display it through your component.

There are two approaches you can use to create a form:

- Create the form outside your component, which allows you to use a designer to create the look and feel of the form.
- Make the form a nested class within your component. An advantage to this approach is that the form is always a part of your component, and you have control over how it is used and exposed. The chief disadvantage, however, is that you will be unable to use a designer with your form, and will have to design it all by hand coding.

To display a form from your component

1. Create the Windows Form you want to display from your component. For details, see Adding Windows Forms to a Project.
2. If the form you want to display is not in the same project as your component, you must create a reference to the project that contains your form by right-clicking **References** in **Solution Explorer** and choosing **Add Reference**. For details, see Adding and Removing References.
3. Declare a local variable as a new instance of the form. For example, if your form was called MyForm, your code would look like the following:

```
' Visual Basic
Dim AFormInstance as New MyForm

// C#
MyForm AFormInstance = new MyForm();
```

4. Call the **Show**, **ShowDialog**, and **Hide** methods of your form to control display, as is shown in the following examples:

```
' Visual Basic
AFormInstance.Show
' Displays the form.
AFormInstance.ShowDialog
' Displays the form and waits for user interaction before continuing.
AFormInstance.Hide
' Hides the form.

// C#
// Displays the form.
AFormInstance.Show();
// Displays the form and waits for user interaction before continuing.
AFormInstance.ShowDialog();
// Hides the form.
AFormInstance.Hide();
```

See Also

Component Authoring | Forms Task Changes in Visual Basic .NET

Code Libraries

Code libraries are convenient ways to package and reuse components. They provide a portable way to encapsulate code in a single file, allow for inheritance and modification of these files, and permit distribution of discrete units of functionality. You can combine components from many different projects into a single code library. For example, you might choose a custom control from one project, a component from another project, and a form from yet another project, and put them all into your code library. You can also customize the library for special purposes. The only limitation is that all files in a single code library must be written in the same language.

A code library is basically an assembly that compiles to a DLL. For more information, see Components and Assemblies.

One of the features of the .NET Framework is the common language specification (CLS). The common language runtime compiles all code written in a CLS-compliant language to the same kind of intermediate code — MSIL. This means that code libraries written in any CLS-compliant language will be accessible from all other languages that support the common language specification. Therefore, a code library written in Visual Basic, for instance, can be used in an application written in C#, C++, or any other CLS-compliant language.

When designing your components and code libraries, it is important to keep in mind that some languages do not support optional parameters. If you think a language that does not support optional parameters may use your code library, then you should take into account that all parameters will have to be supplied for any methods that are called. One way to ensure that a method can be called with any number of optional parameters is to provide multiple overloads of that method. For more information, see Method Implementation in Components.

The access level of the component in your code library will directly determine how these components can be used by the client application. Client components can only access public components. Components marked **Friend** (**internal** in C#) are accessible only to other members of the code library. Private and protected components cannot be accessed from outside the class, (although **Protected** classes can be accessed by derived classes; for more information, see Access Modifiers).

Visual Basic allows for the implementation of modules. If you are implementing modules for distribution in a code library, it is important to realize that other programming languages that don't support modules will see modules as classes containing only shared members. Therefore, other languages will need to use the name of the module as a qualifier to access the members of the module. Accurate documentation of your code library will help prevent problems.

If you plan to make your code library available to more than one application at a time, you might want to install the assembly into the global assembly cache. For a discussion of issues involved in this, see
Working with Assemblies and the Global Assembly Cache.

If you plan to make your code library available to COM clients, there are additional considerations to take into account. For more information, see Exposing .NET Framework Components to COM.

See Also

Creating Code Libraries

Creating Code Libraries

Code libraries allow you to package several related, useful components into a single portable file. This file can then be used to develop other applications, or as a base for inheritance for other components. For more information, see Code Libraries.

To create a code library

1. Create a new **Class Library** project.

   > **Note** The Class Library template is not available in the Standard Edition of Visual Basic .NET. For more information, see Visual Basic Standard Edition Features.

   Two files are created: a default component file and the assembly manifest (AssemblyInfo.vb or AssemblyInfo.cs file), which contains metadata about the code library. For more information, see Assembly Manifest.

2. If you want to write any new code for your library:
   a. From the **Project** menu, choose **Add New Item** to add the appropriate new item (control, component, class, and so on).
   b. Write the appropriate code.
3. From the **Project** menu, choose **Add Existing Item** to add a preexisting item to your code library.
4. Browse to the appropriate **.vb** or **.cs** file and double-click the file to add it.
5. Build the solution, which will be a **.dll** file when compiled.
6. If you want to make your code library available to more than one client application at a time, you may want to install it to the global assembly cache. For more information, see Working with Assemblies and the Global Assembly Cache.
7. If you plan to make your code library available to COM clients, there are additional considerations to take into account. For more information, see Exposing .NET Framework Components to COM.
8. Package and deploy your code library. Because your library is not a freestanding application, it would be packaged as a merge module. For more information, see Creating or Adding a Merge Module Project.

See Also

Component Authoring | Components and Assemblies | Component Authoring Walkthroughs | Assembly Manifest

Components and Containers

Components are hosted by *containers*. A container is a specialized collection class that allows you to track your components, and manages the interaction of your components with other components and the external application environment. The following topics provide information on component-container interaction and describe how to use them in your applications.

In This Section

Containers, Sites, and Components
Provides information on components, sites, containers, and how they interact.
Creating Component Containers
Describes how to create a container for components.
Extending Component Containers
Details how to extend component containers to implement custom functionality.
Communication Between Containers and Components
Describes how a component interacts with its container.

Related Sections

Component Authoring
Provides links to topics on all aspects of component authoring.
Decision Chart
Provides an interactive guide to help you choose which approach or technology to use, as well as links to more information.
Visual Basic Code Example Topics
Lists quickstart-like code examples that help you perform common tasks in Visual Basic .NET.

Containers, Sites, and Components

A container is a specialized collection object that provides logical containment for one or more components. Containers manage the interaction of the components with each other, as well as with the external application environment, by providing an implementation of the **ISite** interface through which interactions can take place. The container allows tracking of your components on a first-in, first-out basis, and allows you to refer to your components by an index. The container also provides for a means of common disposal of your components when you no longer have need for them.

Containment refers to logical containment, not visual or physical containment. The container encapsulates one or more components and provides a wrapper through which clients may interact. The container allows you to add and remove components via the following syntax:

```
' Visual Basic
Imports System.ComponentModel
Dim myComponent As New Component()
Dim myContainer As New Container()
myContainer.Add(myComponent)
myContainer.Remove(myComponent)

// C#
using System.ComponentModel;
Component myComponent = new Component();
Container myContainer = new Container();
myContainer.Add(myComponent);
myContainer.Remove(myComponent);
```

A component container will either be an instance of the **Container** class, or an implementation of the **IContainer** interface. **Container** is the reference implementation of this interface.

Specifying a Component Name

You can also specify a name for your component within the container. This name must be unique within the container, and is specified using the **Add** method.

```
' Visual Basic
Dim myComponent As New Component()
Dim myContainer As New Container()
MyContainer.Add(myComponent, "ThisComponent")

// C#
Component myComponent = new Component();
Container myContainer = new Container();
myContainer.Add(myComponent, "ThisComponent");
```

Managing Resources and Extending Containers

Containers provide a central means of managing the resources associated with your components. When the **Container.Dispose** method is called, the container automatically calls the **Dispose** method of all the contained components, thereby ensuring that resources are released promptly.

Containers are extensible. You can create your own class that inherits from **System.ComponentModel.Container** that incorporates custom functionality. For example, you might create a container that enforced rules that govern which components may be added to the container, as is shown in the following example:

```
' Visual Basic
Public Class MyContainer
   Inherits Container
   Public Overloads Overrides Sub Add(ByVal component As IComponent)
      ' Checks to see if the component is allowed to join this container.
      If TypeOf component Is Widget Then
         ' Calls the Add method of the base class, and adds the component.
         MyBase.Add(component)
      Else
```

```
         ' If the component is not allowed, an exception is thrown.
         Throw New NonWidgetException()
      End If
   End Sub
End Class

// C#
class MyContainer : Container
{
   public override void Add(IComponent component)
   {
      // Checks to see if the component is allowed to join this container.
      if (component is Widget)
      {
         base.Add(component);
      }
      else
      {
         throw new NonWidgetException();
      }
   }
}
```

The previous example creates a new container class that is able to enforce a rule about which components are able to join the container. If a component is not of the specified class (in this case, Widget), then an exception is thrown.

When a component is added to a container, the container creates a **Site** for it. This is an implementation of the **ISite** interface that is exposed through the component's **Site** property. Communication by a component with its host container is accomplished via the component's **Site** property. This property represents the logical site of the component, and is hosted by the container. A component that is not contained in a container returns a **null** reference for its **Site** property. The **Site** property allows you to obtain a reference to the container interface through the **Site.Container** property, or a reference to the component interface being hosted through the **Site.Component** property.

```
' Visual Basic
Dim myComponent As New Component
Dim myContainer As New Container
myContainer.Add(myComponent)
Dim myIComponent as IComponent
Dim myIContainer as IContainer
myIComponent = myComponent.Site.Component
myIContainer = myComponent.Site.Container
' These two messages display True.
MessageBox.Show("Are the components equal? " & _
   myComponent.Equals(myIComponent).ToString)
MessageBox.Show("Are the containers equal? " & _
   myContainer.Equals(myIContainer).ToString)

// C#
Component myComponent = new Component();
Container myContainer = new Container();
myContainer.Add(myComponent);
IComponent myIComponent;
IContainer myIContainer;
myIComponent = myComponent.Site.Component;
myIContainer = myComponent.Site.Container;
MessageBox.Show("Are the components equal? " +
   myComponent.Equals(myIComponent).ToString());
MessageBox.Show("Are the containers equal? " +
   myContainer.Equals(myIContainer).ToString());
```

Both of these properties return only the interface associated with these objects, not a reference to the objects themselves. Note that the component also has a **Container** property that returns the same interface as **Site.Container**. This property is provided by through the **Site** and can be thought of as a shortcut.

If you assign a name to your component using the **Add** method, the name can be retrieved via the **Site.Name** property. If the container has a service object associated with it, a reference to that object can be obtained by the component via the

Site.GetService method.

See Also

Communication Between Containers and Components | Creating Component Containers | Extending Component Containers

Creating Component Containers

A component container is a specialized class that acts as a means of organizing and containing components. Through a container you can track your components, communicate with them via the **Site** that hosts the component, and provide a means of common disposal after they are no longer needed. For details, see Containers, Sites, and Components.

To create a component container

1. Declare a variable of the type **System.ComponentModel.Container**, or any class that implements the **IContainer** interface.

```
' Visual Basic
Dim myContainer as System.ComponentModel.Container

//C#
System.ComponentModel.Container myContainer;
```

2. Create an instance of the container class in your variable.

```
' Visual Basic
myContainer = New System.ComponentModel.Container()

// C#
myContainer = new System.ComponentModel.Container();
```

3. Call the **Container.Add** and **Container.Remove** methods to add and remove components to and from your container.

```
' Visual Basic
myContainer.Add(myComponent)
myContainer.Remove(myComponent)

// C#
myContainer.Add(myComponent);
myContainer.Remove(myComponent);
```

See Also

Containers, Sites, and Components | Communication Between Containers and Components | Extending Component Containers | Container Class

Extending Component Containers

Component containers are completely extensible. You can inherit from the **System.ComponentModel.Container** class and add properties or methods, add custom functionality to enforce rules, override base methods, or any other custom functionality you want your container to incorporate. For details on containers and extending containers, see Containers, Sites, and Components.

You extend a **Container** as you would extend any base class. You create a class that inherits the properties of the base class, override any base methods that you want to extend, and add any additional properties or methods that you need. You can then use your new class as you would a standard **Container** and use any of the new functionality you have encoded.

To extend the Container base class

1. Declare a new class that inherits from the **System.ComponentModel.Container** class.

```
' Visual Basic
Public Class myContainer
   Inherits System.ComponentModel.Container
End Class

// C#
class myContainer: System.ComponentModel.Container
{
}
```

2. Override any base-class methods to add additional functionality. The following example shows how to override the **Add** method.

   Note The **Container** actually has two overloads of **Add**, and in this example, you would want to provide an override of each.

```
' Visual Basic
' Because Add is overloaded, this line includes the Overloads keyword.
Public Overloads Overrides Sub Add(ByVal component As _
   System.ComponentModel.IComponent)
   ' Determines if the component can be added to the container.
   If TypeOf component Is Widget Then
      ' Calls the base Add function to add the component.
      MyBase.Add(component)
   Else
      ' Throws an exception.
      Throw New NonWidgetException()
   End If
End Sub

// C#
public override void Add(System.ComponentModel.IComponent component)
{
   if (component is Widget)
      base.Add(component);
   else
   {
      throw(new NonWidgetException());
   }
}
```

3. Add any new properties or methods you want your new container to incorporate. For details, see Class Properties, Fields, and Methods.

See Also

Communication Between Containers and Components

A container is a nexus for communication between the client application and the components it contains. An application may obtain references to components within the container without knowing the actual name or identity of the component. The component may also interact with the client application through the container in a variety of ways.

The container object exposes the components it contains through its **Components** property. This property is an indexed property that returns an **IComponent** reference object. Components are tracked on a first-in, first-out basis, and are accessible through the index using the following syntax:

```
' Visual Basic
Imports System.ComponentModel
Dim MyContainer As Container
Dim xComponent as IComponent
xComponent = MyContainer.Components(0)

// C#
using System.ComponentModel;
Container MyContainer = new Container();
IComponent xComponent;
xComponent = MyContainer.Components[0];
```

Components may be added to a container with or without a name. If you know the name of the component you are referring to, you may also get the reference through the container using that name, as shown in the following example:

```
' Visual Basic
xComponent = MyContainer.Components("myComponent")

// C#
xComponent = MyContainer.Components["myComponent"];
```

Note that the **Components** property returns an **IComponent** reference, and will not allow you to access methods and properties of the component that are not implemented by that interface.

The component communicates with the container primarily through its **Site** property. Through **Site**, the component can obtain reference to the **IContainer** interface implemented by the container, as shown below:

```
' Visual Basic
Dim myComponent As New Component()
Dim myIContainer as IContainer
myIContainer = myComponent.Site.Container

// C#
Component myComponent = new Component();
IContainer myIContainer;
myIContainer = myComponent.Site.Container;
```

The same reference is also returned by the **Component.Container** property. This can be thought of as a shortcut: The reference is still provided through the **Site** object, but not explicitly in this case.

The component can also get services from the container (if they are supplied) by calling IServiceProvider.GetService Method. This method returns an object of the specified type, as shown below:

```
' Visual Basic
Dim myComponent As Component
Dim myWidget As Widget
Dim serviceObject As Object
' This returns an object of type Widget that is supplied by the container.
serviceObject = myComponent.Site.GetService(GetType(Widget))
myWidget = CType(serviceObject, Widget)

// C#
Component myComponent = new Component();
```

```
Widget myWidget;
object serviceObject;
// This returns an object of type Widget that is supplied by the container.
serviceObject = myComponent.Site.GetService(System.Type.GetType("CommunicateCS.Widget"));
myWidget = (Widget)serviceObject;
```

In order to receive an object via **Site.GetService**, it must be implemented in an inherited container class. The **GetService** method of the **Container** class should be overridden and code to supply the service object implemented.

See Also

Containers, Sites, and Components | Creating Component Containers | Extending Component Containers

Components that Contain Other Components

An object model defines a hierarchy of objects that gives structure to an object-based program. By defining the relationships between objects that are part of the program, an object model organizes the objects in a way that makes programming easier.

The public object model of a component is especially important, because all the programmers who employ the component as part of their applications use it.

Users who are not new to object-oriented programming might be used to seeing *class hierarchies*. A class hierarchy describes *inheritance*. That is, it shows how objects are derived from simpler objects, inheriting their behavior. By contrast, object models are hierarchies that describe *containment*. That is, they show how complex objects, such as the **TabControl** contain collections of other objects, such as the **TabPages** collection.

The following topics on object models for components provide information about designing and implementing your object model.

In This Section

Component Object-Model Recommendations
Recommends practices for designing and implementing object models in your component.
Complex Components
Describes what nested classes are and how to implement them.
Properties and Collections in Object Models
Demonstrates how to expose subordinate objects using properties and collections.
Collection Properties
Provides an in-depth discussion of how to use collections to expose collections of subordinate objects.
Walkthrough: Creating Your Own Collection Class
Provides step-by-step instructions for implementing your own strongly-typed collection class.

Related Sections

Component Authoring
Gives advice regarding authoring components and links to many subtopics of interest.

Component Object-Model Recommendations

When designing your component, you should consider how extensive an object model your component needs. Some components — for example, an inherited control — might not need to contain any objects, and would therefore not need an object model. On the other hand, an extremely complex component might need to manage thousand of contained objects. Here are some general recommendations to keep in mind when designing your component object models.

- When possible, expose contained objects as properties. This allows you to control access to the contained objects and validate interactions with contained objects.
- If your object needs to work with and expose an indefinite number of contained objects, implement a collection property. For details, see Collection Properties.
- If your object needs to work with and expose small, defined numbers of contained objects, implement them each as properties.
- If you do not want client applications to be able to use contained objects, set them as **Friend** (**internal** in C#) or **Private**. In these cases, you might want to consider using nested classes. For details, see Nested Classes in Components.
- When implementing a collection property, you must determine how much collection functionality you want to expose to client applications. If you want to expose relatively little functionality, wrap your collection in a property and expose only selected members. If you want to expose a greater amount of functionality, consider creating a custom collection class and exposing the collection directly as a property. For details about these two approaches, see Collection Properties.
- Keep it simple. Generally speaking, a simpler implementation will be faster. If accessing an item involves a series of nested references and function calls, performance will suffer and your component will be more difficult to use. Simple, well-thought-out object models make your component easy for developers and client applications to use alike.

See Also

Components that Contain Other Components | Properties and Collections in Object Models | Collection Properties | Complex Components

Complex Components

Complex components consist of a lot of subordinate objects. For example, a graphics-rendering component might consist of brushes, pens, colors, and a variety of contained objects.

These contained objects may come from different sources. Your component might instantiate and contain classes that are part of the assembly in which it resides. They may call upon other assemblies, such as those in the .NET Framework, to provide the classes that they need to fulfill their functions. Or they may have the source code for these contained objects nested within their own code.

Nested classes are classes that are completely contained within another class declaration. They are useful for providing objects that are required by a particular class to function, but have no stand-alone functionality.

The following topics provide information on designing and implementing nested classes.

In This Section

Nested Classes in Components
  Explains what nested classes are and how they can be useful.
Recommendations on Nested Classes in Components
  Describes best practices for designing and implementing nested classes in your component.
Implementing Nested Classes
  Demonstrates how to implement classes within classes.

Related Sections

Components that Contain Other Components
  Provides background information on creating an object model.
Component Authoring
  Gives advice and provides links to topics about authoring components.

Nested Classes in Components

Many components are fairly simple. They might consist of a single class with a simple object model or no object model. Other components are more complex, and might need to contain and manage a large number of subordinate objects. Nested classes are one way for complex components to contain and manage the objects they need.

A nested class is a class that is fully enclosed within another class declaration. For example, a nested class and an enclosing class might look like the following example:

```
' Visual Basic
' This is the enclosing class, whose class declaration contains the nested
' class.
Public Class EnclosingClass
   ' This is the nested class. Its class declaration is fully contained
   ' within the enclosing class.
   Public Class NestedClass
   ' Insert code to implement NestedClass.
   End Class
' Insert code to implement EnclosingClass.
End Class

// C#
// This is the enclosing class, whose class declaration contains the
// nested class.
public class EnclosingClass
{
   // This is the nested class. Its class declaration is fully contained
   // within the enclosing class.
   public class NestedClass
   {
      // Insert code to implement NestedClass.
   }
   // Insert code to implement EnclosingClass.
}
```

In this example, the class declaration for NestedClass is fully contained by the class declaration of EnclosingClass.

As a result of being contained within the enclosing class, the nested class gains a certain level of protection. Unless you use the **Imports** (**using** in C#) statement, all references to the nested class must be qualified with the name of the containing class. For example, to instantiate the nested class in the previous example, you would have to use the following syntax:

```
' Visual Basic
Dim aClass as New EnclosingClass.NestedClass()

// C#
EnclosingClass.NestedClass aClass = new EnclosingClass.NestedClass();
```

The access level of the nested class is implicitly at least the access level of the enclosing class. Even if the nested class is **Public**, if the enclosing class is **Friend** (**internal** in C#), then only members of the assembly will be able to access the nested class, and if the enclosing class is **Private**, then the nested class will be unavailable to all callers except the enclosing class.

Assuming a **Public** enclosing class, the access level of the nested classes pretty much follows the same rules as for access of unnested classes. Friend classes are available to members of the assembly, but not external clients. **Private** classes are available to the enclosing class, other nested classes within the enclosing class, and any classes nested within other nested classes.

Nested classes are useful when an object will logically contain subordinate objects, but other objects would have use for those objects. An example might be a Wheel class. This could be a class that clients could create and use wherever a wheel might be needed in their application. But except in the most primitive implementations, a wheel is not just a single object, but is composed of several subordinate objects, each of which build the wheel. A wheel might have a Rim object, Tire object, Spoke objects and other objects, without which the wheel could not function. But the average user would have no need to create a spoke, or a rim, or a bearing — all he's interested in is the wheel.

In a case like this, it makes sense for the Wheel class to contain the implementation for all of its subordinate classes. This way, the wheel can create and manage any contained objects it may need while conveniently hiding the details of this implementation

from the client. Those subordinate objects the client might reasonably need to have contact with now and again (for example, a Tire object) can be exposed as part of the public object model, and those that a client should never see (for example, a Bearings collection) could be declared private and hidden.

See Also

Recommendations on Nested Classes in Components | Implementing Nested Classes |
Components that Contain Other Components

Recommendations on Nested Classes in Components

The use of nested classes in your components is a powerful and convenient way to supply subordinate objects that your component might use. Nested classes allow your component to create and use objects without making them accessible or creatable by external users. Overuse of nested classes, however, can make your components difficult to work with and maintain. Here are some design recommendations that will help you to use nested classes in your components.

- If your class is logically contained by another class, but has no stand-alone significance, implement it as a nested class. An example of this might be a Bearing class nested in a Wheel class. The Wheel object might need Bearing objects, but a client application would never have a need to create Bearing objects independent of a Wheel object.
- If members of your class need to access private member variables of the object containing it, you can implement it as a nested class. For example, a Wheel object might have a private Radius field. If a Spoke object needs to access the Radius field, it can always have that access if it is implemented as a nested class.
- If a class needs to implement an interface for interaction with other components, you can create an implementation of that interface as a nested class and expose an instance of the class as an interface property.
- If your component will need to use one of several different subtypes of a contained object, but the subtype will not be known until run time, you can implement each subtype as a nested class as opposed to implementing all possible subtypes within a single class. When used with Just-In-Time compilation, nesting classes improves program performance because only the required type is compiled.
- If other classes will reasonably need access, or will want to collect or contain your class, do not implement it as a nested class. A class that is accessed by several other classes should stand alone.
- In general, do not implement public nested classes. Nested classes should primarily be for the internal use of the containing class. If you do implement a public nested class, it should be a class that logically belongs to the containing type, and is used infrequently. An exception would be if an instance of the nested class was exposed as a property on the containing class. For instance, a Wheel object might have a SpokesCollection nested class, an instance of which might be exposed as the SpokesCollection property.

See Also

Complex Components | Nested Classes in Components | Implementing Nested Classes

Implementing Nested Classes

If you are developing a complex component, you may need to implement nested classes. Nested classes are classes where the class declaration is fully contained within the class declaration of another class, and are useful for developing object models in your component. For more information on how they can be used, see Nested Classes in Components and Recommendations on Nested Classes in Components.

Implementing nested classes is a trivial matter. Once the class declaration is fully enclosed within another class declaration, the class is considered nested and can only be accessed through the containing class.

To implement a nested class within a containing class

- Within the class declaration of the containing class, add the class declaration for the nested class. If you have already written the code for the nested class, you may cut and paste it into the class declaration.

  The following example shows how to implement a nested class:

```
' Visual Basic
' This class contains the nested class.
Public Class TheClass
   ' This class is the nested class.
   Private Class NestedClass
   ' Insert code to implement the nested class.
   End Class
   ' Insert code to implement TheClass.
End Class

// C#
// This class contains the nested class.
public class TheClass
{
   // This is the nested class
   private class NestedClass
   {
      // Insert code to implement NestedClass.
   }
   // Insert code to implement TheClass.
}
```

See Also

Nested Classes in Components | Recommendations on Nested Classes in Components | Complex Components

Properties and Collections in Object Models

Objects in a hierarchy are linked together by object properties — that is, properties that return references to objects. An object that contains other objects will have properties that return either references to the objects themselves, or references to collections of objects.

For example, consider a Bicycle object that represents a bicycle. The Bicycle object would have a Frame property that returned a reference to its Frame object. The Frame object would have a FrontWheel and BackWheel property, each of which would return a Wheel object. The Wheel object would have a Rim property that would return a Rim object, and a Spokes property that would return a Spokes collection object. The Spokes collection would contain the Spoke objects.

It makes sense to organize the object model for a Bicycle object in this way. A Bicycle needs a Frame, a Frame needs Wheels, and Wheels need Spokes. A functioning bicycle cannot exist without these objects, and they are components that make up the bicycle itself. A Bicycle object might interact with a Rider object, but would not contain a Rider object, because a perfectly good, functional bicycle can exist independent of a rider. The object properties represent components that provide a necessary functional part of the overall object.

You may also have dependent objects that are used internally by classes in your component, and which you do not want to provide to users of your component. For example, both the Frame and Wheel objects might have collections of Bearing objects, but there may be no reason to expose the Bearings object, or the collections containing it, for manipulation by client applications. You can implement these as **Private** nested classes. For details, see Nested Classes in Components.

The Simple Way to Link Dependent Objects

Frequently it makes sense for a complex object to have only one instance of a dependent object. For example, a Bicycle object only needs one Frame object. In this case, you can link the Frame object to a simple property of the complex object:

```
' Visual Basic
Private mFrame As New Frame()
Public ReadOnly Property MyFrame() as Frame
   Get
      Return mFrame
   End Get
End Property

// C#
private Frame mFrame = new Frame();
public Frame MyFrame
{
   get
   {
      return mFrame;
   }
}
```

It is important to implement such properties as shown above, using a read-only property procedure, rather than simply declaring a public module-level variable, as shown below.

```
' Visual Basic
Public myFrame As Frame

// C#
public frame myFrame;
```

With the second implementation, a user of your component might set the Frame property to Nothing. If there are no other references to the Frame object, it will be destroyed.

Linking a Fixed Number of Objects

Even when a complex object contains more than one instance of a dependent object, it may make more sense to implement the linkage with properties instead of with a collection. For example, a Bicycle object always has two wheels:

```
' Visual Basic
```

```
Private mwhlFront As New Wheel
Private mwhlRear As New Wheel
Public ReadOnly Property FrontWheel() As Wheel
   Get
      Return mwhlFront
   End Get
End Property
Public ReadOnly Property RearWheel() As Wheel
   Get
      Return mwhlRear
   End Get
End Property

// C#
private Wheel mwhlFront = new Wheel();
private Wheel mwhlRear = new Wheel();
private Frame mFrame = new Frame();
public Wheel FrontWheel
{
   get
   {
      return mwhlFront;
   }
}
public Wheel RearWheel
{
   get
   {
      return mwhlRear;
   }
}
```

Using Collections in Your Object Model

When the relationship between two objects in a hierarchy is such that the first object contains an indeterminate number of the second, the easiest way to implement the link is with a collection. A *collection* is an object that contains a set of related objects.

A property exposes a collection of objects by acting as a wrapper around the collection. When the property is accessed, it returns the appropriate member of the collection by taking an indexer as a parameter, using that indexer to access the collection and find the specified object, then converts it (if necessary) to the appropriate type before returning the object.

The .NET Framework provides several different collection classes that can serve as the basis for an object collection property. For details, see Collection Properties.

See Also

Collection Properties | Component Object-Model Recommendations

Collection Properties

When your object model exposes an indeterminate number of objects of a particular type as a property, you will want to provide a collection as part of the implementation of that property. You can choose how to implement that property depending on the kind of functionality you want your property to support.

The .NET Framework supplies several different types of collection classes you can use to track and manage objects in your object model. Additionally, Visual Basic .NET provides the **Collection** class, which can be used to keep track of objects collections that are exposed as properties. If using C#, you should implement collections in your object model by using one of the collection classes provided in the **System.Collections** namespace.

Some of the collection classes in the System.Collections are briefly described below. You can use these classes to implement your collection properties depending on the requirements of your class. For additional information on these classes, consult the appropriate help topic.

- ArrayList Class provides basic enumerable, indexed list of objects.
- BitArray Class manages a compact array of bits (1 and 0).
- CollectionBase Class provides a base class for collections. It is an abstract (MustInherit) class that must have some functionality implemented. For information on how to implement your own collection class, see Walkthrough: Creating Your Own Collection Class.
- Queue Class tracks objects on a first in-first out basis.
- SortedList Class provides a collection that allows you access either by index or by key.
- Stack Class tracks objects on a first in-last out basis.

Simple Collection Properties

The following example shows how to implement a simple collection property using the ArrayList Class.

```
' Visual Basic
Private spokesCollection As New System.Collections.ArrayList
Public ReadOnly Property Spokes() As System.Collections.ArrayList
   Get
      Return spokesCollection
   End Get
End Property

// C#
System.Collections.ArrayList spokesCollection = new
   System.Collections.ArrayList();
System.Collections.ArrayList Spokes
{
   get
   {
      return spokesCollection;
   }
}
```

Robust Collection Properties

Although the previous example is a very simple and easy way to provide a collection property, it is not very robust. There are several problems with this implementation as a collection property. First, it allows clients of your application to access all the functionality contained by the collection. You may not want other users to be able to remove objects, for instance, or to add new items. Second, it is not strongly typed. Any kind of object may be added to your spokes collection, as shown above.

The most robust way to address these issues is by creating your own custom collection class by inheriting from **System.Collections.CollectionBase**. This approach allows you to create a collection class that is strongly typed and exposes whatever methods you decide. Creating your own collection class allows you to retain the use of **For Each...Next** with your collections. You can create **Public (public)** methods for public use, **Friend (internal)** methods for use by other objects in your assembly, and **Private (private)** methods for the object's own use. Information on how to create a custom collection class can be found in Walkthrough: Creating Your Own Collection Class.

A less robust, but possibly more convenient way to address these issues is by not directly exposing the collection object, but rather by returning members of the collection. This can be accomplished by wrapping the collection object inside the property

call. Using this method, you are able to retain almost all of the functionality of a collection class (with the exception of being able to use **For Each...Next**), and can control access to all of the methods provided by the collection and enforce type safety. The following example shows how to implement a collection property using the SortedList Class This collection property provides access to members of the collection either by index or by key.

```
' Visual Basic
Private spokesCollection As System.Collections.SortedList
' This is the first of two overloads of the property. This overload allows
' the user to retrieve a Spoke object by specifying an index
Public Overloads ReadOnly Property Spokes(ByVal i As Integer) As Spoke
   Get
      ' CType converts the object reference returned by spokesCollection
      ' to a Spoke object.
      Return CType(spokesCollection(i), Spoke)
   End Get
End Property
' This overload allows the user to retrieve a Spoke object by specifying a ' key
Public Overloads ReadOnly Property Spokes(ByVal key As String) As Spoke
   Get
      ' CType converts the object reference returned by spokesCollection
      ' to a Spoke object.
      Return CType(spokesCollection(key), Spoke)
   End Get
End Property

// C#
private System.Collections.SortedList spokesCollection;
// Because properties in C# cannot accept parameters, we will implement
//  these as methods instead. This is the first of two overloads of the
// method. This overload allows the user to retrieve a Spoke object by
// specifying an index
public Spoke GetSpokeAt(int i)
{
   // returns the appropriate member of the collection, converted to the
   // appropriate type
   return (Spoke)spokesCollection[i];
}

public Spoke GetSpokeAt(string key)
{
   // returns the appropriate member of the collection, converted to the
   // appropriate type
   return (Spoke)spokesCollection[key];
}
```

Exposing Methods in Collection Properties

In the previous example, the methods implemented by the collection itself are inaccessible to external users; only the containing class can **Add** or **Remove** members, or use any of the other collection methods.

You might want to expose one or more of these methods to external users. This is easily accomplished by wrapping the call to the collection method inside a new method. The following example shows how to wrap the spokesCollection object inside a public method, which allows clients to add new Spoke objects to the collection.

```
' Visual Basic
' This example assumes all the code from the previous example is in the same class.
Public Sub AddSpoke(ByVal Key as String)
   Dim aSpoke as New Spoke()
   SpokesCollection.Add(Key, aSpoke)
End Sub
```

```
// C#
public void AddSpoke(string Key)
{
   Spoke aSpoke = new Spoke();
```

```
    spokesCollection.Add(Key, aSpoke);
}
```

See Also

Walkthrough: Creating Your Own Collection Class | System.Collections Namespace | Component Object-Model Recommendations | Properties and Collections in Object models

Walkthrough: Creating Your Own Collection Class

You can create your own collection classes by inheriting from one of the many .NET Framework collection classes and adding code to implement your own custom functionality. In this topic, you will use inheritance to create a simple, strongly typed collection inherited from **CollectionBase**.

The .NET Framework provides several collection-type classes in the **System.Collections** namespace. Some, such as **Stack**, **Queue**, and **Dictionary**, are specialized classes that have been implemented to fulfill specific roles. Others, such as **CollectionBase** and **DictionaryBase**, are **MustInherit (abstract)** classes that have some basic functionality in place, but which leave much of the implementation to the developer.

The **CollectionBase** class already has implementations for the **Clear** method and the **Count** property, and it maintains a **Protected** property called **List**, which it uses for the internal storage and organization. Other methods, such as **Add** and **Remove**, as well as the Item property, require implementation.

In this walkthrough, you use the **CollectionBase** class to create a class called WidgetCollection. This is a collection that accepts only widgets, and exposes its members as Widget types, instead of accepting objects and exposing members as **Object** type. You then implement methods to add widgets to your collection and to remove the widget at a specific index, and you also implement an **Item** property that returns the widget object at the appropriate index.

Creating the Classes

The first step is to create a Widget class to put into your WidgetCollection.

To create the Widget class

1. Open a new Windows Application and name it **WidgetProject**. From the Project menu, choose **Add Class**. In the Add New Item dialog box, name your class **Widget.vb** for Visual Basic and **Widget.cs** for Visual C#.

   The Code Editor for the Widget class opens.

2. Add a public Name variable to your Widget class. Your code should look similar to the following:

```
' Visual Basic
Public Class Widget
   Public Name as String
End Class

// C#
public class Widget
   {
      public string Name;
      public Widget()
      {
      }
   }
```

3. From the File menu, choose **Save All**.

You have now created a Widget class for use with the WidgetCollection class you create next. The Widget class is a simple class, which serves to illustrate how your strongly typed collection will work. The next step is to create the WidgetCollection class.

To create the WidgetCollection class

1. From the Project menu, choose **Add Class**. In the Add New Item dialog box, name your class **WidgetCollection.vb** for Visual Basic and **WidgetCollection.cs** for Visual C#. The code editor for the WidgetCollection class opens.
2. In the code editor, add code to cause the class to inherit from **CollectionBase**, as in this example:

```
' Visual Basic
Public Class WidgetCollection
   Inherits System.Collections.CollectionBase
End Class

// C#
```

```
public class WidgetCollection : System.Collections.CollectionBase
   {
   }
```

3. From the **File** menu, choose **Save All**.

You have now created your WidgetCollection class. Because this class inherits from **CollectionBase**, it already has much of the collection functionality implemented. There is a **Clear** method to clear the collection and a **Count** property that keeps track of the number of current members. Additionally, there is a **Protected** object called **List**, which you use to keep track of your widgets. But there is no **Add** method, no **Remove** method, and no **Item** property. The implementation of these is left up to the developer.

Implementing the Add and Remove Methods

You will now implement the **Add** method so that WidgetCollection will add only Widget objects.

To implement the Add method

1. Add the following code to your WidgetCollection class, beneath the **class** statement in C#, or the **Inherits** statement in Visual Basic:

```
' Restricts to Widget types, items that can be added to the collection.
Public Sub Add(ByVal awidget As Widget)
   ' Invokes Add method of the List object to add a widget.
   List.Add(aWidget)
End Sub

// C#
// Restricts to Widget types, items that can be added to the collection
public void Add(Widget aWidget)
{
   List.Add(aWidget);
}
```

   In this method, you restrict to **Widget** types the items that can be added to the **List** object through the argument taken by the **Add** method. Though the **List** object can accept any type of object, this method prohibits any objects but Widget types from being added, and acts as a "wrapper" around the **List** object.

2. From the **File** menu, choose **Save All.**

You now have a way to add widgets to your collection. You must now implement a way to remove them. You will do this in a manner similar to the way you created the **Add** method, by creating a **Remove** method that accepts an index as an argument and in turn calls the **List.RemoveAt** method.

To implement the Remove method

1. Beneath the `Add` method, add the following code:

```
' Visual Basic
Public Sub Remove(ByVal index as Integer)
   ' Check to see if there is a widget at the supplied index.
   If index > Count - 1 Or index < 0 Then
      ' If no widget exists, a messagebox is shown and the operation is
      ' cancelled.
      System.Windows.Forms.MessageBox.Show("Index not valid!")
   Else
   ' Invokes the RemoveAt method of the List object.
      List.RemoveAt(index)
   End If
End Sub

// C#
public void Remove(int index)
{
```

```
   // Check to see if there is a widget at the supplied index.
   if (index > Count - 1 || index < 0)
      // If no widget exists, a messagebox is shown and the operation
      // is cancelled.
      {
         System.Windows.Forms.MessageBox.Show("Index not valid!");
      }
   else
      {
         List.RemoveAt(index);
      }
}
```

This method accepts an integer value for the index argument. If the value is valid, it is passed to the **RemoveAt** method of the **List** object, thereby removing the item at the indicated index from the collection.

2. From the **File** menu, choose **Save All.**

Implementing the Item Property

You need to implement one final piece in order to complete basic collection functionality, and that is the **Item** property. The **Item** property allows you to obtain a reference to an object in your collection by referring to the index. Since you already have an **Add** method to add members to the collection, **Item** will be a **ReadOnly** property in this demonstration, though it need not be in other contexts. Because C# does not allow properties to have parameters, you will implement **Item** as a method if you are using C#.

To implement the Item property

1. Beneath the `Sub Remove` method, add the following code:

```
' This line declares the Item property as ReadOnly, and
' declares that it will return a Widget object.
Public ReadOnly Property Item(ByVal index as Integer) As Widget
   Get
      ' The appropriate item is retrieved from the List object and
      ' explicitly cast to the Widget type, then returned to the
      ' caller.
      Return CType(List.Item(index), Widget)
   End Get
End Property

// C#
public Widget Item(int Index)
{
   // The appropriate item is retrieved from the List object and
   // explicitly cast to the Widget type, then returned to the
   // caller.
   return (Widget) List[Index];
}
```

In collections, the syntax `Collection.Item(0)` and `Collection(0)` are frequently interchangable. If you want to your collection to support this syntax, you should make the Item property the Default property (for Visual Basic) or implement an indexer (in C#). For more information, see Default Properties for Your Components.

2. From the File menu, choose **Save All**.

You have now implemented basic collection functionality in your class. You have implemented methods to add and remove widgets from the collection, and you have implemented a property that returns a Widget reference to the appropriate item. In this next section, you will test your WidgetCollection class.

Testing the Project

Now that your WidgetCollection class is complete, it is time to test the functionality. To do this, you will create a simple Windows application that will add, remove, and loop through the widgets in your collection.

To create the test project

1. In Solution Explorer, right-click **Form1**, then click **View Designer** on the shortcut menu.

   The designer for Form1 opens.

2. Using the Toolbox, add two **Label** controls, two **TextBox** controls, and three **Button** controls to Form1.
3. Set the properties as follows:

Control	Property	Value
Label1	**Text**	"Widget Name"
Label2	**Text**	"Widget Index"
TextBox1	**Text**	""
TextBox2	**Text**	""
Button1	**Text**	"Add a Widget"
Button2	**Text**	"Remove a Widget"
Button3	**Text**	"Review Widgets"

   In the design of your form, Label1 should label TextBox1, and Label2 should Label TextBox2.

4. In the Form Designer, right-click WidgetTest and choose **View Code**.

   The Code Editor for WidgetTest opens.

5. In the Code Editor, locate the line `Inherits` line (Visual Basic) or the `class` line (Visual C#). Immediately beneath it, type the following code:

```
' Visual Basic
' Creates a WidgetCollection object.
Dim myWidgetCollection as New WidgetCollection()

// C#
WidgetCollection myWidgetCollection = new WidgetCollection();
```

6. In the designer, double-click button1, which now reads "Add a Widget".
7. In the **button1_Click** event handler, add code to add a new widget to the collection and set the name of this widget to the value of `textBox1`.

```
' Visual Basic
If textbox1.Text = "" Then
   MessageBox.Show("Please name your widget.")
Else
   ' Declares and instantiates a new widget.
   Dim aWidget as New Widget()
   ' Sets the Name field of the new widget.
   aWidget.Name = textBox1.Text
   ' Adds the new widget to the collection.
   myWidgetCollection.Add(aWidget)
   ' Updates textBox2 with the index of the widget that was added.
   ' The index of a zero-based collection is Count property minus 1.
   textBox2.Text = (myWidgetCollection.Count - 1).ToString()
End If

// C#
if (textBox1.Text == "")
   MessageBox.Show("Please name your widget.");
else
{
   // Declares and instantiates a new widget.
   Widget aWidget = new Widget();
```

```
    aWidget.Name = textBox1.Text;
    // Adds the new widget to the collection
    myWidgetCollection.Add(aWidget);
    // Updates textbox2 with the index of the widget that was added.
    // The index of a zero-based collection is Count property minus 1
    textBox2.Text = (myWidgetCollection.Count -1).ToString();
}
```

8. In the **button2_Click** event handler, add code to remove the widget at the index specified in `textBox2`.

```
' Visual Basic
Try
   myWidgetCollection.Remove(CInt(textBox2.Text))
Catch ex as System.InvalidCastException
   MessageBox.Show("You have attempted to perform an invalid cast")
End Try
// C#
try
{
   myWidgetCollection.Remove(int.Parse(textBox2.Text));
}
catch (System.InvaldiCastException ex)
{
   MessageBox.Show("You have attempted to perform an invalid cast");
}
```

   Note that you while you must provide error handling to catch any **System.InvalidCastException** that might be thrown as the result of an invalid character in the textbox, there is no need to check if a valid value for the index exists in textBox2, because the **WidgetCollection.Remove** method checks this automatically.

9. In the **button3_Click** event handler, write code to cycle through all members of the collection and sequentially display their names in a message box.

```
' Visual Basic
Dim counter as Integer
For counter = 0 to myWidgetCollection.Count - 1
   MessageBox.Show(myWidgetCollection.Item(counter).Name)
Next

// C#
int counter;
for(counter = 0; counter <= myWidgetCollection.Count - 1; counter++)
{
   MessageBox.Show(myWidgetCollection.Item(counter).Name);
}
```

   You can also use **For Each...Next** to cycle through the members of your collection.

10. In Solution Explorer, right-click **WidgetProject** and choose **Properties**. The WidgetProject Property Page opens. Under **Startup Object**, choose **WidgetProject.Form1** from the menu and click **OK** to close the page.

You will now run your test application. You will test the Add method, the Remove method, and the retrieval of Widget objects from the collection.

To test the collection

1. In the Widget Name text box, type **Widget0** and click the Add a Widget button. Repeat this two times, substituting **Widget1** and **Widget2** for **Widget0**.

   Three widgets have now been added to the collection.

2. Click the **Review Widgets** button.

The name of each widget successively appears in a message box.

3. In the Widget Index text box, type **1** and click the **Remove a Widget** button.

   The widget at index 1 is removed. The index is updated dynamically, so that the widget that previously occupied index 2, now occupies index 1.

4. Click the **Review Widgets** button.

   The name of each Widget is again successively displayed in a message box. Note that "Widget1" no longer displays, as it has been removed from the collection.

In this walkthrough, you learned how to implement basic collection functionality by inheriting from the **System.Collections.CollectionBase** class. You learned how to strongly type your collection, and how to implement the **Add** and **Remove** methods and the **Item** property. For more information on the base classes available to you for the creation of collection classes, see System.Collections Namespace.

See Also

System.Collections Namespace | Object Management with Collections | Using Collections In Classes

Exceptions in Components

If you design and run components, they will occasionally cause or encounter errors. When a client program calls a method of your component, any exceptions you throw and/or decide not to handle will be passed along to the client program.

Throwing exceptions to the client is appropriate behavior for components. A well-behaved component does not intrude on the client application's user interface by displaying message boxes containing error text.

Users of an application may be unaware that your component is part of the application they are using. Seeing error messages from a program unknown to them will not help users solve the problem.

The following topics discuss the strategies and mechanics of returning error information to the client application, for the application's developer to deal with as she chooses.

In This Section

Best Practices for Throwing Exceptions from Components
: Describes when to throw exceptions, when not to throw exceptions, what kind of exceptions to throw, and what information to include with the exception.

Throwing Exceptions from Components
: Details the mechanics of throwing exceptions.

Recommendations on Handling Exceptions in Components
: Provides recommendations regarding what to do when an exception occurs in your component class code.

Handling Exceptions in Your Component
: Describes the mechanics of handling exceptions and including additional information when rethrowing exceptions.

Related Sections

Exception Handling
: Provides background on how exceptions are handled in Visual Basic.

Exception Handling Statements
: Provides background on how exceptions are handled in Visual C#.

Best Practices for Throwing Exceptions from Components

When an application calls a method of your component, the method can provide error information by throwing an exception. The client application can implement an exception handler to trap errors that may be thrown by the method.

There are a number of programming tradeoffs to consider when you select an error generation strategy for your component, but the most important consideration should be the convenience of the developer who will use your component. If it is possible for your component to handle an exception internally, it should. The client application should receive exceptions from your component only when additional intervention is required.

When you raise exceptions, the developer who calls your method has the choice of handling the exception or allowing it to be thrown further up the call stack.

Be Consistent

Whatever exceptions you throw from your component should be logically consistent. If you decide to create custom exceptions to report errors unique to your component, they should be limited to reporting that particular error. Do not use a general exception when a more specific one can be used. The purpose of raising an exception is to provide the client application with as much information about how to correct the error and keep the program running smoothly as possible.

If you create and throw custom exceptions in your component, they should be well documented so that developers using your component will best be able to handle them. Your documentation should include a description of the exception, the conditions under which it is likely to occur, suggested ways for resolving the error condition, and any error codes or additional information that the exception returns.

Exceptions Are For Exceptional Circumstances

Do not throw exceptions as a means of communicating between components-use events for that. Exceptions should not be thrown in cases where completely expected results occur. For example, a component that reads text files should not throw an exception when it reaches the end of the file. This is a case where an event would be appropriate. By limiting throwing exceptions to exceptional cases, you will help developers to debug their application rapidly by not having to wade through numerous 'expected' exceptions.

COM-Visible Components and Exceptions

If your component is to be called by a COM client, there are additional considerations to be taken into account. All exceptions thrown from a component to a COM client must provide an error code. With standard exceptions, this code is provided "behind the scenes" via the **HResult** property. If you are creating and throwing custom exceptions, you must set an appropriate **HResult** value. If your custom exception returns an error condition that is similar to a standard error condition, then the **HResult** value for that error condition may be used. Otherwise, you can set the value however you choose. In this case, it is important to document all error codes that might be passed via **HResult** so the developer using your component can implement appropriate responses. For details, see HRESULTs and Exceptions.

Unstructured Error Handling (Visual Basic Only)

Visual Basic .NET also supports unstructured error handling using the **On Error Goto** syntax. While this kind of error handling will continue to function in your components, it is not recommended that it be implemented. Code written using this method of error handling is difficult to debug and maintain, and it can degrade application performance. Structured error handling, using the **Try...Catch...Finally** block, results in code that is easier to maintain, more robust, and more flexible in the way that errors can be handled.

See Also

Throwing Exceptions from Components | Exception.HResult Property | Exception Handling | Try...Catch...Finally Statements | Exception Handling Statements

Throwing Exceptions from Components

Any exception that occurs in your component but is not handled will automatically be thrown to the client application. For example, the following will throw an **OverflowException**.

> **C# Note** In order to throw the overflow exception, you must set Check for overflow underflow to true in the Build property page in the Configuration Properties folder of the Project Properties dialog box.

```
' Visual Basic
Dim XX As Integer = 100000
Dim YY As Integer = 100000
Dim ZZ As Integer
' The resulting value is to large to be contained by an integer, thus
' the overflow error is thrown.
ZZ = XX * YY

// C#
int number1 = 100000;
int number2 = 100000;
int product;

// The resulting value is to large to be contained by an integer,
// thus the overflow error is thrown.
product = number1 * number2;
```

If this exception is not handled in the component that generates it, the exception will be bubbled up the call stack until it either finds an appropriate error handler, or the common language runtime default error handler handles it. If you want to pass standard runtime exceptions to the client application, no additional coding is necessary. It will happen automatically.

In areas of code where you suspect an exception might be thrown, you will probably want to provide additional information to the developers using your component.

The following example shows how to catch the exception as it is thrown, and pass it to the client application along with additional information in the message property:

```
' Visual Basic
' The code in the Try block is the code expected to cause an error.
Dim XX As New Integer = 100000
Dim YY As New Integer = 100000
Dim ZZ As Integer
Try
   ' The resulting value is to large to be contained by an integer, thus
   ' the overflow error is thrown.
   ZZ = XX * YY
' If an overflow exception is generated, it will be caught here.
Catch e as OverflowException
   ' Throws a new exception that provides information in the message
   ' string and provides the original exception as the InnerException, e.
   Throw New OverflowException( _
   "The value for the product was larger than an integer could hold", e)
' Catches any other kind of exception that might be generated.
Catch e as Exception
   Throw e
End Try

// C#
// The code in the try block is the code expected to cause an error.
int number1 = 100000;
int number2 = 100000;
int product;
try
{
   product = number1 * number2;
}
// If an overflow exception is generated, it will be caught here.
```

```
catch (OverflowException e)
{
   // Thows a new exception that provides information in the message
   // string and sends on the original exception e.
   throw new OverflowException(
      "The value for the product is larger than an integer can hold.", e);
}
// Catches any other kind of exception that might be generated.
catch (Exception e)
{
   throw e;
}
```

In the preceding example, an exception is thrown when the overflow condition is generated. This exception is caught by the **Catch** block, which then throws a new exception that contains an informative message and the original exception in the **InnerException** property. This new exception is then bubbled up the call stack until it is handled.

To provide additional information when throwing exceptions

1. Determine which parts of your code are likely to throw exceptions, and what exceptions might be thrown.
2. Enclose these areas of code within a **Try...End Try** block.
3. Use the **Catch** statement to catch specific exceptions. For example:

```
' Visual Basic
Catch e as OverflowException
   ' Handling code omitted.
Catch e as NullReferenceException
   ' Handling code omitted.

// C#
catch (OverflowException e)
{
   // Handling code omitted.
}
catch (NullReferenceException e)
{
   // Handling code omitted.
}
```

4. Within the **Catch** block, **Throw** a new exception that provides information in the message property and wraps the original exception in the **InnerException** property. An example is shown:

```
' Visual Basic
Catch e as OverflowException
   Throw New OverflowException( _
      "An overflow occurred in the Widget component.", e)

// C#
catch (OverflowException e)
{
   throw new OverflowException(
      "An overflow occurred in the Widget component.", e);
}
```

Custom Exceptions

If you want to throw a custom exception to the client application, you have that opportunity. You must first create a new exception that inherits from the **Exception** class or a derived class. You must then determine the conditions under which this exception will be thrown, and write the appropriate code to throw the exception.

To create and throw a custom exception

1. Create a custom exception class. This class should inherit **System.Exception** or a class derived from **System.Exception**. An example is shown below.

```
' Visual Basic
Public Class SpeedOfLightException
   Inherits System.Exception
   ' Creates a constructor for the exception that allows you to set the
   ' message property when thrown.
   Public Sub New(ByVal Message As String)
      MyBase.New(Message)
   End Sub
End Class

// C#
public class SpeedOfLightException:System.Exception
{
   //Creates a constructor for the exception that allows you to set the
   // message property when thrown.
   public void SpeedOfLightException(string Message):base(Message)
   {
   }
}
```

2. Decide how you intend to use your exception, and when you want to throw it. You should examine your code and determine those parts where error situations are possible, but a standard exception will not be thrown.
3. When you have determined how your custom exception is to be used, test for error conditions and throw your exception when appropriate. For example:

```
' Visual Basic
Public Sub Accelerate(ByVal finalVelocity as Integer)
   If finalVelocity > cnstSpeedOfLight then
      Throw New SpeedOfLightException("You cannot exceed light speed!")
   End If
   ' (Additional code omitted)
End Sub

// C#
public void Accelerate(int finalVelocity)
{
   if (finalVelocity > cnstSpeedOfLight)
   {
     throw new
        SpeedOfLightException("You cannot exceed light speed!");
   }
   // Additional code omitted.
}
```

   The preceding code receives a value as an argument. It then checks to see if this argument is greater than a constant, and if so throws a SpeedOfLightException. This exception is bubbled up to the calling method, which will then decide how to deal with it.

4. If your component will be exposed to COM, you must determine and set an appropriate **Hresult** value for your exception. For details, see Exception.HResult Property.

See Also

Best Practices for Throwing Exceptions from Components

Recommendations on Handling Exceptions in Components

When authoring a component, you should be prepared to handle different kinds of exceptions.

- Exceptions you throw from your component class or its constituent objects, which you handle internally.
- Exceptions you throw from your component class or its constituent objects, which you want to pass along to the client to indicate an error condition in your component.
- Exceptions thrown by a component you are using, but did not write yourself. You may handle these exceptions internally, pass them through to the client, or catch them and then throw an exception of your own.

Handling Errors Internally

Handling the first of these three exception types is no different for a component than for any other application developed using Visual Basic or Visual C#.

Throwing Exceptions to the Caller

Frequently your properties and methods will include code to handle internal errors. Such code may from time to time contain cases in which no sensible internal response is possible. In such cases, it's reasonable for the property or method to fail and return an error to the client. In cases where your component cannot handle the exception that has occurred, you might want to provide additional information about the exception to aid the developer in debugging his code. You might even want to throw a custom exception that provides the developer with very detailed information. These situations are discussed in Throwing Exceptions from Components.

Handling Exceptions from Another Component

If your component calls methods of another component, you must handle exceptions that component may throw. You should be able to obtain a list of these exceptions from the second component's documentation.

It's important to handle such exceptions within your component, and not return them to your caller. The caller was written to use your component, and to handle your component's exceptions. The user or programmer who writes it may have no knowledge of the constituent components you are using.

Encapsulation

The idea that a client application should receive exceptions only from the components it calls directly reflects the object-oriented concept of *encapsulation*. Although your component may use other components, the client application is ignorant of those objects.

In most cases, the best course of action will be for your component to handle any exceptions it receives itself, without breaking or notifying the client application that an exception has occurred. A well-written component will take into account exceptions that may be thrown by secondary components it uses.

In some cases, it may make sense to report the exception to the client application. Cases where additional input from the client is needed will require intervention by the client and cannot be handled by your component alone. In these cases, you should not allow the original exception to bubble up to the client application. Rather, you have two choices:

- Catch the exception thrown in your secondary component and throw a new exception of the same type to the client application.
- Catch the exception thrown in your secondary component and throw a new exception of a custom type, specifically tailored to your situation.

In the first case, you can throw a new exception that originates from your component. You can add a message string to provide the developer with additional information, and you can package all the original exception information in the InnerException property of the new exception. This generates an exception that originates from your component and keeps the client application unaware of any secondary components in use, but contains all of the original information if needed. An example is shown below:

```
' Visual Basic
Try
   ' Calls a method in a secondary component. This call is enclosed in a
   ' Try block in case an exception is thrown by SecondaryComponent.
   SecondaryComponent.CallMethod()
Catch e as OverflowException
```

```
    ' An overflow exception is caught and a new overflow exception is
    ' thrown to the client application. This new excpetion includes the
    ' original exception as the InnerException, and provides an informative
    ' message.
    Throw New OverflowException("An overflow occurred in the xyz method", _
       e)
End Try

// C#
try
{
   // Calls a method in a secondary component. This call is
   // enclosed in a try block in case an exception is thrown
   // by the secondary component.
  secondaryComponent.CallMethod();
}
catch (OverflowException e)
{
   // An overflow exception is caught and a new overflow exception
   // is thrown to the client application. This new exception
   // includes the original exception as the InnerException, and
   // provides an informative message.
   throw new OverflowException("An overflow occurred in the xy method.", _
      e);
}
```

If you need to provide more detailed information to the client application, you might want to create a custom exception. This allows you to pass the original exception to the client application as the **InnerException**, but also allows you to provide information in the form of additional properties specific to your case. Creating and throwing of custom exceptions is discussed in Throwing Exceptions from Components. If you do decide to throw custom exceptions from your component to the client application, it is important to remember to thoroughly document any exceptions your component might throw, so that future developers will be able to respond appropriately to them.

See Also

Throwing Exceptions from Components | Handling Exceptions in Your Component

Handling Exceptions in Your Component

If your component uses additional components, it will likely receive exceptions from them from time to time. Good coding practice dictates that you anticipate potential problems and write error-handling code to deal with predicted situations. The .NET Framework provides structured error handling with the **Try...Catch...Finally** statements. Using this method of error handling, it is possible to catch multiple exceptions and provide different error handling for each using multiple **Catch** statements. The **Finally** statement allows you to specify code that will be executed before the error-handling block is exited.

Occasionally, it will be necessary to return an exception from a secondary component to the client application, either by wrapping it in a new exception, or by throwing a custom exception. Recommendations for these practices are discussed in Recommendations on Handling Exceptions in Components.

When writing exception-handling code for your component, try to anticipate what exceptions might occur and write appropriate code to deal with them. In the event that an exception cannot be handled, you should determine what code must be executed before passing on the exception (such as closing a file, saving changes, and so on) and write the appropriate code to the **Finally** block.

To handle exceptions in your component

1. Enclose calls to secondary components within the context of a **Try** block.
2. Determine what exceptions are likely to be thrown by this call, and devise protocols for dealing with them.
3. Enclose these protocols within separate **Catch** blocks.
4. Create an additional **Catch** block to handle any unanticipated exceptions, and pass them on to the client application.
5. Determine what code must be run, even in the event of an exception, and enclose that in a **Finally** block. This code will execute whether an exception is thrown or not.

The following example shows how to create a simple exception handler:

```
' Visual Basic
Public Sub ExceptionDemo
   Try
      ' Calls a method in a secondary component.
      SecondaryComponent.MyMethod()
   Catch e as ArgumentOutOfRangeException
      ' Insert code to fix this specific exception instance.
   Catch e as UnauthorizedAccessException
      ' Insert code to fix this specific exception instance.
   ' This block will catch any other exceptions that are thrown and
   ' package them inside a new exception which will be thrown to the
   ' client application.
   Catch e as Exception
      Throw New Exception("An unhandled exception occured", e)
   ' The Finally block specifies code that will be executed in the event
   ' of an exception.
   Finally
      ' Insert code to save data, release references, and so on.
   End Try
End Sub

// C#
public void ExceptionDemo(int a)
{
   // C#
   try
   {
      secondaryComponent.MyMethod();
   }
   catch (ArgumentOutOfRangeException e)
   {
      // Code to fix this specific exception.
   }
   catch (UnauthorizedAccessException e)
   {
      // Code to fix this specific exception.
   }
```

```
    // This block will catch any other exceptions that are thrown
    // and package them inside a new exception which will be thrown
    // to the client application.
    catch (Exception e)
    {
       throw new Exception("An unhandled exception occurred.", e);
    }
    // The finally block specifies code that will be executed in the
    // event of an exception
    finally
    {
       // Insert code to save data, release references, and so on.
    }
 }
```

See Also

Structured Exception Handling | Recommendations on Handling Exceptions in Components | Exception Handling Statements

Code Security in Components

Code security is a topic of great importance and concern. Keeping your code secure from unauthorized or malicious use while simultaneously allowing authorized users to use your components should be of the highest priority to you the component developer. The following topics provide information on how to write safe and secure components while allowing authorized users to experience your components to the fullest.

In This Section

Adding Security Attributes to Components
Demonstrates how to add security attributes to your components.
Adding Imperative Security Checks to Components
Describes how to use imperative security checks to protect sensitive lines of code from unauthorized use.
Code Security and Signing in Components
Provides conceptual information on code security and decribes the .NET Framework approach to securing your components.

Related Sections

Component Authoring
Provides a high-level overview of component authoring.
Key Security Concepts
Discusses security in the .NET Framework.

Adding Security Attributes to Components

Permissions are the way that your component interacts with the .NET Framework security policy. Permissions can allow your component to perform a variety of potentially unsafe actions that might otherwise be prohibited by the security policy. You can request permissions for your assembly through the use of security attributes. Security attributes modify code members, such as methods or classes. At compile time, security attributes are emitted into metadata that is then stored in the assembly manifest. The assembly manifest is then examined by the common language runtime, and the requested permissions are examined and applied against the local computer security policy. If policy allows, the requested permissions are granted and the assembly is allowed to run.

Security requests are made through the **SecurityAction** parameter of the attribute. The following example shows how:

```
' Visual Basic
Imports System.Security.Permissions
<Assembly: FileIOPermission(SecurityAction.RequestMinimum)>

// C#
using System.Security.Permissions;
[assembly: FileIOPermission(SecurityAction.RequestMinimum)]
```

You can specify different types of permission requests using the **SecurityAction** parameter.

> **Note** Regardless of where the attribute is attached in code, it is effective for the entire assembly. Either the permission will be granted to the whole assembly or to none of it.

You can only be granted the security permissions you request. Therefore, if you only request **FileIOPermission**, for example, you will be denied all other permissions. This is true even if you would have ordinarily been granted other permissions by the default setting of the local security policy.

Security attributes can also be used to protect individual classes or methods. You can use the **SecurityAction** parameter and the **Demand** method to require a particular permission level before allowing access to a class or method. For example, suppose you have a class that deletes files when no longer needed. Uncontrolled access to the class would leave you vulnerable to attacks by malicious users. By demanding permission to use the class, you ensure that all users will have the appropriate level of permission to use potentially dangerous code. The effect is similar to making an imperative security check, but the entire code member is protected, not just the blocks of code. For details on imperative security checks, see
Adding Imperative Security Checks to Components.

To add a permission request to your component

1. Determine the type and level of permission you are requesting.
2. Attach the security attribute to the declaration of your component. Use the appropriate SecurityAction Enumeration to request the desired level of permission.

SecurityAction	Permission level
RequestMinimum	The requested permission is required for the assembly to run, and if not granted, the assembly will not be started
RequestOptional	The assembly will still be run if the requested permission is not granted
RequestRefuse	The assembly will be denied this permission

The following example shows how to attach the **RequestMinimum** permission level:

```
' Visual Basic
Imports System.Security.Permissions
' Indicates that FileIOPermission is required to run this assembly.
<Assembly: FileIOPermission(SecurityAction.RequestMinimum)>
   Public Class FileManager
' Insert code to add and delete files.
End Class

// C#
using System.Security.Permissions;
```

```
// Indicates that FileIOPermission is required to run this assembly.
[assembly: FileIOPermission(SecurityAction.RequestMinimum)]
public class FileManager
{
// Insert code to add and delete files.
}
```

To add a security attribute to a code member of your component

1. Identify the code member you want to protect and the permission that you will require to access it.
2. Attach the security attribute, by calling the **SecurityAction.Demand** method to require the desired permission.

```
' Visual Basic
Imports System.Security.Permissions
<FileIOPermission(SecurityAction.Demand)> Public Sub FileDeleter()
' Insert code to delete files.
End Sub

// C#
using System.Security.Permissions;
[FileIOPermission(SecurityAction.Demand)]
public void FileDeleter()
{
// Insert code to delete files.
}
```

See Also

Adding Imperative Security Checks to Components | Code Security and Signing in Components | Security Tutorial | Key Security Concepts

Adding Imperative Security Checks to Components

Imperative security checks allow you to protect specific blocks of code by requiring appropriate permissions. You can use permission objects to add imperative security checks to your code. A permission object is an instance of a specialized class that represents a particular type of permission. For example, the **FileIOPermission** class represents the right to read, append, or write files or directories. You add an imperative security check by creating an instance of the appropriate security object, and then calling that object's **Demand** function. This has the effect of denying access to all callers except those that have the appropriate permission. If a caller does not have the appropriate permissions to execute a called block of code, a **SecurityException** is thrown. For this reason, you should always consider how you want to handle a potential security exception. For details, see Handling Exceptions in Your Component.

To add an imperative security check to your component

1. Determine the kind of security permission that is needed to protect your code. For example, if your code involves using the file system, you would choose **FileIOPermission**. For details on types of code-access permissions, see Code Access Permissions.
2. Decide how you will handle a **SecurityException** if thrown. If you choose to handle it in your component, you should enclose your protected code within a **Try...Catch...Finally** block (**try...catch...finally** in C#), with appropriate code to handle the exception in the **Catch** block. For details, see Handling Exceptions in Your Component.
3. Create an instance of the appropriate permission object immediately preceding the code you want to protect.

   Note Your permission object might require parameters when instanced. For details about which parameters are required, see the .NET Framework reference topic for the specific class.

```
' Visual Basic
Dim MyPermission as New _
   Security.Permissions.FileIOPermission(PermissionState.Unrestricted)

// C#
System.Security.Permissions.FileIOPermission MyPermission = new
System.Security.Permissions.FileIOPermission(
PermissionState.Unrestricted);
```

4. Call the **Demand** method of your permission object. This will require any calling methods to demonstrate the appropriate permissions in order to execute the protected code. This will throw an exception if permission is not granted, so you may want to enclose this statement within structured exception handling.

```
' Visual Basic
Try
   MyPermission.Demand()
Catch e as Exception
   ' Insert code to handle the exception.
End Try

// C#
try
   {
   MyPermission.Demand();
   }
catch (Exception e)
   {
   // Insert code to handle the exception.
   }
```

See Also

Code Security and Signing in Components | Adding Security Attributes to Components | Security Tutorial | Key Security Concepts

Code Security and Signing in Components

As a component author, you have three primary concerns when considering security for your components:

- To ensure that your code will run wherever it is deployed.
- To protect your component against unauthorized or malicious use, and in addition, provide safeguards that prevent the exploitation of any bugs in your code.
- To assure the developers and users of your component that your component is authentic, originated from you, and can be trusted.

The .NET Framework security model provides you with the ability to ensure all of these requirements and to provide trustworthy, secure code to your clients.

The common language runtime allows the administrator of a computer to set default permissions for that computer. Simply, the administrator decides what unsafe operations can and cannot be allowed to proceed. On a high-security computer, this may mean that operations required by your component are not allowed to execute. In that case, your component will fail to work properly.

You can address this concern by requesting permissions. Permissions represent potentially unsafe actions, such as access to the file system or calling native code. You request permissions by adding security attributes to your component. At compile time, these attributes are emitted into metadata in the assembly manifest. When the assembly is loaded, the runtime examines the permission requests and applies the security policy. If the policy allows the permissions to be granted, they will be granted. If policy does not allow the requested permissions to be granted, the administrator has the option of revising the policy to allow your component to run. Requesting the permissions required for your component to function ensures that every computer hosting your component will be fully informed as to the permissions required by the code. For details on requesting permissions for your assembly, see Requesting Permissions. For details about how to add security attributes to your code, see Adding Security Attributes to Components.

You can protect your component from unauthorized or malicious use by insuring that callers to your component have the appropriate permissions to use your component. This can be done through *imperative security checks*, which cause permissions to be checked at run time. Using imperative security checks, you are able to create permission objects that safeguard potentially misused regions of code. By calling the **Demand** method of a permission object, you are able to verify that all callers to your protected code member have the appropriate permission to access it. If a caller attempts to access a protected code member without permission, a **SecurityException** is thrown. For details about how to add imperative security checks, see Adding Imperative Security Checks to Components. You can also protect code members using security attributes. For details on protecting your code members with security attributes, see Adding Security Attributes to Components.

Code Signing

When you distribute your component, you will want to be able to assure users that you are the author, that the code is safe and can be trusted, and that the identity of the assembly can be verified. You accomplish this by *signing* your code. You can sign an assembly in two ways:

- Assign your assembly a *strong name*. A strong name consists of the assembly's identity, plus a public key and a digital signature. A strong name is guaranteed to be a unique name, and thereby ensures the identity of your assembly. For details, see Strong-Named Assemblies.

  Although providing a strong name for your assembly guarantees its identity, it is by no means a guarantee that the code can be trusted. Put another way, the strong name uniquely identifies and guarantees the identity of the assembly, but does not guarantee that you wrote it. Nevertheless, a trust decision can reasonably be made based on strong-name identity alone, provided that the assembly is a well-known assembly.

- Use a digital certificate, also called *signcode*. To sign your assembly using signcode, you must go to a third-party authority and prove your identity, whereupon you will obtain a digital certificate that guarantees you as the originator of the component. For details about strong names and signcode, see Assembly Security Considerations.

Special Considerations for Custom Controls

It is important to realize that user code that is executed by the designer at design time will always run fully trusted, even if the project with the user code is located where it would receive less than full-trust at runtime. For example, say you create a Custom Control that makes API calls every second, and locate the code on a network share. Another developer on the network might incorporate your custom control into his project. At design time, the code in the Custom Control will run as fully trusted, and will require no additional permissions. There are two important consequences of this: First is the fact that you might expose your local

machine to a security risk through the network by importing unsecure code. This would only be a concern in the case of a malicious user creating a damaging custom control, followed by a developer mistakenly adding it to his project. A second concern is that applications created by this process may fail to work in a real world environment as a developer may be unaware of what additional permissions may need to be granted.

Code security is a complex and important topic. You are encouraged to make a thorough study of it, so that you might better protect yourself and your clients. For details on how the .NET Framework handles security, see Key Security Concepts.

See Also

Key Security Concepts | Strong-Named Assemblies | Assembly Security Considerations | Requesting Permissions | Adding Security Attributes to Components | Adding Imperative Security Checks to Components | Securability | Introduction to Code Access Security

User Assistance for Components

As a component author, you should provide assistance and documentation for your users about your components. You can provide assistance at design time through descriptive metadata, and at run time by calling HTML Help.

You can supply descriptive information about your component through the use of *attributes*. Attributes are specialized classes that are attached to other classes or class members and are emitted into metadata at compile time. Attributes can provide information to the user or directly to the development environment. For example, the **DescriptionAttribute** attribute, when attached to a property or event, displays a brief descriptive string in the Properties window. Other attributes affect the way the development environment behaves with the component. For example, the **BrowsableAttribute** attribute determines whether a property appears in the Properties window.

Run-time assistance is not usually supplied by a component, but there are some cases where it may make sense. For example, a very large and complex component may have built-in Help so the developers do not have to write extensive documentation. The .NET Framework **Help** object encapsulates the HTML Help 1.*x* engine, and can be used to supply Help to your users at run time with HTML or .chm files. For details, see Specifying a Help File for Your Component.

In This Section

Specifying a Help File for Your Component
: Explains how to specify and display a Help file for your component at run time.

Providing Metadata Descriptions About Your Component
: Demonstrates how to create and attach standard and custom descriptive attributes to your component.

Providing Metadata for Component Properties, Methods, and Events
: Describes how to attach descriptive attributes to code members.

Related Sections

Help Support Changes in Visual Basic .NET
: Describes changes in writing and providing Help in Visual Basic .NET.

Providing Help in a Windows Application
: Describes how to provide Help in a Windows application.

Attributes Overview
: Provides an overview of attributes and how they are used in the .NET Framework.

Specifying a Help File for Your Component

In most situations, you should let the developers who are using your component enable the run-time Help. In some cases, however, it will make sense to allow your component to display HTML Help when called. HTML Help can be provided for components through the **System.Windows.Forms.Help** object. This object is a static class that encapsulates the HTML Help 1.*x* engine. This class cannot be instantiated, and its methods must be called directly. To display Help, invoke the Help.ShowHelp Method method. This overloaded method requires at least two arguments: the control that acts as the parent control of the Help dialog box, and the URL of the Help file. The Help file can be a compiled HTML Help 1.*x* file (.chm file) or an HTML file in the HTML Help format.

If you are going to incorporate support for a Help file directly in your component, you have two options for when and how to show it:

- The preferred option is to implement a Help method that can be called by the client application. The client application can pass parameters to the Help method to ensure that the correct topics are displayed, and the developer coding with your component has the option of bypassing Help altogether.
- The other option is to invoke the **Help.ShowHelp** method in response to conditions as they occur in code. This approach provides you the most control over what Help is displayed when, but it severely limits future developers in the use of your component.

To specify and display a Help file for your component

1. Create and compile your .chm Help file.
2. If you do not already have a reference to the **System.Windows.Forms** namespace in your component, add one.
3. Create a public method to show Help. This method should provide an easy way for developers to specify what Help they need to display.

```
' Visual Basic
' This method takes parameters from the client application that allow
' the developer to specify when Help is displayed.
Public Sub DisplayHelp (ByVal parent as System.Windows.Forms.Control, _
                        ByVal topic as MyHelpEnum)
   ' The file to display is chosen by the value of the topic
   ' parameter.
   Select Case topic
      Case MyHelpEnum.enumWidgets
         Windows.Forms.Help.ShowHelp(parent, "C:\Help\Widgets.chm")
      Case MyHelpEnum.enumMechanisms
         ' Insert code to implement additional functionality.
   End Select
End Sub

// C#
// This method takes parameters from the client application that allow
// the developer to specify when Help is to be shown.
public void MyHelp(System.Windows.Forms.Control parent,
myHelpEnum topic)
{
   // The file to display is chosen by the value of the topic.
   switch (topic)
   {
      case myHelpEnum.enumWidgets:
         System.Windows.Forms.Help.ShowHelp(parent, " C:\\help\\widgets.chm ");
         break;
      case myHelpEnum.enumMechanism:
         // Insert code to implement additional functionality.
         break;
   }
}
```

See Also

Help Class | User Assistance for Components

Providing Metadata Descriptions About Your Component

You can provide descriptive metadata about your component through *attributes*. Attributes are specialized classes that are applied to code elements. At compile time, attributes are emitted into metadata that is available to the common language runtime or to custom tools and applications through the **System.Reflection** namespace.

Attributes are attached to a component by preceding the component with a reference to the attribute and providing any relevant parameters or flags. This call to the constructor is placed within angle brackets <> in Visual Basic and regular brackets [] in C#.

By convention, all attribute classes end in "Attribute." For example, there are **DescriptionAttribute**, **ObsoleteAttribute**, and **BrowsableAttribute** classes. However, several languages that target the common language runtime, including Visual Basic and C#, do not require the full name of the attribute. For example, the **ObsoleteAttribute** may be referred to in code as **Obsolete**.

To attach an existing attribute to your component

1. Determine which attributes are required for your component.
2. Attach the attributes to your component. Note that you must either use the fully qualified name of the attribute or add the appropriate **Imports** (**using**) statement. The following example shows how to attach the **Description** attribute:

```
' Visual Basic
Imports System.ComponentModel
<Description("This is a description string")> Public Class TheClass
End Class

// C#
using System.ComponentModel
[Description("This is a description string")]
public class TheClass
{
}
```

Custom Attributes

You can also create your own attributes for use with your own custom tools or applications by inheriting from **System.Attribute**. You can add to this base class any custom properties or methods needed by your application.

To create and apply a custom attribute

1. Create a class that inherits from **System.Attribute**.

```
' Visual Basic
Public Class WidgetAttribute
   Inherits System.Attribute
End Class

// C#
public class WidgetAttribute: System.Attribute
{
}
```

2. Determine which properties and methods your attribute needs and write the code for them. The following example shows how to create a WidgetType property that is set in the constructor of the WidgetAttribute class. The **AttributeUsage** attribute sets what code members the attribute can target.

```
' Visual Basic
<AttributeUsage(System.AttributeTargets.Class)> Public Class _
   WidgetAttribute
   Inherits System.Attribute
   Private mWidgetType as WidgetTypeEnum
   ' Creates a readonly property for the WidgetAttribute class.
   Public ReadOnly Property WidgetType as WidgetTypeEnum
```

```
      Get
         Return mWidgetType
      End Get
   End Property
   ' Creates a constructor that accepts a parameter and assigns the
   ' value of that parameter to the WidgetType property.
   Public Sub New(type as WidgetTypeEnum)
      MyBase.New()
      mWidgetType = type
   End Sub
End Class

//C#
public class WidgetAttribute: System.Attribute
{
   private WidgetTypeEnum widgetType;

   // Creates a readonly property for the WidgetAttribute class.
   public WidgetTypeEnum WidgetType
      {
         get {return widgetType;}
      }

   public WidgetAttribute(WidgetTypeEnum type): base()
      {
         widgetType = type;
      }
}
```

3. Apply this attribute as you would any other attribute, being certain to apply any needed parameters.

```
' Visual Basic
<WidgetAttribute(WidgetTypeEnum.VerticalWidget)> _
   Public Class WidgetFortyFive
End Class

// C#
[WidgetAttribute(WidgetTypeEnum.VerticalWidget)]
public class WidgetFortyFive
{
}
```

See Also

Accessing Custom Attributes | Attribute Class | Retrieving Information Stored in Attributes | Providing Metadata for Component Properties, Methods, and Events | User Assistance for Components

Providing Metadata for Component Properties, Methods, and Events

You can attach descriptive metadata to your properties, methods, and events by using attributes. Attributes are specialized classes that modify code elements and are emitted into metadata about those code elements at compile time. Design-time attributes interact with the development environment to provide the developer with information about your component. An example of a commonly used attribute is the **DescriptionAttribute**. When attached to a property or event, the **Description** attribute causes a short string to be displayed in the Properties window. An example is shown below:

```
' Visual Basic
<Description("This property specifies the active Widget")> Public _
   Property ActiveWidget as Widget
' Insert code to implementat functionality.
End Property

// C#
[Description("This property specifies the active Widget")]
public Widget ActiveWidget
{
   // Insert code to implementat functionality.
}
```

For details about attributes you can use to enhance design-time support for your component, see Design-Time Attributes for Components.

To attach an attribute to a property, method, or event

- Add a call to the attribute's constructor to the code element you wish to modify. The call is added by placing it in angle brackets <> (for Visual Basic) or brackets [] (for C#), preceding the code element you want to modify.

```
' Visual Basic
<Browsable(False)> Public Property ServingSize as Integer

// C#
[Browsable(false)]
public int ServingSize
```

See Also

Design-Time Attributes for Components | Accessing Custom Attributes | Attribute Class | Retrieving Information Stored in Attributes | Providing Metadata Descriptions About Your Component | User Assistance for Components

Control Authoring for Windows Forms

For control authors, the .NET Framework provides an unprecedented wealth of control authoring technology. Authors are no longer limited to designing user controls that act as a collection of preexisting controls. Now, through inheritance, authors can inherit from preexisting user controls, preexisting Windows Forms controls, or even design their own controls that execute custom painting. The new options available give an unprecedented level of flexibility to the design and functionality of the visual interface. To take advantage of these new features, you should be familiar with object-based programming concepts.

> **Visual Basic Note** It is not necessary to have a thorough understanding of inheritance, but you may find it useful to refer to Inheritance.

If you want to create custom controls to use on Web Forms, see Developing ASP.NET Server Controls.

In This Section

Walkthrough: Authoring a User Control With Visual Basic .NET
Demonstrates how to create a user control with Visual Basic.
Walkthrough: Authoring a User Control with Visual C#
Demonstrates how to create a user control with Visual C#.
Walkthrough: Inheriting from a Windows Forms Control with Visual Basic .NET
Demonstrates how to create a custom control that inherits from an existing Windows Form control in Visual Basic.
Walkthrough: Inheriting from a Windows Forms Control with Visual C#
Demonstrates how to create a custom control that inherits from an existing Windows Form control in Visual C#.
Control Type Recommendations
Provides guidance on inheriting from the **UserControl** class versus inheriting from the **Control** class versus inheriting from a Windows Forms control.
Authoring Controls for Windows Forms
Gives the high-level process for creating a custom control.
Authoring User Controls
Gives general information regarding the creation of user controls, control class libraries, and inheriting from user control in control class libraries.
Inheriting from the UserControl Class
Describes how to create user controls for Windows Forms by inheriting from the **UserControl** class.
Inheriting from the Control Class
Explains how to inherit from the **Control** class to create custom client controls that paint their own visual interface.
Inheriting from Existing Windows Forms Controls
Gives general information on inheriting from standard Windows Forms controls, such as **Button** or **TextBox**.
Properties, Methods, and Events for Custom Controls
Describes adding properties, methods, and events to custom controls.
Custom Control Painting and Rendering
Provides links to information about incorporating custom painting in your control.
Making Your Control Invisible at Run Time
Demonstrates how to hide a user control.
Aligning Your Control to the Edges of Forms
Demonstrates how to make your control align to the edge of the form it occupies.
Providing a Toolbox Bitmap for Your Control
Describes how to specify a toolbox bitmap for your control.
Extender Provider Objects
Gives an overview of extender objects, describes their general functionality, and gives tips on creating custom extenders.
Giving Your Control a Transparent Background
Demonstrates how to make your control transparent.
Displaying Your Control in the Customize Toolbox Dialog Box
Describes how to enable your control to be displayed in the **Customize Toolbox** dialog box.

Related Sections

What is the Common Language Specification?
Introduces the Common Language Runtime, which is designed to simplify the creation and use of components. An important aspect of this simplification is enhanced interoperability between components written using different programming languages. The Common Language Specification (CLS) makes it possible to create tools and components that work with multiple programming languages.

Walkthrough: Authoring a User Control with Visual Basic .NET

User controls provide a means by which custom graphical interfaces can be created and reused. A user control is essentially a component with a visual representation. As such, it might consist of one or more Windows Forms controls, components, or blocks of code that can extend functionality by validating user input, modifying display properties, or performing other tasks required by the author. User controls can be placed on Windows Forms in the same manner as other controls. In the first part of this walkthrough, you create a simple user control called ctlClock. In the second part of the walkthrough, you extend the functionality of ctlClock through inheritance.

Creating the Project

When you create a new project, you specify its name to set the root namespace, assembly name, and project name, and ensure that the default component will be in the correct namespace.

To create the ctlClockLib control library and the ctlClock control

1. On the **File** menu, point to **New** and then select **Project** to open the **New Project** dialog box.
2. Select the **Windows Control Library** project template from the list of Visual Basic Projects, and type **ctlClockLib** in the **Name** box.

   > **Note** The Windows Control Library template is not available in the Standard Edition of Visual Basic .NET. For more information, see Visual Basic Standard Edition Features.

   The project name, **ctlClockLib**, is also assigned to the root namespace by default. The root namespace is used to qualify the names of components in the assembly. For example, if two assemblies provide components named ctlClock, you can specify your ctlClock component using `ctlClockLib.ctlClock`. In Solution Explorer, right-click **UserControl1** and select **View Code** from the shortcut menu.

3. Locate the **Class** statement, `Public Class UserControl1`, and change **UserControl1** to **ctlClock** to change the name of the component.

   > **Note** Notice the **Inherits** statement immediately below the **Class** statement. By default, a user control inherits from the **UserControl** class provided by the system. The **UserControl** class provides functionality required by all user controls, and implements standard methods and properties.

4. In Solution Explorer, click **UserControl1**, then in the Properties window, change the **FileName** property to **ctlClock.vb**.
5. From the **File** menu, choose **Save All** to save the project.

Adding Windows Controls and Components to your User Control

A visual interface is an essential part of your user control. This visual interface is implemented by the addition of one or more Windows controls to the User Control Designer. In the following demonstration, you will incorporate Windows controls into your user control and write code to implement functionality.

To add a Label and a Timer to your user control

1. In Solution Explorer, right-click **ctlClock.vb,** and then select **View Designer** from the shortcut menu.
2. In the Toolbox, click the **Windows Forms** tab, and then double-click **Label**.

   A label control named **Label1** is added to your control on the **User Control Designer**.

3. In the designer, click **Label1**. In the Properties window, set the following properties.

Property	Change to
Name	lblDisplay
Text	(blank space)
TextAlign	MiddleCenter
Font.Size	14

4. In the Toolbox, click **Windows Forms**, and then double-click **Timer**.

   Because a timer is a component, it has no visual representation at run time. It therefore does not appear with the controls on the **User Control Designer**, but rather in the component tray.

5. In the component tray, click **Timer1** and set the **Interval** property to **1000** and the **Enabled** property to **True**.

The **Interval** property controls the frequency with which the timer component ticks. Each time **Timer1** ticks, it runs the code in the **Timer1_Tick** event. The interval represents the number of milliseconds between ticks.

6. In the component tray, double-click **Timer1** to go to the **Timer1_Tick** event for ctlClock.
7. Modify the code so that it resembles the following code sample. Be sure to change the access modifier from **Private** to **Protected**:

```
Protected Sub Timer1_Tick(ByVal sender As Object, ByVal e As _
   System.EventArgs) Handles Timer1.Tick
   ' Causes the label to display the current time
   lblDisplay.Text = Format(Now, "hh:mm:ss")
End Sub
```

This code will cause the current time to be shown in **lblDisplay**. Because the interval of **Timer1** was set to 1000, this event will fire every thousand milliseconds, thus updating the current time every second.

8. Modify the method to be overridable. For more information, see the next section of this topic, "Inheriting from a User Control".

```
' Visual Basic
Protected Overridable Sub Timer1_Tick(ByVal sender As Object, ByVal _
   e As System.EventArgs) Handles Timer1.Tick
```

9. From the **File** menu, choose **Save ctlClock** to save the project.

Adding Properties to Your User Control

Your clock control now encapsulates a **Label** control and a **Timer** component, each with its own set of inherent properties. While the individual properties of these controls will not be accessible to subsequent users of your control, you can create and expose custom properties by writing the appropriate blocks of code. In the following section, you will add properties to your control that enable the user to change the color of the background and text.

To add a Property to your user control

1. In Solution Explorer, right-click **ctlClock.vb**, and then click **View Code** from the shortcut menu.

   The Code Editor for your control opens.

2. Locate the `Inherits System.Windows.Forms.UserControl` statement. Beneath it, type:

```
Private colFColor as Color
Private colBColor as Color
```

These statements create the private variables that you will use to store the values for the properties you are about to create.

3. Type the following code beneath the variable declarations from step 2:

```
' Declares the name and type of the property.
Property ClockBackColor() as Color
   ' Retrieves the value of the private variable colBColor.
   Get
      Return colBColor
   End Get
   ' Stores the selected value in the private variable colBColor, and
   ' updates the backcolor of the label control lblDisplay.
   Set(ByVal Value as Color)
      colBColor = Value
      lblDisplay.BackColor = ColBColor
   End Set

End Property
' Provides a similar set of instructions for the forecolor.
Property ClockForeColor() as Color
```

```
    Get
       Return colFColor
    End Get
    Set(ByVal Value as Color)
       colFColor = Value
       lblDisplay.ForeColor = ColFColor
    End Set
 End Property
```

The preceding code makes two custom properties, ClockForeColor and ClockBackColor, available to subsequent users of this control by invoking the `Property` statement. The **Get** and **Set** statements provide for storage and retrieval of the property value, as well as code to implement functionality appropriate to the property.

4. From the **File** menu, choose **Save ctlClock.vb** to save the project.

Testing Your Control

Controls are not stand-alone projects; they must be hosted in a container. In order to test your control, you must provide a test project within which it runs. In this section, you will build your control and test it in a Windows Form.

To build your control

- On the **Build** menu, click **Build ctlClockLib**.

To create a test project

1. On the **File** menu, point to **Add Project** and then click **New Project** to open the **Add New Project** window.
2. Click **Windows Application**, and in the **Name** box, type **Test**, and click **OK**.
3. In Solution Explorer, right-click the **References** node for your test project. Click **Add Reference** to display the **Add Reference** dialog box.
4. Click the tab labeled **Projects**. Your project will be listed under **Project Name**.
5. Double-click your project.

   Note that your project now appears in the **Selected Components** window.

After you have added your reference, you can place your control in your form.

To test your control

1. In the Toolbox, click **My User Controls**, and then scroll down until the control icon representing **ctlClock** is displayed.
2. Double-click the **ctlClock** icon.

   A copy of your control is now added to your form. Note that it displays the current time and is updated each second.

3. In the Toolbox, select the icon and move your mouse over your form.
4. Hold down the left mouse button while moving the mouse across the form.

   Another copy of your control is drawn onto the form. You can add as many copies of your timer to your form as you want.

5. In the designer, click one of the instances of **ctlClock**.

   The properties for this instance are displayed in the Properties window.

6. In the Properties window, locate the **ClockBackColor** property, and then select the property to display the color palette.
7. Choose a color by clicking it.

   The background color of your control changes to the color you selected.

8. Use a similar sequence of events to verify that the **ClockForeColor** property is functioning as expected.

In this section, you have seen how components and Windows controls can be combined with code and packaging to provide custom functionality in the form of a user control. You have learned to expose properties in your user control, and how to test your control after it is complete. In the next section you will learn how to construct an inherited user control using **ctlClock** as a base.

Inheriting from a User Control

In the previous section, you learned how to combine Windows controls, components, and code into reusable user controls. Your user control can now be used as a base upon which other controls can be built. The process of deriving a class from a base class is called *inheritance*. In this section, you will create a user control called ctlAlarmClock. This control will be derived from its parent control, ctlClock. You will learn to extend the functionality of ctlClock by overriding parent methods and adding new methods and properties.

Creating the Inherited Control

The first step in creating an inherited control is to derive it from its parent. This action creates a new control that has all of the properties, methods, and graphical characteristics of the parent control, but can also act as a base for the addition of new or modified functionality.

To create the inherited control

1. In Solution Explorer, click **ctlClockLib**.
2. From the **Project** menu, select **Add Inherited Control**.

   The **Add New Item** window opens with **Inherited User Control** selected.

3. In the **Name** box, type **ctlAlarmClock.vb**, and click **Open**.

   The **Inheritance Picker** window appears.

4. Under **Component Name**, double-click **ctlClock**.
5. In Solution Explorer, browse the current projects. Note that a file called **ctlAlarmClock** has been added.

Adding the Alarm Properties

Properties are added to an inherited control in the same way they are added to a user control. You will now use the property declaration syntax to add two properties to your control: **AlarmTime**, which will store the value of the date and time the alarm is to go off, and **AlarmSet**, which will indicate whether or not the alarm is set.

To add Properties to your user control

1. In Solution Explorer, right-click **ctlAlarmClock** and select **View Code** from the shortcut menu.
2. Locate the `Inherits` statement. Note that your control inherits from **ctlClockLib.ctlClock**. Beneath the `Inherits` statement, type the following code:

```
Private dteAlarmTime As Date
Private blnAlarmSet As Boolean
' These properties will be declared as public to allow future
' developers to access them.
Public Property AlarmTime() As Date
   Get
      Return dteAlarmTime
   End Get
   Set(ByVal Value as Date)
      dteAlarmTime = Value
   End Set
End Property
Public Property AlarmSet() As Boolean
   Get
      Return blnAlarmSet
   End Get
   Set(ByVal Value as Boolean)
      blnAlarmSet = Value
   End Set
End Property
```

Adding to the Graphical Interface of Your Control

Your inherited control has a visual interface that is identical to the control it inherits from. It possesses the same constituent controls as its parent control, but the properties of the constituent controls will not be available unless they were specifically

exposed. You may add to the graphical interface of an inherited user control in the same manner as you would add to any user control. To continue adding to your alarm clock's visual interface, you will add a label control that will flash when the alarm is sounding.

To add the label control

1. In Solution Explorer, right-click **ctlAlarmClock** and select **View Designer** from the shortcut menu.

   The designer for **ctlAlarmClock** opens in the main window.

2. Click the display portion of the control, and observe the Properties window.

   Note that while all the properties are displayed, they are dimmed. This indicates that these properties are native to **lblDisplay**, and cannot be modified or accessed in the properties window. By default, controls contained in a user control are `Private`, and their properties are not accessible by any means.

   > **Tip** If you want subsequent users of your user control to have access to its internal controls, declare them as Public or Protected. This will allow you to set and modify properties of controls contained within your user control by using the appropriate code.

3. Add a **Label** control to your user control.
4. Using the mouse, move the label control immediately beneath the display box. In the Properties window, set the following properties:

Property	Setting
Name	**lblAlarm**
Text	**Alarm Is Sounding!**
TextAlign	**Middle Center**
Visible	**false**

Adding the Alarm Functionality

In the previous sections, you added properties and a control that will enable alarm functionality in your user control. In this section, you will add code to compare the current time to the alarm time, and if they are the same, to sound and flash an alarm. By overriding the **Timer1_Tick** method of **ctlClock**, and adding additional code to it, you will extend the capability of **ctlAlarmClock** while retaining all of the inherent functionality of **ctlClock**.

To override the Timer1_Tick method of ctlClock

1. In the Code Editor, locate the `Private blnAlarmSet As Boolean` statement. Immediately beneath it, add the following statement:

```
Dim blnColorTicker as Boolean
```

2. In the Code Editor, locate the `End Class` statement. It can be found at the bottom of the page.
3. Just before the `End Class` statement, add the following code:

```
Protected Overrides Sub Timer1_Tick(ByVal sender As Object, ByVal e _
   As System.EventArgs)
   ' Calls the Timer1_Tick method of ctlClock.
   MyBase.Timer1_Tick(sender, e)
   ' Checks to see if the Alarm is set.
   If AlarmSet = False Then
      Exit Sub
   End If
   ' If the date, hour and minute of the alarm time are the same as
   ' now, flash and Beep!
   If AlarmTime.Date = Now.Date And AlarmTime.Hour = Now.Hour And _
      AlarmTime.Minute = Now.Minute Then
      ' Sounds an audible Beep
```

```
        Beep()
        ' Makes lblAlarmVisible, and changes the backcolor based on the
        ' value of blnColorTicker. The backcolor of the label will
        ' flash once per tick of the clock.
        lblAlarm.Visible = True
        If blnColorTicker = False Then
           lblAlarm.BackColor = Color.PeachPuff
           blnColorTicker = True
        Else
           lblAlarm.BackColor = Color.Plum
           blnColorTicker = False
        End If
     Else
        ' Once the alarm has sounded for a minute, the label is made
        ' invisible again.
        lblAlarm.Visible = False
     End If
  End Sub
```

The addition of this code accomplishes several tasks. The `Overrides` statement directs the control to use this method in place of the method that was inherited from the base control. When this method is called, it calls the method it overrides by invoking the `MyBase.Timer1_Tick` statement, ensuring that all of the functionality incorporated in the original control is reproduced in this control. It then runs additional code to incorporate the alarm functionality. A flashing label control will appear when the alarm is sounding, and an audible beep will be heard.

Note that since you are overriding an inherited event handler, you do not have to specify the event with the `Handles` keyword. The event is already hooked up. All you are overriding is the implementation of the handler.

Your alarm clock control is almost complete. The only thing that remains is to implement a way to turn it off. To do this, we will add code to the **lblAlarm_Click** method.

To implement the shutoff method

1. In Solution Explorer, right-click **ctlAlarmClock.vb** and click **View Designer**.

   The designer opens.

2. In the designer, double-click **lblAlarm**. The **Code Editor** opens to the `Protected Sub lblAlarm_Click` line.
3. Modify this method so that it resembles the following:

```
Protected Sub lblAlarm_Click(ByVal sender As Object, ByVal e As _
 System.EventArgs) Handles lblAlarm.Click
   ' Turns off the alarm.
   AlarmSet = False
   ' Hides the flashing label.
   lblAlarm.Visible = False
End Sub
```

4. From the **File** menu, choose **Save ctlAlarmClock.vb** to save the project.

Testing Your Inherited Control

As with a standard user control, an inherited user control cannot stand alone and must be hosted in a form or other container. Since **ctlAlarmClock** has a greater depth of functionality, additional code is required to test it. In this section, you will write a simple program to test the functionality of **ctlAlarmClock**. You will write code to set and display the **AlarmTime** property of **ctlAlarmClock**, and will test its inherent functions.

To build and add your control to a test form

1. In Solution Explorer, click **ctlClockLib**. On the **Build** menu, choose **Build ctlClockLib**.
2. Add a new **Windows Application** project to the solution, and name it **Test2**.
3. In Solution Explorer, right-click the **References** node for your test project. Click **Add Reference** to display the **Add**

Reference window. Click the tab labeled **Projects**. The project **ctlClockLib** will be listed under Project Name. Double-click **ctlClockLib**, and note that it now appears in the Selected Components window.

4. In the Toolbox, click **My User Controls**.
5. Scroll down until the icon for **ctlAlarmClock** comes into view.
6. Double-click **ctlAlarmClock** to add a copy of **ctlAlarmClock** to your form.
7. Locate and double-click **DateTimePicker** to add a **DateTimePicker** control to your form, and add a **Label** control by double-clicking **Label**.
8. Use the mouse to position the controls in a convenient place on the form.
9. Set the properties of these controls as follows:

Control	Property	Value
Label1	**Text**	(leave blank)
	Name	**lblTest**
DateTimePicker1	**Name**	**dtpTest**
	Format	**Time**

10. In the designer, double-click **dtpTest**.

    The Code Editor opens to `Protected Sub dtpTest_ValueChanged`.

11. Modify the code so that it resembles the following:

```
Protected Sub dtpTest_ValueChanged(ByVal sender As Object, ByVal e As _
   System.EventArgs) Handles dtpTest.ValueChanged
   ctlAlarmClock1.AlarmTime = dtpTest.Value
   ctlAlarmClock1.AlarmSet = True
   lblTest.Text = "Alarm Time is " & Format(ctlAlarmClock1.AlarmTime, _
      "hh:mm")
End Sub
```

12. In Solution Explorer, right-click **Test2** and choose **Set as StartUp Project** from the shortcut menu.
13. From the **Debug** menu, choose **Start**.

    The test program starts. Note that the current time is updated in the **ctlAlarmClock** control, and that the starting time is shown in the **DateTimePicker** control.

14. Click the **DateTimePicker** where the minutes of the hour are displayed.
15. Using the keyboard, set a value for minutes that is one minute greater than the current time shown by **ctlAlarmClock**.

    The time for the alarm setting is shown in **lblTest**.

16. Wait for the displayed time to reach the alarm setting time.

    When the displayed time reaches the time to which the alarm is set, the beep will sound and **lblAlarm** will flash. Turn off the alarm by clicking **lblAlarm**. You may now reset the alarm.

This walkthrough has covered a number of key concepts. You have learned to create a user control by combining controls and components into a user control container. You have learned to add properties to your control, and to write code to implement custom functionality. In the second section, you learned to extend the functionality of a given user control through inheritence, and to alter the functionality of host methods by overriding those methods.

See Also

Authoring User Controls | Displaying Your Control in the Customize Toolbox Dialog Box | Component Authoring Walkthroughs

Walkthrough: Authoring a User Control with Visual C#

User controls provide a means by which custom graphical interfaces can be created and reused. A user control is essentially a component with a visual representation. As such, it might consist of one or more Windows Forms controls, components, or blocks of code that can extend functionality by validating user input, modifying display properties, or performing other tasks required by the author. User controls can be placed on Windows Forms in the same manner as other controls. In the first part of this walkthrough, you create a simple user control called ctlClock. In the second part of the walkthrough, you extend the functionality of ctlClock through inheritance.

Creating the Project

When you create a new project, you specify its name to set the root namespace, assembly name, and project name, and ensure that the default component will be in the correct namespace.

To create the ctlClockLib control library and the ctlClock control

1. On the **File** menu, point to **New** and then select **Project** to open the **New Project** dialog box.
2. Select the **Windows Control Library** project template from the list of C# Projects, and type **ctlClockLib** in the **Name** box.

   > **Note** The Windows Control Library template is not available in the Standard Edition of Visual C# .NET. For more information, see Visual C# Standard Edition Features.

   The project name, **ctlClockLib**, is also assigned to the root namespace by default. The root namespace is used to qualify the names of components in the assembly. For example, if two assemblies provide components named ctlClock, you can specify your ctlClock component using `ctlClockLib.ctlClock`. In Solution Explorer, right-click **UserControl1** and select **View Code** from the shortcut menu.

3. Locate the **Class** statement, `public class UserControl1`, and change **UserControl1** to **ctlClock** to change the name of the component.

   > **Note** By default, a user control inherits from the **UserControl** class provided by the system. The **UserControl** class provides functionality required by all user controls, and implements standard methods and properties.

4. Locate the constructor, `public UserControl1()`, and change `UserControl1` to `ctlClock`.
5. In Solution Explorer, click **UserControl1**, then in the Properties window, change the **FileName** property to **ctlClock.cs**.
6. From the **File** menu, choose **Save All** to save the project.

Adding Windows Controls and Components to your User Control

A visual interface is an essential part of your user control. This visual interface is implemented by the addition of one or more Windows controls to the User Control Designer. In the following demonstration, you will incorporate Windows controls into your user control and write code to implement functionality.

To add a Label and a Timer to your user control

1. In Solution Explorer, right-click **ctlClock.cs,** and then select **View Designer**.
2. In the Toolbox, click the **Windows Forms** tab, and then double-click **label**.

   A label control named **label1** is added to your control on the **User Control Designer**.

3. In the designer, click **label1**. In the Properties window, set the following properties.

Property	Change to
Name	lblDisplay
Text	(blank space)
TextAlign	MiddleCenter
Font.Size	14

4. In the Toolbox, click **Windows Forms**, and then double-click **Timer**.

   Because a timer is a component, it has no visual representation at run time. It therefore does not appear with the controls on the **User Control Designer**, but rather in the component tray.

5. In the component tray, click **timer1** and set the **Interval** property to **1000**, and the **Enabled** property to **true**.

The **Interval** property controls the frequency with which the timer component ticks. Each time **timer1** ticks, it runs the code in the **timer1_Tick** event. The interval represents the number of milliseconds between ticks.

6. In the component tray, double-click **timer1** to go to the **timer1_Tick** event for ctlClock.
7. Modify the code so that it resembles the following code sample. Be sure to change the access modifier from **private** to **protected**:

```
protected void timer1_Tick(object sender, System.EventArgs e)
{
   // Causes the label to display the current time
   lblDisplay.Text = DateTime.Now.ToLongTimeString();
}
```

This code will cause the current time to be shown in **lblDisplay**. Because the interval of **timer1** was set to 1000, this event will fire every thousand milliseconds, thus updating the current time every second.

8. Modify the method to be overridable with the **virtual** keyword. For more information, see the next section of this topic, "Inheriting from a User Control".

```
protected virtual void timer1_Tick(object sender, System.EventArgs e)
```

9. From the **File** menu, choose **Save All** to save the project.

Adding Properties to Your User Control

Your clock control now encapsulates a **Label** control and a **Timer** component, each with its own set of inherent properties. While the individual properties of these controls will not be accessible to subsequent users of your control, you can create and expose custom properties by writing the appropriate blocks of code. In the following section, you will add properties to your control that enable the user to change the color of the background and text.

To add a Property to your user control

1. In Solution Explorer, right-click **ctlClock.cs**, and then click **View Code** from the shortcut menu.

   The Code Editor for your control opens.

2. Locate the `public class ctlClock` statement. Beneath the opening `{`, type:

```
private Color colFColor;
private Color colBColor;
```

These statements create the private variables that you will use to store the values for the properties you are about to create.

3. Type the following code beneath the variable declarations from step 2:

```
// Declares the name and type of the property.
public Color ClockBackColor
// Retrieves the value of the private variable colBColor.
{
   get
   {
      return colBColor;
   }
   // Stores the selected value in the private variable colBColor, and
   // updates the backcolor of the label control lblDisplay.
   set
   {
      colBColor = value;
      lblDisplay.BackColor = colBColor;
   }
}
// Provides a similar set of instructions for the forecolor.
public Color ClockForeColor
```

```
{
   get
   {
      return colFColor;
   }
   set
   {
      colFColor = value;
      lblDisplay.ForeColor = colFColor;
   }
}
```

The preceding code makes two custom properties, ClockForeColor and ClockBackColor, available to subsequent users of this control. The **get** and **set** statements provide for storage and retrieval of the property value, as well as code to implement functionality appropriate to the property.

4. From the **File** menu, choose **Save All** to save the project.

Testing Your Control

Controls are not stand-alone applications; they must be hosted in a container. In order to test your control, you must provide a test project within which it runs. In this section, you will build your control and test it in a Windows Form.

To build your control

- On the **Build** menu, click **Build**.

To create a test project

1. On the **File** menu, point to **Add Project** and then click **New Project** to open the **Add New Project** window.
2. Click **Windows Application**, and in the **Name** box, type **Test**.
3. In Solution Explorer, right-click the **References** node for your test project. Click **Add Reference** to display the **Add Reference** dialog box.
4. Click the tab labeled **Projects**. Your user control project will be listed under **Project Name**.
5. Double-click your project.

   Note that your project now appears in the **Selected Components** window.

After you have added your reference, you can place your control in your form.

To test your control

1. In the Toolbox, click **My User Controls**, and then scroll down until the control icon representing **ctlClock** is displayed.
2. Double-click the **ctlClock** icon.

   A copy of your control is now added to your form. Note that it displays the current time and is updated each second.

3. In the **Toolbox**, select the icon and move your mouse over your form.
4. Hold down the left mouse button while moving the mouse across the form.

   Another copy of your control is drawn onto the form. You can add as many copies of your timer to your form as you want.

5. In the designer, click one of the instances of **ctlClock**.

   The properties for this instance are displayed in the Properties window.

6. In the Properties window, locate the **ClockBackColor** property, then select the property to display the color palette.
7. Choose a color by clicking it.

   The background color of your control changes to the color you selected.

8. Use a similar sequence of events to verify that the **ClockForeColor** property is functioning as expected.

In this section, you have seen how components and Windows controls can be combined with code and packaging to provide custom functionality in the form of a user control. You have learned to expose properties in your user control, and how to test your control after it is complete. In the next section you will learn how to construct an inherited user control using **ctlClock** as a

base.

Inheriting from a User Control

In the previous section, you learned how to combine Windows controls, components, and code into reusable user controls. Your user control can now be used as a base upon which other controls can be built. The process of deriving a class from a base class is called *inheritance*. In this section, you will create a user control called ctlAlarmClock. This control will be derived from its parent control, ctlClock. You will learn to extend the functionality of ctlClock by overriding parent methods and adding new methods and properties.

Creating the Inherited Control

The first step in creating an inherited control is to derive it from its parent. This action creates a new control that has all of the properties, methods, and graphical characteristics of the parent control, but can also act as a base for the addition of new or modified functionality.

To create the inherited control

1. In Solution Explorer, click **ctlClockLib**.
2. From the **Project** menu, select **Add Inherited Control**.

   The **Add New Item** window opens with **Inherited User Control** selected.

3. In the **Name** box, type **ctlAlarmClock.cs**, and click **Open**.

   The **Inheritance Picker** window appears.

4. Under **Component Name**, double-click **ctlClock**.
5. In Solution Explorer, browse the current projects. Note that a file called **ctlAlarmClock** has been added.

Adding the Alarm Properties

Properties are added to an inherited control in the same way they are added to a user control. You will now use the property declaration syntax to add two properties to your control: **AlarmTime**, which will store the value of the date and time the alarm is to go off, and **AlarmSet**, which will indicate whether or not the alarm is set.

To add Properties to your user control

1. In Solution Explorer, right-click **ctlAlarmClock** and select **View Code**.
2. Locate the `public class` statement. Note that your control inherits from **ctlClockLib.ctlClock**. Beneath the `{` statement, type the following code:

```
private DateTime dteAlarmTime;
private bool blnAlarmSet;
// These properties will be declared as public to allow future
// developers to access them.
public DateTime AlarmTime
{
   get
   {
      return dteAlarmTime;
   }
   set
   {
      dteAlarmTime = value;
   }
}
public bool AlarmSet
{
   get
   {
      return blnAlarmSet;
   }
   set
```

```
    {
       blnAlarmSet = value;
    }
 }
```

Adding to the Graphical Interface of Your Control

Your inherited control has a visual interface that is identical to the control it inherits from. It possesses the same constituent controls as its parent control, but the properties of the constituent controls will not be available unless they were specifically exposed. You may add to the graphical interface of an inherited user control in the same manner as you would add to any user control. To continue adding to your alarm clock's visual interface, you will add a label control that will flash when the alarm is sounding.

To add the label control

1. In Solution Explorer, right-click **ctlAlarmClock** and select **View Designer** from the shortcut menu.

   The designer for **ctlAlarmClock** opens in the main window.

2. Click the display portion of the control, and observe the Properties window.

   Note that while all the properties are displayed, they are dimmed. This indicates that these properties are native to **lblDisplay**, and cannot be modified or accessed in the properties window. By default, controls contained in a user control are `private`, and their properties are not accessible by any means.

   > **Tip** If you want subsequent users of your user control to have access to its internal controls, declare them as **public** or **protected**. This will allow you to set and modify properties of controls contained within your user control by using the appropriate code.

3. Add a **Label** control to your user control.
4. Using the mouse, move the label control immediately beneath the display box. In the Properties window, set the following properties:

Property	Setting
Name	**lblAlarm**
Text	**Alarm!**
TextAlign	**Middle Center**
Visible	**false**

Adding the Alarm Functionality

In the previous sections, you added properties and a control that will enable alarm functionality in your user control. In this section, you will add code to compare the current time to the alarm time, and if they are the same, to flash an alarm. By overriding the **timer1_Tick** method of **ctlClock**, and adding additional code to it, you will extend the capability of **ctlAlarmClock** while retaining all of the inherent functionality of **ctlClock**.

To override the timer1_Tick method of ctlClock

1. In the Code Editor, locate the `private bool blnAlarmSet;` statement. Immediately beneath it, add the following statement:

```
private bool blnColorTicker;
```

2. In the Code Editor, locate the `}` at the end of the class.
3. Just before the `}`, add the following code:

```
protected override void timer1_Tick(object sender, System.EventArgs e)
{
   // Calls the Timer1_Tick method of ctlClock.
   base.timer1_Tick(sender, e);
   // Checks to see if the Alarm is set.
   if (AlarmSet == false)
      return;
   else
```

```
        // If the date, hour and minute of the alarm time are the same as
        // now, flash!
    {
        if (AlarmTime.Date == DateTime.Now.Date && AlarmTime.Hour ==
           DateTime.Now.Hour && AlarmTime.Minute == DateTime.Now.Minute)
        {
           // Makes lblAlarmVisible, and changes the backcolor based on
           // the value of blnColorTicker. The backcolor of the label
           // will flash once per tick of the clock.
           lblAlarm.Visible = true;
           if (blnColorTicker == false)
           {
              lblAlarm.BackColor = Color.Red;
              blnColorTicker = true;
           }
           else
           {
              lblAlarm.BackColor = Color.Blue;
              blnColorTicker = false;
           }
        }
        else
        {
           // Once the alarm has sounded for a minute, the label is made
           // invisible again.
           lblAlarm.Visible = false;
        }
    }
}
```

The addition of this code accomplishes several tasks. The `override` statement directs the control to use this method in place of the method that was inherited from the base control. When this method is called, it calls the method it overrides by invoking the `base.timer1_Tick` statement, ensuring that all of the functionality incorporated in the original control is reproduced in this control. It then runs additional code to incorporate the alarm functionality. A flashing label control will appear when the alarm is triggered.

Your alarm clock control is almost complete. The only thing that remains is to implement a way to turn it off. To do this, we will add code to the **lblAlarm_Click** method.

To implement the shutoff method

1. In Solution Explorer, right-click **ctlAlarmClock.cs** and click **View Designer**.

   The designer opens.

2. Add a button to the control. Set the properties of the button as follows:

Property	Value
Name	**btnAlarmOff**
Text	**Disable Alarm**

3. In the designer, double-click **btnAlarmOff**.

   The Code Editor opens to the `private void btnAlarmOff_Click` line.

4. Modify this method so that it resembles the following:

```
private void btnAlarmOff_Click(object sender, System.EventArgs e)
{
   // Turns off the alarm
   AlarmSet = false;
   // Hides the flashing label
   lblAlarm.Visible = false;
}
```

5. From the **File** menu, choose **Save All** to save the project.

Testing Your Inherited Control

As with a standard user control, an inherited user control cannot stand alone and must be hosted in a form or other container. Since **ctlAlarmClock** has a greater depth of functionality, additional code is required to test it. In this section, you will write a simple program to test the functionality of **ctlAlarmClock**. You will write code to set and display the **AlarmTime** property of **ctlAlarmClock**, and will test its inherent functions.

To build and add your control to a test form

1. In Solution Explorer, click **ctlClockLib**. On the **Build** menu, choose **Build ctlClockLib**.
2. Add a new **Windows Application** project to the solution, and name it **Test2**.
3. In Solution Explorer, right-click the **References** node for your test project. Click **Add Reference** to display the **Add Reference** window. Click the tab labeled **Projects**. Your project will be listed under **Project Name**. Double-click your project, and note that it now appears in the **Selected Components** window.
4. In the Toolbox, click **My User Controls**.
5. Scroll down until the icon for **ctlAlarmClock** comes into view.
6. Double-click **ctlAlarmClock** to add a copy of **ctlAlarmClock** to your form.
7. In the Toolbox, locate and double-click **DateTimePicker** to add a **DateTimePicker** control to your form, and add a **Label** control by double-clicking **Label**.
8. Use the mouse to position the controls in a convenient place on the form.
9. Set the properties of these controls as follows:

Control	Property	Value
label1	**Text**	(leave blank)
	Name	**lblTest**
dateTimePicker1	**Name**	**dtpTest**
	Format	**Time**

10. In the designer, double-click **dtpTest**.

    The Code Editor opens to `private void dtpTest_ValueChanged`.

11. Modify the code so that it resembles the following:

```
private void dtpTest_ValueChanged(object sender, System.EventArgs e)
{
   ctlAlarmClock1.AlarmTime = dtpTest.Value;
   ctlAlarmClock1.AlarmSet = true;
   lblTest.Text = "Alarm Time is " +
      ctlAlarmClock1.AlarmTime.ToShortTimeString();
}
```

12. In Solution Explorer, right-click **Test2** and choose **Set as StartUp Project** from the shortcut menu.
13. From the **Debug** menu, choose **Start**.

    The test program starts. Note that the current time is updated in the **ctlAlarmClock** control, and that the starting time is shown in the **DateTimePicker** control.

14. Click the **DateTimePicker** where the minutes of the hour are displayed.
15. Using the keyboard, set a value for minutes that is one minute greater than the current time shown by **ctlAlarmClock**.

    The time for the alarm setting is shown in **lblTest**.

16. Wait for the displayed time to reach the alarm setting time.

    When the displayed time reaches the time to which the alarm is set the **lblAlarm** will flash. Turn off the alarm by clicking **btnAlarmOff**. You may now reset the alarm.

This walkthrough has covered a number of key concepts. You have learned to create a user control by combining controls and components into a user control container. You have learned to add properties to your control, and to write code to implement custom functionality. In the second section, you learned to extend the functionality of a given user control through inheritence,

and to alter the functionality of host methods by overriding those methods.

See Also

Programming with Components | Component Authoring Walkthroughs |
Displaying Your Control in the Customize Toolbox Dialog Box |
Walkthrough: Inheriting From a Windows Forms Control with Visual C#

Walkthrough: Inheriting from a Windows Forms Control with Visual Basic .NET

Visual Basic .NET provides you with the ability to create powerful custom controls through *inheritance*. Through inheritance you are able to create controls that retain all of the inherent functionality of standard Windows Forms controls but also incorporate custom functionality. In this walkthrough, you will create a simple inherited control called ValueButton. This button will inherit functionality from the standard Windows Forms button, and will expose a custom property called ButtonValue.

Creating the Project

When you create a new project, you specify its name in order to set the root namespace, assembly name, and project name, and to ensure that the default component will be in the correct namespace.

To create the ValueButtonLib control library and the ValueButton control

1. On the **File** menu, point to **New** and then click **Project** to open the **New Project** dialog box.
2. Select the **Windows Control Library** project template from the list of Visual Basic projects, and type **ValueButtonLib** in the **Name** box.

   > **Note** The Windows Control Library template is not available in the Standard Edition of Visual Basic .NET or Visual C# .NET. For more information, see Visual Basic Standard Edition Features or Visual C# Standard Edition Features.

   The project name, ValueButtonLib, is also assigned to the root namespace by default. The root namespace is used to qualify the names of components in the assembly. For example, if two assemblies provide components named ValueButton, you can specify your ValueButton component using `ValueButtonLib.ValueButton`. For more information, see Components and Assemblies.

3. In Solution Explorer, right-click **UserControl1.vb** and select **View Code** from the shortcut menu.
4. Locate the **Class** statement, `Public Class UserControl1`, and change **UserControl1** to **ValueButton** to change the name of the component.
5. Locate the statement `Inherits System.Windows.Forms.UserControl`, and change **System.Windows.Forms.UserControl** to **System.Windows.Forms.Button.** This allows your inherited control to inherit all the functionality of the **Button** control.
6. In Solution Explorer, click **UserControl1.vb** and in the Properties window, change the **FileName** property to **ValueButton.vb**.
7. From the **File** menu, choose **Save All** to save the project.

   Note that the User Control Designer is no longer available. Because the **Button** control does its own painting, you are unable to modify its appearance in the designer. Its visual representation will be exactly the same as that of the class it inherits from (that is, **Button**) unless modified in the code.

Adding a Property to Your Inherited Control

One possible use of inherited Windows Forms controls is the creation of controls that are identical in look and feel of standard Windows Forms controls, but expose custom properties. In this section, you will add a property called ButtonValue to your control.

To add the Value property

1. In Solution Explorer, right-click **ValueButton.vb**, and then click **View Code** from the shortcut menu.
2. Locate the `Inherits System.Windows.Forms.Button` statement. Immediately beneath this statement, type the following code:

```
' Creates the private variable that will store the value of your
' property.
Private varValue as integer
' Declares the property.
Property ButtonValue() as Integer
' Sets the method for retrieving the value of your property.
   Get
      Return varValue
```

```
    End Get
  ' Sets the method for setting the value of your property.
    Set(ByVal Value as Integer)
       varValue = Value
    End Set
  End Property
```

This code sets the methods by which the ButtonValue property is stored and retrieved. The **Get** statement sets the value returned to the value that is stored in the private variable **varValue**, and the **Set** statement sets the value of the private variable by use of the **Value** keyword.

3. From the **File** menu, choose **Save All** to save the project.

Testing Your Control

Controls are not stand-alone projects; they must be hosted in a container. In order to test your control, you must provide a test project for it to run in. You must also make your control accessible to the test project by building (compiling) it. In this section, you will build your control and test it in a Windows Form.

To build your control

- On the **Build** menu, click **Build**.

  The build should be successful with no compiler errors or warnings.

To create a test project

1. On the **File** menu, point to **Add Project** and then click **New Project** to open the **Add New Project** dialog box.
2. Select the Visual Basic projects node, and click **Windows Application**.
3. In the **Name** box, type **Test**.
4. In Solution Explorer, right-click the **References** node for your test project, then select **Add Reference** from the shortcut menu to display the **Add Reference** dialog box.
5. Click the **Projects** tab.
6. Double-click your project, and note that it now appears in the **Selected Components** pane of the dialog box.

After you have added your reference, you should add your new control to the Toolbox.

To add your control to the Toolbox

1. Right-click the Toolbox and choose **Add/Remove Items** from the shortcut menu.

   The **Customize Toolbox** dialog box opens.

2. Choose the **.NET Framework Components** tab and click **Browse**.
3. Browse to the ValueButtonLib\bin folder and select ValueButton.dll.

   ValueButton appears in the list of components in the **Customize Toolbox** dialog box.

4. In the **Customize Toolbox** dialog box, check the box next to ValueButton and close the window.

   The ValueButton is added to the tab of the Toolbox that was selected.

To add your control to the form

1. In Solution Explorer, right-click **Form1.vb** and choose **View Designer** from the shortcut menu.
2. In the Toolbox, scroll down until you come to the icon labeled **ValueButton**. Double-click this icon.

   A **ValueButton** appears on the form.

3. Right-click the **ValueButton** and select **Properties** from the shortcut menu.
4. In the Properties window, examine the properties of this control. Note that they are identical to the properties exposed by a standard button, except that there is an additional property, **ButtonValue**.
5. Set the **ButtonValue** property to **5**.
6. In the **Windows Forms** tab of the Toolbox, double-click **Label** to add a **Label** control to your form.
7. Relocate the label to the center of the form.

8. Double-click **ValueButton1**.

   The Code Editor opens to the **ValueButton1_Click** event.

9. Type the following line of code.

```
Label1.Text = CStr(ValueButton1.ButtonValue)
```

10. In Solution Explorer, right-click **Test**, and choose **Set as Startup Project** from the shortcut menu.
11. From the **Debug** menu, select **Start**.

    Form1 appears.

12. Click **Valuebutton1**.

    The numeral '5' is displayed in Label1, demonstrating that the ButtonValue property of your inherited control has been passed to Label1 via the ValueButton1_Click method. Thus your ValueButton control inherits all the functionality of the standard Windows Forms button, but exposes an additional, custom property.

See Also

Walkthrough: Authoring a User Control With Visual Basic .NET | Displaying Your Control in the Customize Toolbox Dialog Box | Developing Windows Forms Controls | Inheritance

Walkthrough: Inheriting from a Windows Forms Control with Visual C#

Visual C# .NET provides you with the ability to create powerful custom controls through *inheritance*. Through inheritance you are able to create controls that retain all of the inherent functionality of standard Windows Forms controls but also incorporate custom functionality. In this walkthrough, you will create a simple inherited control called ValueButton. This button will inherit functionality from the standard Windows Forms button, and will expose a custom property called ButtonValue.

Creating the Project

When you create a new project, you specify its name in order to set the root namespace, assembly name, and project name, and to ensure that the default component will be in the correct namespace.

To create the ValueButtonLib control library and the ValueButton control

1. On the **File** menu, point to **New** and then click **Project** to open the **New Project** dialog box.
2. Select the **Windows Control Library** project template from the list of C# Projects, and type **ValueButtonLib** in the **Name** box.

   > **Note** The Windows Control Library template is not available in the Standard Edition of Visual C# .NET. For more information, see Visual C# Standard Edition Features.

   The project name, ValueButtonLib, is also assigned to the root namespace by default. The root namespace is used to qualify the names of components in the assembly. For example, if two assemblies provide components named ValueButton, you can specify your ValueButton component using `ValueButtonLib.ValueButton`. For more information, see Components and Assemblies.

3. In Solution Explorer, right-click **UserControl1.cs** and select **View Code** from the shortcut menu.
4. Locate the **class** statement, `public class UserControl1`, and change **UserControl1** to **ValueButton** to change the name of the component.
5. Locate the Constructor, `public UserControl1()` and change it to `public ValueButton()`.
6. In the Class statement, change the type from which this control inherits from **System.Windows.Forms.UserControl** to **System.Windows.Forms.Button.** This allows your inherited control to inherit all the functionality of the **Button** control.
7. In Solution Explorer, click **UserControl1.cs** and in the Properties window, change the **FileName** property to **ValueButton.cs**.
8. From the **File** menu, choose **Save All** to save the project.

   Note that a designer is no longer available. Because the **Button** control does its own painting, you are unable to modify its appearance in the designer. Its visual representation will be exactly the same as that of the class it inherits from (that is, **Button**) unless modified in the code.

Adding a Property to Your Inherited Control

One possible use of inherited Windows Forms controls is the creation of controls that are identical in look and feel of standard Windows Forms controls, but expose custom properties. In this section, you will add a property called ButtonValue to your control.

To add the Value property

1. In Solution Explorer, right-click **ValueButton.cs**, and then click **View Code** from the shortcut menu.
2. Locate the **class** statement. Immediately after the {, type the following code:

```
// Creates the private variable that will store the value of your
// property.
private int varValue;
// Declares the property.
public int ButtonValue
{
   // Sets the method for retrieving the value of your property.
   get
   {
```

```
        return varValue;
    }
    // Sets the method for setting the value of your property.
    set
    {
        varValue = value;
    }
}
```

This code sets the methods by which the ButtonValue property is stored and retrieved. The **get** statement sets the value returned to the value that is stored in the private variable **varValue**, and the **set** statement sets the value of the private variable by use of the **value** keyword.

3. From the **File** menu, choose **Save All** to save the project.

Testing Your Control

Controls are not stand-alone projects; they must be hosted in a container. In order to test your control, you must provide a test project for it to run in. You must also make your control accessible to the test project by building (compiling) it. In this section, you will build your control and test it in a Windows Form.

To build your control

- On the **Build** menu, click **Build**.

  The build should be successful with no compiler errors or warnings.

To create a test project

1. On the **File** menu, point to **Add Project** and then click **New Project** to open the **Add New Project** dialog box.
2. Select the Visual C# projects node, and click Windows Application.
3. In the **Name** box, type **Test**.
4. In Solution Explorer, right-click the **References** node for your test project, then select **Add Reference** from the shortcut menu to display the **Add Reference** dialog box.
5. Click the tab labeled **Projects**.
6. Double-click the ValueButtonLib project, and note that it now appears in the **Selected Components** pane.

   After you have added your reference, you should add your new control to the toolbox. If your control already appears in the toolbox, you should skip the next section.

To add your control to the toolbox

1. Right-click the Toolbox and choose **Add/Remove Items** from the shortcut menu.

   The **Customize Toolbox** dialog box opens.

2. Choose the **.NET Framework Components** tab and click **Browse**. Browse to the ValueButtonLib\bin\debug folder and select ValueButton.dll.

   ValueButton appears in the list of components in the **Customize Toolbox** dialog box.

3. In the **Customize Toolbox** dialog box, check the box next to ValueButton and close the window.

   The ValueButton is added to the tab of the Toolbox that was selected.

To add your control to the form

1. In Solution Explorer, right-click **Form1.cs** and choose **View Designer** from the shortcut menu.
2. In Toolbox, scroll down until you come to the icon labeled **ValueButton**. Double-click this icon.

   A **ValueButton** appears on the form.

3. Right-click the **ValueButton** and select **Properties** from the shortcut menu.
4. In the Properties window, examine the properties of this control. Note that they are identical to the properties exposed by a standard button, except that there is an additional property, **ButtonValue**.
5. Set the **ButtonValue** property to **5**.

6. In the **Windows Forms** tab of the Toolbox, double-click **Label** to add a **Label** control to your form.
7. Relocate the label to the center of the form.
8. Double-click **valueButton1**.

   The Code Editor opens to the **valueButton1_Click** event.

9. Type the following line of code.

```
label1.Text = valueButton1.ButtonValue.ToString();
```

10. In Solution Explorer, right-click **Test**, and choose **Set as Startup Project** from the shortcut menu.
11. From the **Debug** menu, select **Start**.

    Form1 appears.

12. Click **valueButton1**.

The numeral '5' is displayed in Label1, demonstrating that the ButtonValue property of your inherited control has been passed to Label1 via the ValueButton1_Click method. Thus your ValueButton control inherits all the functionality of the standard Windows Forms button, but exposes an additional, custom property.

See Also

Programming with Components | Component Authoring Walkthroughs |
Displaying Your Control in the Customize Toolbox Dialog Box | Walkthrough: Authoring a User Control with Visual C#

Control Type Recommendations

The .NET Framework gives you power to develop and implement new controls. In addition to the familiar user control, you will now find that you are able to write custom controls that perform their own painting, and are even able to extend the functionality of existing controls through inheritance. Deciding which type of control to create can be confusing. This section highlights the differences between the various types of controls from which you can inherit, and gives considerations regarding the type to choose for your project.

> **Note** If you want to author a control to use on Web Forms, see Developing ASP.NET Server Controls.

Inheriting from a Windows Forms Control

You can derive an inherited control from any existing Windows Forms control. This approach allows you to retain all of the inherent functionality of a Windows Forms control, and to then extend that functionality by adding custom properties, methods, or other functionality. For example, you might create a control derived from **TextBox** that can accept only numbers and automatically converts input into a value. Such a control might contain validation code that was called whenever the text in the text box changed, and could have an additional property, Value. In some controls, you can also add a custom appearance to the graphical interface of your control by overriding the **OnPaint** method of the base class.

Inherit from a Windows Forms control if:

- Most of the functionality you need is already identical to an existing Windows Forms control.
- You do not need a custom graphical interface, or you want to design a new graphical front end for an existing control.

Inheriting from the UserControl Class

A user control is a collection of Windows Forms controls encapsulated into a common container. The container holds all of the inherent functionality associated with each of the Windows Forms controls and allows you to selectively expose and bind their properties. An example of a user control might be a control built to display customer address data from a database. This control would include several textboxes to display each field, and button controls to navigate through the records. Data-binding properties could be selectively exposed, and the entire control could be packaged and reused from application to application.

Inherit from the **UserControl** class if:

- You want to combine the functionality of several Windows Forms controls into a single reusable unit.

Inheriting from the Control Class

Another way to create a control is to create one substantially from scratch by inheriting from **Control**. The **Control** class provides all of the basic functionality required by controls (for example, events), but no control-specific functionality or graphical interface. Creating a control by inheriting from the **Control** class requires a lot more thought and effort than inheriting from user control or an existing Windows Forms control. The author must write code for the **OnPaint** event of the control, as well as any functionality specific code that is needed. Greater flexibility is allowed, however, and you can custom tailor a control to suit your exact needs. An example of a custom control is a clock control that duplicates the look and action of an analog clock. Custom painting would be invoked to cause the hands of the clock to move in response to **Tick** events from an internal timer component.

Inherit from the **Control** class if:

- You want to provide a custom graphical representation of your control.
- You need to implement custom functionality that is not available through standard controls.

See Also

Walkthrough: Authoring a User Control With Visual Basic .NET |
Walkthrough: Inheriting from a Windows Forms Control with Visual Basic .NET | Authoring Controls for Windows Forms

Authoring Controls for Windows Forms

A control represents a graphical link between the user and the program. A control can provide or process data, accept user input, respond to events, or perform any number of other functions that connect the user and the application. Because a control is essentially a component with a graphical interface, it can serve any function that a component does, as well as providing user interaction. Controls are created to serve specific purposes, and authoring controls is just another programming task. With that in mind, the following steps represent an overview of the control authoring process. Links provide additional information on the individual steps.

> **Note** If you want to author a custom control to use on Web Forms, see Developing ASP.NET Server Controls.

To author a control

1. Determine what you want your control to accomplish, or what part it will play in your application. Factors to consider are:
   - What kind of graphical interface do I need?
   - What specific user interactions will this control handle?
   - Is the functionality I need approached by any existing controls?
   - Can I get the functionality I need by combining several Windows Forms controls?
2. If you need an object model for your control, determine how functionality will be distributed throughout the object model, and divide up functionality between the control and any subobjects. An object model may be useful if you are planning a complex control, or wish to incorporate several functionalities.
3. Determine the type of control (for example, user control, custom control, inherited Windows Forms control) you need. For details, see Control Type Recommendations.
4. Express functionality as properties, methods, and events of the control and its subobjects or subsidiary structures, and assign appropriate access levels (public, protected, and so on). For more information, see Properties, Methods, and Events for Custom Controls.
5. If you need custom painting for your control, add code for it. For details, see Custom Control Painting and Rendering.
6. Create a new project (such as a Windows Application) so that you can test and debug your control.

   Another project is required because controls cannot be run as stand-alone projects. They must be placed into a container before they can be tested. This process is demonstrated as part of Walkthrough: Authoring a User Control With Visual Basic .NET.
7. As you add each feature, add features to your test project to exercise the new functionality.
8. Repeat, refining the design.
9. Package and deploy your control. For details, see Merge Module Projects.

See Also

Walkthrough: Authoring a User Control With Visual Basic .NET | Walkthrough: Inheriting from a Windows Forms Control with Visual Basic .NET | Authoring Controls for Windows Forms | Inheriting from the UserControl Class | Inheriting from the Control Class | Inheriting from Existing Windows Forms Controls

Authoring User Controls

User controls can be employed in many ways. You can author them as part of a Windows desktop application project, and use them only on forms in the project. Or you can author them in a Windows Control Library project, compile the project into an assembly, and use the controls in other projects. You can even inherit from them, and use visual inheritance to customize them quickly for special purposes.

> **Note** If you want to author a user control to use on Web Forms, see Developing ASP.NET Server Controls.

To author a user control

1. Open a new **Windows Application** project.
2. On the **Project** menu, click **Add User Control**.
3. Add controls from the **Toolbox** to the user control surface.
4. Place code in event procedures, to handle events raised by the UserControl or by its constituent controls.
5. Close the designer for the user control, and save the file.
6. Give the class file (.vb or .cs file) the name you want the user control to have.
7. On the **Build** menu, click **Build**.

   The project must be built in order for user controls to appear in the **Toolbox**.

8. Use the **Windows Forms** tab of the **Toolbox** to add instances of your control to Form1.

To author a control class library

1. Open a new **Windows Control Library** project.

   By default, the project contains a user control.

   > **Note** The Windows Control Library template is not available in the Standard Edition of Visual Basic and Visual C# .NET. For more information, see Visual Basic Standard Edition Features or Visual C# Standard Edition Features.

2. Add controls and code as described in the procedure above.
3. Choose a control you do not want inheriting classes to change, and set the **Modifiers** property of this control to **Private**.
4. Build the DLL.

To inherit from a user control in a control class library

1. Open a **Windows Application** project.
2. In **Solution Explorer**, right-click the **References** folder and choose **Add Reference** to open the **Add Reference** dialog box.
3. Select the **Projects** tab and double-click your DLL.
4. On the **Project** menu, click **Add Inherited Control**.
5. In the **Inheritance Picker** dialog box, double-click the control you want to inherit from.
6. In the Add New Item dialog box, click Open (Inherited User Control is already selected).
7. In the **Inheritance Picker** dialog box, browse to and double-click the control you want to inherit from.

   A new control is added to your project.

8. Open the visual designer for the new control and add additional constituent controls.

   You can see the constituent controls that were inherited from the user control in your DLL, and you can alter the properties of controls whose **Modifiers** property is **Public**. You cannot change the properties of the control whose **Modifiers** property is **Private**.

See Also

Walkthrough: Authoring a User Control With Visual Basic .NET | Walkthrough: Inheriting from a Windows Forms Control with Visual Basic .NET | Control Type Recommendations | Authoring Controls for Windows Forms

Inheriting from the UserControl Class

If you want to combine the functionality of one or more Windows Forms controls with custom code, then you should create a *user control*. User controls combine rapid control development with standard Windows Forms control functionality and the versatility created by adding custom properties and methods. When creating a user control, you are presented with a visible designer, upon which you may place standard Windows Forms controls. These controls retain all of their inherent functionality, as well as the look and feel of standard controls, but once they are built into the user control they are no longer available to the developer through code. The user control does its own painting and also handles all of the basic functionality associated with controls.

To create a user control

1. Create a new **Windows Control Library** project.

   A new project is created with a blank user control.

   > **Note** The Windows Control Library template is not available in the Standard Edition of Visual Basic and Visual C# .NET. For more information, see Visual Basic Standard Edition Features or Visual C# Standard Edition Features.

2. Drag controls from the Windows Forms tab of the Toolbox onto your designer.
3. These controls should be positioned and designed as you want them to appear in the final user control. If you want to allow developers to access the constituent controls, you must declare them as public, or selectively expose properties of the constituent control. For details, see Exposing Properties of Constituent Controls.
4. Implement any custom methods or properties that your control will incorporate. For details, see Properties, Methods, and Events for Custom Controls.
5. Save and test your control.

See Also

Inheriting from the Control Class | Inheriting from Existing Windows Forms Controls | Authoring Controls for Windows Forms | Troubleshooting Inherited Event Handlers in Visual Basic .NET

Inheriting from the Control Class

If you want to create a completely custom control to use on a Windows Form, the class to inherit from is **Control**. While inheriting from the **Control** class requires greater planning and implementation, it also makes available the largest range of options. When inheriting from **Control**, you inherit the very basic functionality that makes controls work. The functionality inherent in the **Control** class handles user input through the keyboard and mouse, defines the bounds and size of the control, provides a windows handle, and provides message handling and security. It does not incorporate any painting (the actual rendering of the graphical interface of the control), nor does it incorporate any specific user interaction functionality. The author must provide all of these aspects through custom code.

To create a custom control

1. Create a new **Windows Application** or **Windows Control Library** project.

   > **Note** The Windows Control Library template is not available in the Standard Edition of Visual Basic and Visual C# .NET. For more information, see Visual Basic Standard Edition Features or Visual C# Standard Edition Features.

2. From the **Project** menu, choose **Add Inherited Control**.

   The **Add New Item** dialog box appears.

3. In the **Add New Item** dialog box, click **Custom Control**.

   A new custom control is added to your project.

4. Open the Code Editor for your **CustomControl**.
5. Locate the **OnPaint** method, which will be empty except for a call to the **OnPaint** of the base class.
6. Modify the code to incorporate any custom painting you want for your control.

   For details on writing code to render graphics for controls, see Custom Control Painting and Rendering.

7. Implement any custom methods, properties, or events that your control will incorporate. For more information, see Properties, Methods, and Events for Custom Controls.
8. Save and test your control.

See Also

Inheriting from the UserControl Class | Inheriting from Existing Windows Forms Controls | Authoring Controls for Windows Forms | Troubleshooting Inherited Event Handlers in Visual Basic .NET

Inheriting from Existing Windows Forms Controls

If you want to extend the functionality of an existing control, you can create a control derived from an existing control through inheritance. When inheriting from an existing control, you inherit all of the functionality and visual properties of that control. For instance, if you were creating a control that inherited from **Button**, your new control would look and act exactly like a standard **Button** control. You could then extend or modify the functionality of your new control through the implementation of custom methods and properties. In some controls, you can also change the visual appearance of your inherited control by overriding its **OnPaint** method.

To create an inherited control

1. Create a new **Windows** project.

   This project can be of any type, such as a **Windows Application** project or a **Windows Control Library** project. If you choose a **Windows Control Library**, you may use the blank control provided and skip steps 2 and 3.

2. From the **Project** menu, choose **Add Inherited Control**.

   The **Add New Item** dialog box appears.

3. In the **Add New Item** dialog box, double-click **Custom Control**.

   A new custom control is added to your project.

4. In the Code Editor, locate the line that specifies **Control** as the base class to inherit from. Change the name of the base class to the name of the control from which you want to inherit.

   For example, if you want to inherit from **Button**, the line would read:

```
' Visual Basic
Inherits System.Windows.Forms.Button

// C#
public class CustomControl1 : System.Windows.Forms.Button
```

5. Implement any custom methods or properties that your control will incorporate. For further information see Properties, Methods, and Events for Custom Controls.
6. If you want to modify the graphical appearance of your control, override the **OnPaint** method.

   Note Overriding **OnPaint** will not allow you to modify the appearance of all controls. Those controls that have all of their painting done by Windows (for example, **Textbox**) never call their **OnPaint** method, and thus will never use the custom code. Please refer to the documentation of the particular control you wish to modify to see if the **OnPaint** method is available. If a control does not have **OnPaint** listed as a member method, then you cannot alter its appearance by overriding this method. For more information on custom painting, see Custom Control Painting and Rendering.

```
' Visual Basic
Protected Overrides Sub OnPaint(ByVal pe As _
   System.Windows.Forms.PaintEventArgs)
   MyBase.OnPaint(pe)
   ' Insert code to do custom painting. If you want to completely
   ' change the appearance of your control, do not call
   ' MyBase.OnPaint(pe).
End Sub

// C#
protected override void OnPaint(PaintEventArgs pe)
{
   // Do not call base.OnPaint if you want to completely
   // control the appearance of the control.
   base.OnPaint(pe);
```

```
    // Insert code to do custom painting.
  }
```

7. Save and test your control.

See Also

Inheriting from the Control Class | Inheriting from the UserControl Class | Authoring Controls for Windows Forms | Troubleshooting Inherited Event Handlers in Visual Basic .NET

Properties, Methods, and Events for Custom Controls

A custom control is any control that is created by the user, be it derived from the **UserControl** class, the **Control** class, or an existing Windows Forms control. Your control provides front-end functionality through its graphical interface, but provides back-end functionality through its methods, properties, and events. Methods are the actions a user can tell your control to execute. Properties and member variables are data a user of your control can get or set. Events are notifications that something interesting has happened to your component.

Adding properties and methods is easy — a method is any public procedure you declare in your control class; a property can be declared using a **Property** statement in Visual Basic, or implemented as a public member variable in C#.

The following topics examine the issues you should be aware of when declaring properties and methods in a control.

In This Section

Exposing Properties of Constituent Controls
Describes how to expose the properties of a constituent control to developers using your user control.

Method Implementation in Custom Controls
Provides conceptual information on implementing methods in your control and overloading methods.

Related Sections

Event Implementation for Components and Custom Controls with Visual Basic
Provides conceptual information for Visual Basic developers on events in your user control.

Developing ASP.NET Server Controls
Provides links to topics about how to create methods and properties for custom server controls that work with Web Forms.

Exposing Properties of Constituent Controls

The controls that make up a user control are called *constituent controls*. These controls are normally declared private, and thus cannot be accessed by the developer. If you want to make properties of these controls available to future users, you must expose them to the user. A property of a constituent control is exposed by creating a property in the user control, and using the **Get** and **Set** accessors of that property to effect the change in the private property of the constituent control.

Consider a hypothetical user control with a constituent button named MyButton. In this example, when the user requests the ConstituentButtonBackColor property, the value stored in the **BackColor** property of MyButton is delivered. When the user assigns a value to this property, that value is automatically passed to the **BackColor** property of MyButton and the **Set** code will execute, changing the color of MyButton.

The following example shows how to expose the **BackColor** property of the constituent button:

```
' Visual Basic
Public Property ButtonColor() as System.Drawing.Color
   Get
      Return MyButton.BackColor
   End Get
   Set(Value as System.Drawing.Color)
      MyButton.BackColor = Value
   End Set
End Property

// C#
public Color ButtonColor
{
   get
   {
      return(myButton.BackColor);
   }
   set
   {
      myButton.BackColor = value;
   }
}
```

To expose a property of a constituent control

1. In the Code Editor for your user control, create a public property.
2. In the **Get** section of the property, write code that retrieves the value of the property you want to expose.
3. In the **Set** section of the property, write code that passes the value of the property to the exposed property of the constituent control.

See Also

Walkthrough: Authoring a User Control With Visual Basic .NET

Method Implementation in Custom Controls

A method is implemented in a control in the same manner a method would be implemented in any other component.

In Visual Basic, if a method is required to return a value, it is implemented as a **Public Function**. If no value is returned, it is implemented as a **Public Sub**. Methods are declared using the following syntax:

```
' Visual Basic
Public Function ConvertMatterToEnergy(Matter as Integer) As Integer
   ' Conversion code goes here.
End Function
```

Because functions return a value, they must specify a return type, such as integer, string, object, and so on. The arguments **Function** or **Sub** procedures take, if any, must also be specified.

C# makes no distinction between functions and procedures, as Visual Basic does. A method either returns a value or returns void. The syntax for declaring a C# public method is:

```
// C#
public int ConvertMatterToEnergy(int matter)
{
   // Conversion code goes here.
}
```

When you declare a method, declare all of its arguments as explicit data types whenever possible. Arguments that take object references should be declared as specific class types — for example, **As Widget** instead of **As Object**. In Visual Basic, the default setting **Option Strict** automatically enforces this rule.

Typed arguments allow many developer errors to be caught by the compiler, rather than at run time. The compiler always catches errors, whereas run-time testing is only as good as the test suite.

Overloaded Methods

If you want to allow users of your control to supply different combinations of parameters to a method, provide multiple overloads of the method, using explicit data types. Avoid creating parameters declared **As Object** that can contain any data type, as this can lead to errors that might not be caught in testing.

> **Visual Basic Note** The universal data type in the common language runtime is **Object** rather than **Variant**. **Variant** has been removed from the language. For details, see Universal Data Type Changes in Visual Basic.

For example, the **Spin** method of a hypothetical Widget control might allow either direct specification of spin direction and speed, or specification of another Widget object from which angular momentum is to be absorbed:

```
' Visual Basic
Overloads Public Sub Spin( _
   ByVal SpinDirection As SpinDirectionsEnum, _
   ByVal RevolutionsPerSecond As Double)
   ' Implementation code here.
End Sub
Overloads Public Sub Spin(ByVal Driver As Widget) _
   ' Implementation code here.
End Sub

// C#
public void Spin(SpinDirectionsEnum spinDirection, double revolutionsPerSecond)
{
   // Implementation code here.
}

public void Spin(Widget driver)
{
   // Implementation code here.
}
```

See Also

Event Implementation for Components and Custom Controls with Visual Basic

Custom Control Painting and Rendering

Custom painting of controls is one of the many complicated tasks made easy by the .NET Framework. When authoring a custom control, you have many options regarding your control's graphical appearance. If you are authoring a control that inherits from the Control Class, you must provide code that allows your control to render its graphical representation. If you are creating a user control by inheriting from the UserControl Class, or are inheriting from one of the Windows Forms controls, you may override the standard graphical representation and provide your own graphics code. If you want to provide custom rendering for the constituent controls of a UserControl you are authoring, your options become more limited, but still allow a wide range of graphical possibilities for your controls and applications.

In this Section

User-Drawn Controls
Gives an overview of the steps involved in writing and overriding rendering code for your control.
Constituent Controls
Describes how to implement custom rendering code for constituent controls in your user controls and forms.

Related Sections

Creating Graphical Images with GDI+
Introduces GDI+ graphics functionality from a Visual Studio perspective and gives links to more information.
Control Authoring for Windows Forms
Provides links to topics about creating your own custom control.

User-Drawn Controls

The .NET Framework provides you with the ability to easily develop your own controls. You can create a user control, which is a set of standard controls bound together by code, or you can design your own control from the ground up. You can even use inheritance to create a control that inherits from an existing control and add to its inherent functionality. Whatever approach you use, the .NET Platform provides the functionality to draw a custom graphical interface for any control you create.

Painting of a control is accomplished by the execution of code in the control's Control.OnPaint Method. The single argument of the **OnPaint** method is a **PaintEventArgs** object that provides all of the information and functionality required to render your control. The **PaintEventArgs** provides as properties two principal objects that will be used in the rendering of your control:

- **ClipRectangle** object - the rectangle that represents the part of the control that will be drawn. This can be the entire control, or part of the control depending on how the control is drawn.
- **Graphics** object - encapsulates several graphics-oriented objects and methods that provide the functionality necessary to draw your control.

For more information on the **Graphics** object and how to use it, see Creating Graphical Images with GDI+.

The **OnPaint** event is fired whenever the control is drawn or refreshed on the screen, and the **ClipRectangle** object represents the rectangle in which painting will take place. If the entire control needs to be refreshed, the **ClipRectangle** will represent the size of the entire control. If only part of the control needs to be refreshed, however, the **ClipRectangle** object will represent only the region that needs to be redrawn. An example of such an example would be when a control was partially obscured by another control or form in the user interface.

When inheriting from the **Control** class, you must override the **OnPaint** method and provide graphics-rendering code within. If you want to provide a custom graphical interface to a user control or an inherited control, you can also do so by overriding the **OnPaint** method. An example is shown below:

```
' Visual Basic
Protected Overrides Sub OnPaint(ByVal pe As PaintEventArgs)
   ' Call the OnPaint method of the base class.
   MyBase.OnPaint(pe)

   ' Declare and instantiate a drawing pen.
   Dim myPen As System.Drawing.Pen = New System.Drawing.Pen(Color.Aqua)

   ' Draw an aqua rectangle in the rectangle represented by the control.
   pe.Graphics.DrawRectangle(myPen, New Rectangle(Me.Location, Me.Size))
End Sub

// C#
protected override void OnPaint(PaintEventArgs pe)
{
   // Call the OnPaint method of the base class.
   base.OnPaint(pe);

   // Declare and instantiate a new pen.
   System.Drawing.Pen myPen = new System.Drawing.Pen(Color.Aqua);

   // Draw an aqua rectangle in the rectangle represented by the control.
   pe.Graphics.DrawRectangle(myPen, new Rectangle(this.Location,
      this.Size));
}
```

The preceding example demonstrates how to render a control with a very simple graphical representation. It calls the **OnPaint** method of the base class, it creates a **Pen** object with which to draw, and finally draws an ellipse in the rectangle determined by the **Location** and **Size** of the control. Although most rendering code will be significantly more complicated than this, this example demonstrates the use of the **Graphics** object contained within the **PaintEventArgs** object. Note that if you are inheriting from a class that already has a graphical representation, such as **UserControl** or **Button**, and you do not wish to incorporate that representation into your rendering, you should not call your base class's **OnPaint** method.

The code in the **OnPaint** method of your control will execute when the control is first drawn, and whenever it is refreshed. To ensure that your control is redrawn every time it is resized, add the following line to the constructor of your control:

```
' Visual Basic
SetStyle(ControlStyles.ResizeRedraw, True)

//C#
SetStyle(ControlStyles.ResizeRedraw, true);
```

See Also

Creating Graphical Images with GDI+ | Constituent Controls | ControlStyles Enumeration

Constituent Controls

The controls that make up a user control, or *constituent controls* as they are termed, are relatively inflexible when it comes to custom graphics rendering. All Windows Forms controls handle their own rendering through their own **OnPaint** method. Because this method is protected, it is not accessible to the developer, and thus cannot be prevented from executing when the control is painted. This does not mean, however, that you cannot add code to affect the appearance of constituent controls. Additional rendering can be accomplished by adding an event handler. For example, suppose you were authoring a UserControl with a button named MyButton. If you wished to have additional rendering beyond what was provided by the Button Class, you would add code to your user control similar to the following:

```
' Visual Basic
Public Sub MyPaint(ByVal sender as Object, e as PaintEventArgs) Handles _
   MyButton.Paint
   'Additional rendering code goes here
End Sub

// C#
// Add the event handler to the button's Paint event.
MyButton.Paint +=
   new System.Windows.Forms.PaintEventHandler (this.MyPaint);
// Create the custom painting method.
protected void MyPaint (object sender,
System.Windows.Forms.PaintEventArgs e)
{
   // Additional rendering code goes here.
}
```

> **Note** Some Windows Forms controls, such as **TextBox**, are painted directly by Windows. In these instances, the **OnPaint** method is never called, and thus the above example will never be called.

This creates a method that executes every time the MyButton.Paint event executes, thereby adding additional graphical representation to your control. Note that this does not prevent the execution of MyButton.OnPaint, and thus all of the painting usually performed by a button will still be performed in addition to your custom painting. For details about GDI+ technology and custom rendering, see the Creating Graphical Images with GDI+. If you wish to have a unique representation of your control, your best course of action is to create an inherited control, and to write custom rendering code for it. For details, see User-Drawn Controls.

See Also

User-Drawn Controls | Creating Graphical Images with GDI+ |
Walkthrough: Inheriting from a Windows Forms Control with Visual Basic .NET

Making Your Control Invisible at Run Time

There are times when you might want to create a user control that is invisible at run time. For example, a control that is an alarm clock might be invisible except when the alarm was sounding. This is easily accomplished by setting the **Visible** property. If the **Visible** property is **true**, your control will appear as normal. If **false**, your control will be hidden. Although code in your control may still run while invisible, you will not be able to interact with the control through the user interface. If you want to create an invisible control that still responds to user input (for example, mouse clicks), you should create a transparent control. For more information, see Giving Your Control a Transparent Background.

To make your control invisible at run time

- Set the **Visible** property to **false**.

```
' Visual Basic
' To set the Visible property from within your object's own code.
Me.Visible = False
' To set the Visible property from another object.
myControl1.Visible = False

// C#
// To set the Visible property from within your objects own code.
this.Visible = false;
// To set the Visible property from another object.
myControl1.Visible = false;
```

See Also

Control Authoring for Windows Forms | Giving Your Control a Transparent Background

Aligning Your Control to the Edges of Forms

You can make your control align to the edge of your forms by setting the Dock Property. This property designates where your control resides in the form. The **Dock** property can be set to the following values:

Setting	Effect on your control
DockStyle.Bottom	Docks to the bottom of the form.
DockStyle.Fill	Fills all remaining space in the form.
DockStyle.Left	Docks to the left side of the form.
DockStyle.None	Does not dock anywhere, and it appears at the location specified by its Location Property.
DockStyle.Right	Docks to the right side of the form.
DockStyle.Top	Docks to the top of the form.

These values can be set in code, but they are usually set at design time in the Properties window.

To set the Dock property for your control at run time

- Set the **Dock** property to the appropriate value in code.

```
' Visual Basic
' To set the Dock property internally.
Me.Dock = DockStyle.Top
' To set the Dock property from another object.
UserControl1.Dock = DockStyle.Top

// C#
// To set the Dock property internally.
this.Dock = DockStyle.Top;
// To set the Dock property from another object.
UserControl1.Dock = DockStyle.Top;
```

To set the Dock property for your control at design time

1. In the designer, select your control.
2. In the Properties window, click the drop-down box next to the **Dock** property.

   A graphical interface representing the six possible **Dock** settings is displayed.

3. Choose the appropriate setting.

   Your control will now dock in the manner specified by the setting.

See Also

Control Authoring for Windows Forms | Control.Anchor Property

Providing a Toolbox Bitmap for Your Control

You might want to have a special icon for your control appear in the Toolbox. You can specify a particular image by using the ToolboxBitmapAttribute Class. This class is an *attribute*, a special kind of class you can attach to other classes. For more information on attributes, see Attributes Overview for Visual Basic and Introduction to Attributes for C#.

Using the **ToolboxBitmapAttribute**, you can specify a **String** that indicates the path and file name for a 16 by 16 pixel bitmap. This bitmap then appears next to your control when added to the Toolbox. You can also specify a **Type**, in which case the bitmap associated with that type is loaded. If you specify both a **Type** and a **String**, the control searches for an image resource with the name specified by the **String** parameter in the assembly containing the type specified by the **Type** parameter.

To specify a Toolbox bitmap for your control

- Add the **ToolBoxBitmapAttribute** to the class declaration of your control before the **Class** keyword for Visual Basic, and above the class declaration for Visual C#.

```
' Visual Basic
' Specifies the bitmap associated with the Button type.
<ToolboxBitmap(GetType(Button))> Class MyControl1
' Specifies a bitmap file.
End Class
<ToolboxBitmap("C:\Documents and Settings\Joe\MyPics\myImage.bmp")> _
   Class MyControl2
End Class
' Specifies a type that indicates the assembly to search, and the name
' of an image resource to look for.
<ToolboxBitmap(GetType(MyControl), "MyControlBitmap")> Class MyControl
End Class

// C#
// Specifies the bitmap associated with the Button type.
[ToolboxBitmap(typeof(Button))]
class MyControl1 : UserControl
{
}
// Specifies a bitmap file.
[ToolboxBitmap(@"C:\Documents and Settings\Joe\MyPics\myImage.bmp")]
class MyControl2 : UserControl
{
}
// Specifies a type that indicates the assembly to search, and the name
// of an image resource to look for.
[ToolboxBitmap(typeof(MyControl), "MyControlBitmap")]
class MyControl : UserControl
{
}
```

See Also

Control Authoring for Windows Forms | Visual Basic .NET Attributes | Introduction to Attributes

Extender Provider Objects

An *extender provider* is a component that provides properties to other components. For example, when a ToolTip Component (Windows Forms) component is added to a form, it provides a property called **ToolTip** to each control on that form. The **ToolTip** property then appears in the Properties window for each control and allows the developer to set a value for this property at design time.

The property provided by the extender provider actually resides in the extender provider object itself and therefore is not a true property of the component it modifies. At design time, the property will appear in the Properties window for the component that is being modified. At run time, however, the property cannot be accessed through the component itself. In the following code example, a form has been created with a button called MyButton and a **ToolTip** control called MyToolTip, which provides a **ToolTip** property.

```
' Visual Basic
' This is an example of code that is NOT CORRECT!
Dim myString as String
myString = MyButton.ToolTip

// C#
// This is an example of code that is NOT CORRECT!
string myString;
myString = MyButton.ToolTip;
```

This syntax will generate a compile error because the compiler does not recognize **ToolTip** as a property of MyButton, because the property is actually provided by MyToolTip. The following example shows how to correctly access that property:

```
' Visual Basic
Dim myString as String
myString = MyToolTip.GetToolTip(MyButton)

// C#
string myString;
myString = MyToolTip.GetToolTip(MyButton);
```

The extender provider is a class, and as such can have its own properties and methods. In order to designate a property as the property to be provided to other components, you apply the ProvidePropertyAttribute attribute at the class level. This attribute specifies the name of the property to provide and the type of object it can provide that property to. By convention, the property provided is not implemented as a property, but rather as a pair of methods. These methods must have "Get" and "Set" added to the beginning of the name of the property to be provided. The following example shows how.

```
' Visual Basic
Imports System.ComponentModel
<ProvideProperty("MyText", GetType(Control))> Public Class MyExtender
   <ExtenderProvidedProperty()> Public Function GetMyText(acontrol as _
      Control) as String
      ' Insert code to implement function.
   End Function
   Public Sub SetMytext (acontrol as Control)
      ' Insert code to implement function.
   End Function
End Class

// C#
using System.ComponentModel;
[ProvideProperty("MyText", typeof("Control"))]
public class MyExtender
{
[ExtenderProvidedProperty()]
   public string GetMyText(Control acontrol)
   {
      // Insert code to implement method.
   }
   public void SetMyText(Control acontrol)
```

```
    {
       // Insert code to implement method.
    }
}
```

The implementation of the provided property will require a **Hashtable** or some other such collection object to record and retrieve the property values for each control. For details, see Implementing an Extender Provider.

Every extender class must also implement the IExtenderProvider Interface. That interface consists of a single method, **CanExtend**, which returns a Boolean value and indicates to the designer whether a component is a candidate to be extended or not. For example, you might want to create an extender that provides a property only to controls. The following example shows how to implement the **CanExtend** method:

```
' Visual Basic
Imports System.ComponentModel
Public Function CanExtend(ByVal extendee As Object) As Boolean _
   Implements IExtenderProvider.CanExtend
   If Typeof extendee Is Control Then
      Return True
   Else
      Return False
   End If
End Function

// C#
public bool CanExtend(object extendee)
{
   if (extendee is Control)
      return true;
   else
      return false;
}
```

See Also

IExtenderProvider Interface | Implementing an Extender Provider

Giving Your Control a Transparent Background

By default, controls do not support transparent backcolors. However, you can allow your control to have a background color that is opaque, transparent, or partially transparent by using the Control.SetStyle Method in the constructor. The **SetStyle** method of the **Control** class allows you to set particular style preferences for your controls, and can be used to enable or disable support for transparent backcolors.

To give your control a transparent backcolor

1. In the Code Editor for your control, locate the constructor.
2. Call the **SetStyle** method of your form in the constructor.

   ```
   ' Visual Basic
   SetStyle(ControlStyles.SupportsTransparentBackColor, True)

   // C#
   SetStyle(ControlStyles.SupportsTransparentBackColor, true);
   ```

   This will enable your control to support a transparent backcolor.

3. Beneath the line of code you added in step 1, add the following line. This will set your control's **BackColor** to **Transparent**.

   ```
   ' Visual Basic
   Me.BackColor = Color.Transparent

   // C#
   this.BackColor = Color.Transparent;
   ```

   Note that you can also create colors that are partially transparent using the **Color.FromArgb** method. For more information on colors, see Pens, Brushes, and Colors.

See Also

Control Authoring for Windows Forms | Pens, Brushes, and Colors

Displaying Your Control in the Customize Toolbox Dialog Box

As you develop and distribute controls, you might want those controls to appear in the **Customize Toolbox** dialog box, which is displayed when you customize the toolbox. You can enable your control to appear in this dialog box by either setting the appropriate registry key value or by installing your control to the Global Assembly Cache.

To display your control in the Customize Toolbox dialog box

- Install your control assembly to the Global Assembly Cache. For additional information, see Installing an Assembly into the Global Assembly Cache

  -or-

- Use the Windows utility **Regedit.exe** to create a registry key at HKEY_LOCAL_MACHINE\Software\Microsoft\Visual Studio\7.1\AssemblyFolders\<KeyName> and set the Default string value to the file location where your assembly is found. <KeyName> can be any value you prefer.

  > **Note** In Visual Studio .NET 2002, this key should be created at HKEY_LOCAL_MACHINE\Software\Microsoft\.NET\AssemblyFolders\<KeyName>.

See Also

Control Authoring for Windows Forms | Installing an Assembly into the Global Assembly Cache

Private Communication Between Components in Assemblies

At times, you might want to create components in an assembly that are exposed to other members of your assembly, but not to clients. For example, you might have an assembly that exposes Machine and Mechanism components to the user. Each of these components might in turn require Widget components, but the client application would have no need for a Widget component on its own.

A public component is accessible to all users of an assembly, including the client, but a private component is accessible only to other members of its namespace. Neither of these access modifiers allows the level of access that you require.

The solution is to use **Friend** (**internal** in C#) components. If a component is marked as **Friend** (**internal**), it can be freely accessed and instantiated by all other components in your assembly, but not by any users outside your assembly. The following example shows how to use the **Friend** (**internal**) access modifier.

```
' Visual Basic
Namespace Widgets
   ' This class will be accessible to other classes in the assembly,
   ' but not to client users.
   Friend Class Widget
      ' Insert code to implement the Widget class.
   End Class
End Namespace

' This namespace contains a class that will use a class in the Widgets
' namespace.
Namespace Mechanisms
   ' This class is accessible to client users, and can itself access the
   ' Widget class to create and contain instances of it.
   Public Class Mechanism
      Dim myWidget as New Widgets.Widget
      ' Insert code to implement additional functionality.
   End Class
End Namespace

// C#
namespace Widgets
{
   // This class will be accessible to other classes in the assembly,
   // but not to client users.
   internal class Widget
   {
      // Insert code to implement the Widget class.
   }
}

// The Mechanisms namespace declares Mechanism class that will contain an
// instance of the Widgets.Widget class.
namespace Mechanisms
{
   // This class is accessible to client users, and can itself access the
   // Widget class to create and contain instances of it.
   public class Mechanism
   {
      Widgets.Widget widget = new Widgets.Widget();
   }
}
```

Access Levels for Nested Classes

When assigning access levels to nested classes, there are additional considerations. For details, see Nested Classes in Components.

Accessing Members of Friend (internal) Components

Members of **Friend** (**internal**) components can be accessed according to the same access rules as members of a public

component, provided that the caller has access to the class. For example, a component named Mechanism that is a member of the same assembly as **Friend** (**internal**) component Widget can call the public method **Spin** of Widget, but an external user cannot. In practice, all public members of a **Friend** (**internal**) class have **Friend** (**internal**) access levels. Private members are accessible only from within the class itself.

Accessing Friend (internal) Members of Public Components

You might want to make methods or properties of a public component accessible only to members of your assembly. For example, a public Mechanism class might have a method named AddWidget. Any member of the assembly could invoke this method, but it would not make sense to allow the client to invoke it. In such a case, use the **Friend** (**internal**) access modifier on the method itself, and keep the access modifier on the class public.

> **Note** In the example below, the method AddWidget is accessible to all members of the assembly, but is not added to the public interface and cannot be called by client applications.

```
' Visual Basic
Public Class Mechanism
   Friend Sub AddWidget()
      ' Insert code to implement method.
   End Sub
   ' Insert code to implement the rest of the class.
End Class

// C#
public class Mechanism
{
   internal void AddWidget()
   {
      // Insert code to implement method.
   }
   // Insert code to implement the rest of the class.
}
```

See Also

internal (C#) | 4.6 Access Types (Visual Basic) | 3.5.2 Accessibility domains (C#) | Component Authoring

COM Interoperability in Visual Basic and Visual C#

When you want to use COM objects and .NET objects in the same application, you need to address the differences in how the objects exist in memory. A .NET object is located in managed memory — the memory controlled by the common language runtime — and may be moved by the runtime as needed. A COM object is located in unmanaged memory and is not expected to move to another memory location. Visual Studio and the .NET Framework provide tools to control the interaction of these managed and unmanaged components. For more information about managed code, see Common Language Runtime.

In addition to using COM objects in .NET applications, you may also want to continue to develop COM objects using Visual Basic .NET or Visual C# .NET.

The links on this page provide information about the concepts and implementation details for interoperating between COM and .NET objects.

In the Visual Basic and Visual C# Documentation

COM Interop
: Provides links to topics covering COM interop in Visual Basic, including COM objects, ActiveX controls, Win32 DLLs, managed objects, and inheritance of COM objects.

COM Interop Part 1: C# Client Tutorial
: Shows how to use Visual C# to interoperate with COM objects.

COM Interop Part 2: C# Server Tutorial
: Describes using a Visual C# server with a C++ COM client.

COM Interop Wrapper Error
: Describes the consequences and options if the project system cannot create a COM interop wrapper for a particular component.

Additional Information

Interoperating with Unmanaged Code
: Briefly describes some of the interaction issues between managed and unmanaged code, and provides links for further study.

COM Wrappers
: Discusses runtime callable wrappers, which allow managed code to call COM methods, and COM callable wrappers, which allow COM clients to call .NET object methods.

Advanced COM Interop
: Provides links to topics covering COM interop with respect to wrappers, exceptions, inheritance, threading, events, conversions, and marshaling.

Automating Office Applications
: Provides information on using Microsoft Office and Visual Studio .NET as part of your business application.

Type Library Importer (Tlbimp.exe)
: Discusses the tool you can use to convert the type definitions found within a COM type library into equivalent definitions in a common language runtime assembly.

Creating Messaging Components

The **MessageQueue** component allows you to build complex messaging into your applications. Using **MessageQueue** component instances, you can easily send messages to and receive messages from any queue in your network.

XML Web services provide another means for incorporating messaging-type behavior into your applications. Using XML Web services, you can pass information back and forth between a Web application and any client.

The **MessageQueue** component allows you to easily incorporate message-based communication into your applications. Using this component and its associated language features, you can send and receive messages, explore existing queues, create and delete queues, and perform a variety of other operations using a simple programming model.

In This Section

Introduction to Messaging
Provides basic information about messaging features.
Message Queues and Messaging Technology Backgrounder
Explains the essential messaging concepts and provides links to more information.
MessageQueue Programming Architecture
Describes the language elements used in **MessageQueue** component programming.
Creating, Deleting, and Administering Queues
Recommends additional topics on finding, creating, and deleting message queues in Server Explorer.
Sending and Serializing Messages
Recommends additional topics on sending simple and complex messages to a variety of queue types, requesting acknowledgement, using journals, and sending datasets in messages.
Reading and Receiving Messages
Recommends additional topics on reading messages procedurally, handling asynchronous operations, and peeking at messages in a queue.
Message Queue Security
Provides an overview of encrypting and authenticating the messages you send and receive.
Troubleshooting MessageQueue Components
Explains how to resolve common problems with this component.

Related Sections

Creating Installation Components
Describes how to use installation components to configure the system resources needed by your **MessageQueue** component.
Programming with Components
Presents a high-level overview of component programming and authoring.
System.Messaging Namespace
Describes the major features of the **System.Messaging** namespace
Visual Basic and Visual C# Projects
Describes the projects types used in this chapter and how to choose between them.
Deploying Applications and Components
Lists the major deployment topics and the information they contain.

Introduction to Messaging

Microsoft Windows Message Queuing makes it easy for application developers to communicate with application programs quickly and reliably by sending and receiving messages. Messaging provides you with guaranteed message delivery and a robust, fail-safe way to carry out many of your business processes. For example, suppose you have a retail point-of-sale application that must run 24 hours a day, seven days a week. If the database system behind the application goes down your sales staff might need to start taking orders manually. Using Message Queuing, you can set up the system so that the orders that cannot be processed during the downtime are automatically put into a queue and processed as soon as the database comes back up.

In Microsoft Visual Studio .NET and the Microsoft .NET Framework SDK, you can use an instance of the **MessageQueue** component to quickly and easily connect to existing message queues, examine their contents, and send and receive messages. You can also use Server Explorer to view message queues on any server to which you have access, and to view information about the messages contained in these queues. You can add a queue from Server Explorer to your component's designer to automatically create a component that is configured to interact with the queue.

> **Note** The Servers node of Server Explorer is not available in the Standard Edition of Visual Basic and Visual C# .NET. For more information, see Visual Basic Standard Edition Features or Visual C# Standard Edition Features.

Interacting with Message Queues

Messaging and messages provide a powerful and flexible mechanism for interprocess communication between components of a server-based application. They have a number of advantages over direct calls between components, including:

- **Robustness** — Messages are considerably less affected by component failures than direct calls between components, because messages are stored in queues and remain there until processed appropriately. Messaging is similar to transaction processing, because message processing is guaranteed.
- **Message prioritization** — More urgent or important messages can be received before less important messages, so you can guarantee adequate response time for critical applications.
- **Offline capabilities** — Messages can be sent to temporary queues when they are sent and remain that way until they are successfully delivered. Users can continue to perform operations when access to the necessary queue is unavailable for whatever reason. In the meantime, additional operations can proceed as if the message had already been processed, because the message delivery is guaranteed when the network connection is restored.
- **Transactional messaging** — You can couple several related messages into a single transaction, ensuring that the messages are delivered in order, delivered only once, and are successfully retrieved from their destination queue. If any errors occur, the entire transaction is cancelled.
- **Security** — The Message Queuing technology on which the **MessageQueue** component is based uses Windows security to secure access control, provide auditing, and encrypt and authenticate the messages your component sends and receives.

Restrictions

Your system must meet the following requirements to develop **MessageQueue** components:

- To see queue information in Server Explorer or to access queues programmatically, you must install Message Queuing on your client computer.

  > **Note** When you install Message Queuing, you select the primary Message Queuing server to which you want to connect. You cannot view or manipulate queues that reside on another primary Message Queuing server.

- Message Queuing can be run in either a domain or a workgroup environment. In the context of Message Queuing, a domain environment includes domain controllers that provide a directory service such as Active Directory, and a workgroup environment is any environment that does not provide such a directory service.

See Also

Message Queues and Messaging Technology Backgrounder | Creating MessageQueue Component Instances

Message Queues and Messaging Technology Backgrounder

Microsoft Windows Message Queuing makes it easy for application developers to communicate with application programs quickly and reliably by sending and receiving messages. If you are unfamiliar with **MessageQueue** technology, this topic provides a brief overview of essential terms and concepts.

Key Concepts in Messaging

A *message* is a unit of data sent between two computers. A message can be very simple, consisting of just a string of text, or more complex, possibly involving embedded objects.

Messages are sent to queues. A *message queue* is a container that holds messages while they are in transit. The message queue manager acts as the middleman in relaying a message from its source to its destination. A queue's main purpose is to provide routing and guarantee the delivery of messages; if the recipient is not available when a message is sent, the queue holds the message until it can be successfully delivered.

Message Queuing, the Microsoft messaging technology, provides messaging and message queue facilities for any application and among any combination of computers that have Microsoft Windows installed, regardless of whether they are on the same network or online at the same time.

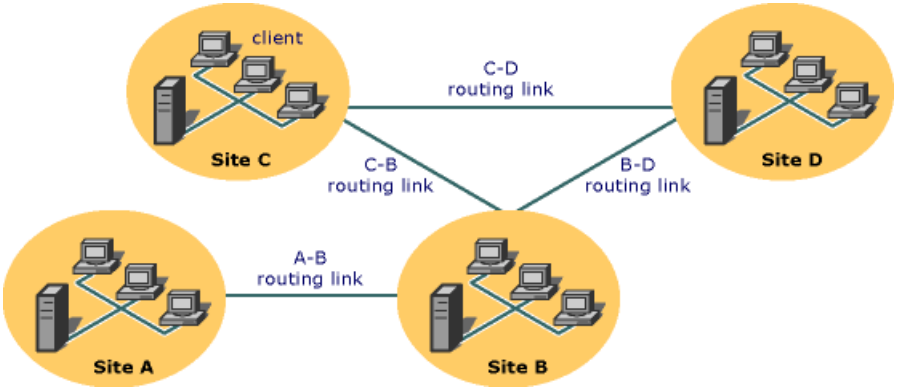
A *Message Queuing network* is any set of computers that are enabled to send messages back and forth to one another. Different computers in the network play different roles in ensuring that messaging proceeds smoothly. Some provide routing information to determine how messages are sent, some hold key information for the entire network, and some simply send and receive messages.

During Message Queuing setup, an administrator makes decisions about which servers can communicate with each other and sets up special roles for specific servers. The computers that make up this Message Queuing network are called *sites*, and they are connected to one another by *site links*. Each site link has an associated *cost*, determined by the administrator, that indicates how quickly messages can be passed across it.

The Message Queuing administrator also sets up one or more computers in the network to act as *routing servers*. A routing server makes decisions about how a message is delivered by looking at the cost of various site links and determining the quickest and most efficient way to deliver the message across multiple sites.

The following picture shows a typical configuration of Message Queuing sites and how they interact:

Message routing between sites



> **Tip** You do not need to worry about the details of message routing during development, because your Message Queuing network can be tuned by administrators for efficient message delivery.

Types of Queues

There are two main categories of queues — queues that you or other users of the network create, and system queues. User-created queues can be any of the following:

- *Public queues* are replicated throughout the Message Queuing network and can potentially be accessed by all of the sites connected by the network.
- *Private queues* are not published across the entire network. Instead they are available only on the local computer that contains them. Private queues can be accessed only by applications that know the full path name or label of the queue.
- *Administration queues* contain messages acknowledging the receipt of messages sent within a given Message Queuing network. You specify the administration queue you want your **MessageQueue** components to use, if any.

- *Response queues* contain response messages that are returned to the sending application when the message is received by the destination application. You specify the response queue you want your **MessageQueue** components to use, if any.

System-generated queues generally fall in one of the following categories:

- *Journal queues* optionally store copies of messages that you send and copies of messages removed from a queue. A single journal queue on each Message Queuing client stores copies of messages sent from that computer. On the server, a separate journal queue is created for each individual queue. This journal tracks messages removed from that queue.
- *Dead-letter queues* store copies of undeliverable or expired messages. If the expired or undeliverable message was a transactional message, it is stored in a special kind of dead-letter queue called a *transaction dead-letter queue*. Dead letters are stored on the computer on which the message expired. For more information on time-out periods and expired messages, see Default Message Properties.
- *Report queues* contain messages that indicate the route a message took to its destination, and can also contain test messages. There can be only one report queue per computer.
- *Private system queues* are a series of private queues that store administrative and notification messages that the system needs to process messaging actions.

Most of the work you do in your applications will involve accessing public queues and their messages. However, you will most likely use several different kinds of the system queues in your day-to-day operations, depending on your application's need for journal recording, acknowledgement, and other special processing.

Synchronous and Asynchronous Communication

Queue communication is inherently *asynchronous* in that messages are sent to a queue and received from a queue in separate processes. In addition, you can perform receive operations asynchronously in that the person who wants to receive a message can call the **BeginReceive** method for any given queue and then immediately go on to other tasks without waiting for a reply. This is very different from what is known as *synchronous communication*.

In synchronous communication, the sender of a request must wait for a response from the intended receiver before performing other tasks. The amount of time that the sender must wait depends entirely on the amount of time it takes for the receiver to process the request and send a response.

> **Note** Synchronous or asynchronous retrieval of messages is a separate issue from sending messages. Messages are always sent asynchronously.

For more information on asynchronous processing in message queues, see Asynchronous Message Processing.

Message Security

You can help secure the content of messages that you send and receive in the following ways:

- You can use authentication to verify the source of messages your applications receive.
- You can use encryption to ensure that your messages cannot be read or used by unauthorized persons.
- You can use access control rights to restrict users with ACL-based security and code with code access security from sending messages to, or reading messages from, a particular queue on a computer.
- You can use auditing to record which users attempt to access Message Queuing objects, the type of operation attempted, and if that access succeeded or failed.

For more information, see Message Queue Security.

Installing Message Queuing

You must have Message Queuing installed on your development computer if you want to work with message queues either in code or in Server Explorer. You can find instructions on installing Message Queuing in your Windows 2000 or Windows NT documentation. When you install, you must specify whether you are creating a new primary Message Queuing server that will act as the hub of a new network, an independent client of an existing network, or a dependent client of an existing network. In most cases you will not create a new network, but will join the existing Message Queuing network for your company or workgroup.

Where to Go for Background Reading

The remaining topics in this section provide you with the information you need to use Message Queuing components in your applications. If you need more information on messaging technologies outside the context of Visual Studio or the .NET framework, there are many sources of information available to you in the MSDN library and on the Microsoft Web site:

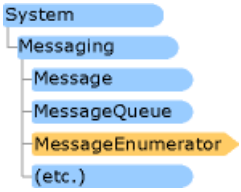
For information about	See this topic in your Windows 2000 Server documentation
Setting up a Message Queuing network	Installing Message Queuing
General Message Queuing concepts	Understanding Message Queuing
Fine-tuning Message Queuing performance	Administering Message Queuing

See Also

Creating Messaging Components

MessageQueue Programming Architecture

The **MessageQueue** component uses these portions of the Microsoft .NET Framework namespaces:



When you add an instance of the **MessageQueue** component to your Visual Studio project, the system automatically creates the references and import statements you need to access these namespaces and classes. However, if you are creating your **MessageQueue** components in code or in the .NET Framework SDK, you might need to create a reference to **System.Messaging** and add a statement to your project importing (in Visual Basic) or using (in C#) that namespace.

For instructions on adding and removing project references, see Adding and Removing References.

Key Programming Elements in the MessageQueue Base Class

You can use the following methods to interact with an instance of the **MessageQueue** component:

- Use the Create method to create a new message queue using the path you specify, and use the Delete method to delete an existing queue.
- Use the Exists method to see whether a particular message queue exists.
- Use the GetPublicQueues method to locate message queues in your Message Queuing network.
- Use the Peek or BeginPeek method to look at messages in a particular queue without removing the messages from the queue.
- Use the Receive and BeginReceive methods to retrieve the message at the front of the specified queue and remove it from the queue.
- Use the Send method to send a message to the specified queue.

You can view details about your Message Queuing installation by using Server Explorer to look at the messaging server. For more information, see Finding Queues in Server Explorer. You can also get detailed information about the configuration of your message queue network by using Message Queuing Explorer, which is installed automatically with Message Queuing.

> **Note** The Servers node of Server Explorer is not available in the Standard Edition of Visual Basic and Visual C# .NET. For more information, see Visual Basic Standard Edition Features or Visual C# Standard Edition Features.

See Also

Creating Messaging Components | Adding and Removing References | Creating MessageQueue Component Instances

Creating, Deleting, and Administering Queues

You can use the **MessageQueue** component to create new queues on the Message Queuing network, delete queues, and perform some administrative actions on existing queues.

In This Section

Walkthrough: Creating a Queue and Working with Messages
Presents step-by-step instructions on how to interact with a message queue and send and receive messages.
Finding Queues in Server Explorer
Gives directions for determining what queues exist on the Message Queuing network.
Creating Queues
Gives directions for creating your own message queues within an existing network.
Queue Reference Recommendations
Suggests criteria for choosing identifying elements for queues, or linking your **MessageQueue** components to existing queues.
Setting Queue References
Describes how to refer to a queue in your project — by the path to the queue, by the queue's automatically generated format name, or by the non-unique descriptive label of the queue.
Deleting Queues
Gives directions for deleting queues from the network.
Purging Queue Contents
Gives directions for clearing the contents of a queue.
Creating MessageQueue Component Instances
Explains how to create **MessageQueue** component instances to build messaging functionality into your application.
Message Queue Configuration Properties
Describes properties you can manipulate on the queues and **MessageQueue** components you create.

Related Sections

Introduction to Messaging
Provides an introduction to messaging in Visual Studio and the .NET Framework.
Creating Messaging Components
Lists the major topics on using messaging in your applications.
Message Queues and Messaging Technology Backgrounder
Provides background explanation on message queuing technology.

Walkthrough: Creating a Queue and Working with Messages

The procedures in this topic demonstrate the process of creating a message queue using the **MessageQueue** component. Using that component, you send messages to the queue and retrieve messages from it. The messages log help requests.

> **Note** To see queue information in Server Explorer or to access queues programmatically, you must install the **Message Queuing Services** Windows component on your client computer. To add this service, use the **Add/Remove Programs** tool in the **Control Panel**.

> **Note** The Servers node of Server Explorer is not available in the Standard Edition of Visual Basic and Visual C# .NET. For more information, see Visual Basic Standard Edition Features or Visual C# Standard Edition Features.

To create a message queue on your computer

1. On the **File** menu, point to **New**, and then click **Project**.
2. In the **New Project** dialog box, choose Visual Basic or Visual C# in the left pane, then choose the **Windows Application** template. Name it **MessageQ**.
3. Open Server Explorer. For more information, see Accessing and Initializing Server Explorer .
4. Expand the **Servers** node.

   > **Note** The Servers node of Server Explorer is not available in the Standard Edition of Visual Basic and Visual C# .NET. For more information, see Visual Basic Standard Edition Features or Visual C# Standard Edition Features.

5. Expand the node for your local server. The node for your local server is identified by the computer name.
6. Expand the **Message Queues** node.
7. Right-click **Private Queues** and select **Create Queue** from the shortcut menu.
8. Enter **HelpRequest** for the queue name. Do not check **Make Transactional**.
9. A new private queue named **HelpRequest** is created and appears in Server Explorer.

   > **Note** You can also view the newly created queue through the **My Computer** icon. Right-click the **My Computer** icon on your desktop and select **Manage** from the shortcut menu. Expand the **Services and Applications** node. Expand the **Message Queuing** node and select the **Private Queues** folder. The new queue appears in the queue list.

To add a MessageQueue component for your message queue

1. Drag the **HelpRequest** queue from Server Explorer onto your form. A new **MessageQueue** component is added to the project, configured for the **HelpRequest** queue.

   The **MessageQueue** component is used to programmatically access the messages contained in the **HelpRequest** queue you created in the previous section.

2. Set the **Name** property of the **MessageQueue** component to **helpRequestQueue**.
3. In the Properties window, expand the **MessageReadPropertyFilter** node. Set the value of **Priority** to **true**. This causes the priority of the message to be retrieved when a message is retrieved from the queue.

The user interface that you create in the next procedure allows the user to enter some text for a help request and set the priority of the message. The user clicks a Send button to send the request to the queue. A datagrid displays the contents of the queue. The user interface also contains buttons to update the grid with the current contents of the queue and purge the queue.

To create the user interface

1. From the **Windows Forms** tab of the Toolbox, add the following controls to Form1:

   - Two labels
   - Two text boxes
   - Three buttons
   - One check box
   - One datagrid

2. Set the following properties of the controls:

Control	Property	New Value

Label1	Text	Name
Label2	Text	Message
TextBox1	Name	txtName
	Text	(blank)
TextBox2	Name	txtMessage
	Text	(blank)
	Multiline	true
Button1	Name	sendMessage
	Text	Send message
Button2	Name	refreshMessages
	Text	Refresh message list
Button3	Name	purgeMessages
	Text	Purge message list
CheckBox1	Name	highPriority
	Text	High priority
DataGrid1	Name	messageGrid

3. Arrange the controls in an orderly way.

To send a message to the queue

1. In the designer, double-click the **sendMessage** button to create the **Click** event handler in the Code Editor.
2. Add code to the method to create a new **Message** instance and send it to the queue, and to update the message display. You will write the `DisplayMessages` method in the next step.

```
' Visual Basic
Private Sub sendMessage_Click(ByVal sender As System.Object, _
   ByVal e As System.EventArgs) Handles sendMessage.Click
   Dim theMessage As System.Messaging.Message = _
      New System.Messaging.Message(Me.txtMessage.Text)
   theMessage.Label = Me.txtName.Text
   If highPriority.Checked Then
      theMessage.Priority = Messaging.MessagePriority.Highest
   Else
      theMessage.Priority = Messaging.MessagePriority.Normal
   End If
   helpRequestQueue.Send(theMessage)
   DisplayMessages()
End Sub

// C#
private void sendMessage_Click(object sender, System.EventArgs e)
{
   System.Messaging.Message theMessage =
      new System.Messaging.Message(txtMessage.Text);
   theMessage.Label = txtName.Text;
   if (highPriority.Checked)
      theMessage.Priority = System.Messaging.MessagePriority.Highest;
   else
      theMessage.Priority = System.Messaging.MessagePriority.Normal;
   helpRequestQueue.Send(theMessage);
   DisplayMessages();
}
```

3. Add a method to display the contents of the queue in the **DataGrid** control. This method uses the **MessageQueue.GetAllMessages** method to retrieve all the messages from the queue. The selected queue properties are added to a **DataTable** object, which is used as the data source for the **DataGrid** control. To retrieve the text of the message, you need to create a formatter for the message.

```
' Visual Basic
Private Sub DisplayMessages()
   ' Create a DataTable
   Dim messageTable As New DataTable()
   messageTable.Columns.Add("Name")
   messageTable.Columns.Add("Message")
   messageTable.Columns.Add("Priority")
   Dim messages() As System.Messaging.Message
   messages = helpRequestQueue.GetAllMessages()
   ' Need a formatter to get the text of the message body.
   Dim stringFormatter As System.Messaging.XmlMessageFormatter = _
      New System.Messaging.XmlMessageFormatter(New String() _
      {"System.String"})
   Dim index As Integer
   Dim am As System.Messaging.Message
   ' Add each message to the DataTable
   For index = 0 To messages.Length - 1
      messages(index).Formatter = stringFormatter
      am = messages(index)
      messageTable.Rows.Add(New String() _
         {am.Label, am.Body.ToString(), am.Priority.ToString()})
   Next

   messageGrid.DataSource = messageTable
End Sub

// C#
private void DisplayMessages()
{
   DataTable messageTable = new DataTable();
   messageTable.Columns.Add("Name");
   messageTable.Columns.Add("Message");
   messageTable.Columns.Add("Priority");
   System.Messaging.Message[] messages;
   messages = helpRequestQueue.GetAllMessages();
   System.Messaging.XmlMessageFormatter stringFormatter;
   stringFormatter = new System.Messaging.XmlMessageFormatter(
      new string[] {"System.String"});
   for (int index = 0; index < messages.Length; index++)
   {
      messages[index].Formatter = stringFormatter;
      messageTable.Rows.Add(new string[] {
         messages[index].Label,
         messages[index].Body.ToString(),
         messages[index].Priority.ToString() });
   }
   messageGrid.DataSource = messageTable;
}
```

To display the contents of the queue

1. In the designer, double-click the **refreshMessage** button to create the **Click** event handler in the Code Editor.
2. Within the **Click** event handler, call the **DisplayMessages** method.

```
' Visual Basic
Private Sub refreshMessages_Click(ByVal sender As System.Object, _
   ByVal e As System.EventArgs) Handles refreshMessages.Click
   DisplayMessages()
```

```
End Sub

// C#
private void refreshMessages_Click(object sender, System.EventArgs e)
{
   DisplayMessages();
}
```

To purge the contents of the queue

1. In the designer, double-click the **purgeMessage** button to create the **Click** event handler in the Code Editor.
2. Call the **Purge** method of the **helpRequestQueue**, and then refresh the contents of the **DataGrid** control.

```
' Visual Basic
Private Sub purgeMessages_Click(ByVal sender As System.Object, _
ByVal e As System.EventArgs) Handles purgeMessages.Click
   helpRequestQueue.Purge()
   DisplayMessages()
End Sub

// C#
private void purgeMessages_Click(object sender, System.EventArgs e)
{
   helpRequestQueue.Purge();
   DisplayMessages();
}
```

To test the application

1. Press F5 to run the application.
2. Enter your name and a short message.
3. Click **Send message** to send the message to the queue and update the display.
4. Click **Purge messages** to delete all the messages in the queue. The list of messages will be empty.

See Also

Introduction to Messaging | Creating Queues | Creating MessageQueue Component Instances | Sending Simple Messages | Retrieving Messages | Creating Messaging Components

Finding Queues in Server Explorer

Using Server Explorer, you can explore existing queues on any computer to which you have access. You can view any public, private, or system queue available on these computers, including those queues you create in Visual Studio .NET.

> **Note** The Servers node of the Server Explorer is not available in the Standard Edition of Visual Basic and Visual C# .NET. For more information, see Visual Basic Standard Edition Features or Visual C# Standard Edition Features.

You can also see information about queues in the Windows 2000 Computer Management window, or in the Create Message Queue dialog box that appears when you set the path for an instance of the **MessageQueue** component.

To locate a queue in Server Explorer

1. Expand the **Server** node, and then expand the node for the server you want to view.

   > **Tip** If the server you want to view is not listed, you need to add it. For more information, see Accessing and Initializing Server Explorer.

2. Locate and expand the **Message Queues** node beneath it.

   > **Note** If the Message Queues node will not expand, an error icon appears next to it. This means that the computer on which you are trying to view queues does not have Message Queuing installed. For information on how to install and configure Message Queuing, see your Message Queuing documentation.

3. Expand **Public Queues**, **Private Queues**, or **System Queues** depending on the type of queue you are trying to locate.

   > **Note** Expanding a public message queue can take a while if the domain is large, or if the domain controller uses an older version of Message Queuing. Use Message Queuing version 2.0 or above for the best performance.

4. Expand the node for the queue you want to work with.

To find a queue in the Message Queuing node of Computer Management

1. On the desktop for your computer, right-click **My Computer**, and then click **Manage**.
2. In the dialog box that appears, expand the **Services and Applications** node.
3. Expand the **Message Queuing** node. A full list of the queues on the computer appears.

See Also

Creating, Deleting, and Administering Queues | Create Message Queue Dialog Box

Creating Queues

You create a queue to provide a resource with which your messaging components can interact. For example, suppose you are building an order entry system that places orders into queues as they are received from the sales force or from direct customer interaction on a Web site. You might begin by creating an OrderEntry queue on your Message Queuing enterprise. All of the components you create to process orders will interact with this queue.

There are two ways you can create a queue: by using Server Explorer window or by using the Create constructor from your code. You can use Server Explorer or the Windows 2000 Computer Management window to verify that your queue has been created successfully.

> **Note** The Servers node of Server Explorer is not available in the Standard Edition of Visual Basic and Visual C# .NET. For more information, see Visual Basic Standard Edition Features or Visual C# Standard Edition Features.

You can create a public queue on your own computer or any Message Queuing computer to which you have domain or enterprise administrative access rights. You can also create private queues on your local computer only. For more information on access rights, see Message Queue Security, or see "Access Control for Message Queuing" in the documentation for the Computer Management dialog box.

> **Note** It is important to note the difference between creating a queue and creating an instance of the **MessageQueue** component. When you use the Create constructor, you are creating a new queue in the Message Queuing network, rather than a component in your project or application. When you create an instance of the **MessageQueue** component, you create a component inside your project that references an existing queue in the operating system.

You can also configure an installation component to create or locate an existing queue when your application is deployed. For more information, see Custom Actions Management in Deployment.

To create a queue from Server Explorer

When you add an instance of the **MessageQueue** component to your Visual Studio project, the system automatically creates the references and import statements you need to access these namespaces and classes.

1. Open Server Explorer and access the server on which you want to create a queue. For more information, see Accessing and Initializing Server Explorer.
2. Right-click the **Message Queues** node.
3. Right-click either **Public Queues** or **Private Queues**, depending on the type of queue you want to create, and then click **Create Queue**.
4. In the **Create Message Queue** dialog box, type a name for your queue.
5. If you want to make the queue transactional, select **Make Queue Transactional**.

   > **Note** For more information, see Transactional Message Processing.

   > **Note** Creating a public message queue can take longer than expected if the domain is large and the domain has many domain controllers. This is because it takes time for the new public queue to replicate from your machine's domain controller to all other domain controllers in the domain. You may experience a delay during this replication process.

To create a public queue programmatically

1. Right-click on the project and select **Add Reference**. Add a reference to **System.Messaging.dll**.
2. Use the **Create** method to create a public queue on your computer. Your code should look like this:

```
' Visual Basic
System.Messaging.MessageQueue.Create("myMachine\MyQueue")

// C#
System.Messaging.MessageQueue.Create(@"myMachine\MyQueue");
```

> **Tip** You can use an overload of the **Create** method to indicate that you want to create a transactional queue. You can also use a period (.) in the path to indicate the local machine.

Security Note When you create a message queue, be aware that the resource may already exist. Another process, perhaps a malicious one, may have already created the resource and have access to it. When you put data in the message queue, the data is available to the other process. For information on existing queues, see Retrieving Queues.

To create a private queue programmatically

- Use the **Create** method, specifying the necessary syntax, **Private$**, in order to create a private queue on your computer. Your code should look like this:

```
' Visual Basic
System.Messaging.MessageQueue.Create(".\Private$\MyPrivateQueue")

// C#
System.Messaging.MessageQueue.Create(@".\Private$\MyPrivateQueue");
```

To verify that your message queue was created

You can use Server Explorer or the Windows 2000 Computer Management window to verify that your message queue was created.

- In Server Explorer, refresh the **Message Queues** node by right-clicking **Queues** and then clicking **Refresh**. It may take a moment for your queue to appear.

  – or –

- To see message-queuing information in Windows 2000, follow these steps:

  1. Right-click **My Computer** on the desktop, and then click **Manage**.
  2. In the dialog box that appears, expand the **Services and Applications** node, and then expand **Message Queuing**. You can find your queue and its messages in this node.

See Also

Creating, Deleting, and Administering Queues | Creating MessageQueue Component Instances

Queue Reference Recommendations

If you create an instance of the **MessageQueue** component, you need to indicate what queue you want the component to communicate with. There are three ways to refer to a queue in your code:

- By path — the path to a queue uniquely identifies the computer and queue name for the queue in which you are interested.
- By format name — a unique identifier for the queue that is generated by MSMQ when the queue is created or generated later by the application.
- By label — a descriptive and potentially non-unique name for the queue that is assigned by the queue administrator when the queue is created.

When you create a new queue, rather than a new instance of the **MessageQueue** component, you must use the path. Format name and label are assigned by the Message Queuing system after the queue is created. When you refer to an existing queue in code or from the Toolbox, however, you can choose which method of referencing you want to use. Whatever method you choose is stored in the component's **Path** property.

Using Path to Refer to a Queue

A queue path takes the form **servername\queuename**. Paths to a queue are always unique. The following table lists the path information used for each type of queue:

Queue Type	Syntax Used in Path
Public queue	MachineName\QueueName
Private queue	MachineName\Private$\QueueName
Journal queue	MachineName\QueueName\Journal$
Machine journal queue	MachineName\Journal$
Machine dead-letter queue	MachineName\Deadletter$
Machine transactional dead-letter queue	MachineName\XactDeadletter$

You can use `"."` for the local machine name, so that `"YourMachine\MyQueue"` is equivalent to `".\MyQueue"`.

The following are a few caveats about using a path to refer to your queues:

- When the domain controller on a Message Queuing server receives an operation for a queue that is referenced by path, it must resolve that path and determine the format name of the queue in question. This may result in slightly slower performance than if you refer to the queue by format name directly.
- If you are designing your component to be used as an XML Web service, you will get the best performance results by using the format name method of referring to queues rather than the path method.
- You cannot refer to a queue by path if you want to send messages to it when the queue is disconnected. Disconnected messages must be sent using the format name.

You can retrieve the path for a queue by querying and combining the return results of two separate properties in the MessageQueue base class — the MachineName property and the QueueName property.

In code, referencing a queue by path looks like this:

```
' Visual Basic
MessageQueue1.Path = "YourMachine\MyQueue"

// C#
MessageQueue1.Path= @"YourMachine\MyQueue";
```

> **Note** The @".\MyQueue" syntax for C# is equivalent to using ".\\MyQueue" instead. The @ sign indicates a literal string. For more information, see string.

Using Format Name to Refer to a Queue

Format names take the form of a string indicating whether a queue is public or private, followed by a generated GUID for the queue and other identifiers as necessary. The following table lists the path information used for each type of queue:

Queue Type	Syntax Used in Format Name
Public queue	FORMATNAME:PUBLIC=QueueGUID

Private queue	FORMATNAME:PRIVATE=MachineGUID\QueueNumber
Journal queue	FORMATNAME:PUBLIC=QueueGUID;JOURNAL – or – FORMATNAME:PRIVATE=MachineGUID\QueueNumber;JOURNAL

You do not assign a format name to a queue; instead, the queue manager generates this value when the queue is created. Referencing a queue by its format name is the most direct way to access a queue, because the domain controller on the server does not have to interpret the reference as it does when you refer to the queue by path.

The following are a few hints about referencing your queues by format name:

- If you intend to send messages to a disconnected queue, you must refer to the queue by format name rather than path, because the path will not be available to resolve with if the queue is offline.
- Be aware that format names may become invalid when your network topography changes or when a queue is deleted and re-created.

You can retrieve a format name for a queue by querying the FormatName property in the **MessageQueue** class.

In code, referencing a queue by format name looks like this:

```
' Visual Basic
MessageQueue1.Path = _
   "FORMATNAME:PUBLIC=3d3dc813-c555-4fd3-8ce0-79d5b45e0d75"

// C#
MessageQueue1.Path =
   "FORMATNAME:PUBLIC=3d3dc813-c555-4fd3-8ce0-79d5b45e0d75";
```

Using Labels to Refer to Queues

You can also refer to a queue by its label, which is a descriptive text label given to the queue by its administrator. Labels are not always unique, so you will receive an error if a name conflict exists when you try to connect to a specific queue using its label.

Labels can be useful in a situation where you know you are going to be moving a queue from one computer to another. If you refer to the queue by label only, all of your operations will continue to work successfully after the queue has been moved to its new location, provided that there are no other queues with that label on the new computer. If there are, the **Send** method will produce an error.

You can retrieve the label for a queue by querying the Label property in the **MessageQueue** class.

In code, referencing a queue by label name looks like this:

```
' Visual Basic
MessageQueue1.Path = "LABEL:MyQueue"

// C#
MessageQueue1.Path= "LABEL:MyQueue";
```

See Also

Creating, Deleting, and Administering Queues | Setting Queue References | Creating Queues | Creating MessageQueue Component Instances

Setting Queue References

You have three choices in the way you refer to a queue in your project — you can refer to it by the path to the queue, by the queue's automatically generated format name, or by the non-unique descriptive label of the queue. Each of these ways of referencing a queue has its advantages and disadvantages. For more information, see Queue Reference Recommendations.

> **Note** When you set a queue reference, you are merely linking the **MessageQueue** component you created to the existing queue. You are not changing the way the actual queue itself is identified on the server.

To set a queue reference

1. Create an instance of the **MessageQueue** component. For more information, see Creating MessageQueue Component Instances.
2. In the Properties window, select the **Ellipsis** button ( ) next to the **Path** property to display the **Message Queue Browser** dialog box.
3. In the dialog box, browse the **Select Message Queue** list and select the queue you want to reference.
4. In the **Reference Queue By** drop-down list, select one of the following:
   - To refer to the queue by its path, select **Path**.
   - To refer to the queue by its format name, select **FormatName**.
   - To refer to the queue by its label, select **Label**.
5. Click **OK** to return to the Properties window. The **Path** property now shows the path, format name, or label for the queue you selected.

See Also

Creating, Deleting, and Administering Queues | Queue Reference Recommendations

Deleting Queues

You can delete queues that exist on your local computer in two ways: from code or from Server Explorer. You can delete any public or private queues on your local computer, provided that you have the appropriate access rights. To delete a local queue, you must have either full control access rights assigned to you by the queue's owner, or explicit Message Queuing access rights to delete queue-related items from that machine. You cannot delete queues on other computers.

> **Note** The Servers node of Server Explorer is not available in the Standard Edition of Visual Basic and Visual C# .NET. For more information, see Visual Basic Standard Edition Features or Visual C# Standard Edition Features.

> **Note** You do not need to purge a queue's messages before deleting the queue. When you delete a queue, all of its messages are deleted with it. These messages are not sent to a dead-letter queue and cannot be retrieved. For more information on purging, see Purging Queue Contents.

To delete a message queue from Server Explorer

1. Open Server Explorer, and then expand the node for your server.
2. Expand the **Message Queues** node and locate the queue you want to delete.
3. Right-click the queue, and then click **Delete.**

> **Note** Deleting a public message queue can take longer than expected if the domain is large and the domain has many domain controllers. This is because it takes time for the deletion command to replicate from your machine's domain controller to all other domain controllers in the domain. You may experience a delay during this replication process.

To delete a message queue programmatically

- Use the Delete method to delete the queue you want. Your code should look something like this:

```
' Visual Basic
System.Messaging.MessageQueue.Delete("myMachine\MyQueue")

// C#
System.Messaging.MessageQueue.Delete (@"myMachine\MyQueue");
```

> **Note** This example shows the queue referenced by path. You can also refer to the queue you want to delete by format name or by label. For more information, see Queue Reference Recommendations.

See Also

Creating, Deleting, and Administering Queues | Queue Reference Recommendations | Purging Queue Contents

Purging Queue Contents

You can use the **Purge** method to clear the contents of any queue to which you have access rights in the Message Queuing system. For example, suppose you use a journal queue on your local Message Queuing client to record copies of all messages you send out. When that journal reaches its maximum size, you might use the Purge method to clear out the items you no longer need.

> **Note** Messages stored in journal and dead-letter queues count against the quota for the computer where the queue resides. When the quota is reached, these queues stop accepting new messages. It is important to periodically clear messages that are no longer needed from journal and dead-letter queues if you are using these in your application.

You must have Receive rights in Message Queuing to delete the contents of a queue. When you purge a queue, all of the messages it contains are discarded without being sent. You can purge a queue's contents from code or from Server Explorer.

To purge a queue's contents from Server Explorer

1. Open Server Explorer, and then expand the node for the server on which the queue you want to purge resides.

   > **Tip** If you do not see the server you want listed in the **Servers** node, right-click **Servers**, click **Add Server**, enter the name of your server, and then click **OK**.

   > **Note** The Servers node of Server Explorer is not available in the Standard Edition of Visual Basic and Visual C# .NET. For more information, see Visual Basic Standard Edition Features or Visual C# Standard Edition Features.

2. Expand the **Message Queues** node.
3. Expand the queue from which you want to delete messages. You will see one or more nodes for the queue's messages and its associated journal messages.
4. Right-click the message node you want to clear, and then click **Clear Messages**.

To purge a queue's contents programmatically

- Use the **Purge** method to clear the contents of the appropriate queue. Your code should look something like this:

```
' Visual Basic
Dim MessageQueue1 As New System.Messaging.MessageQueue()
MessageQueue1.Path = ".\MyQueue"
MessageQueue1.Purge()

// C#
System.Messaging.MessageQueue MessageQueue1 =
   new System.Messaging.MessageQueue();
MessageQueue1.Path = @".\MyQueue";
MessageQueue1.Purge();
```

See Also

Creating, Deleting, and Administering Queues | Creating Queues

Creating MessageQueue Component Instances

You create **MessageQueue** components when you want to build messaging functionality into your application. **MessageQueue** components allow you to connect to existing queues, send and receive messages, and otherwise add communication services to your application using a very small amount of code. For example, suppose you are building an order entry system that places orders into queues as they are received from the sales force or from direct customer interaction on a Web site. You might begin by adding an instance of the **MessageQueue** component to your project and configuring it to interact with an existing OrderEntry queue on your Message Queuing server.

You can add an instance of the **MessageQueue** component to Windows Forms, Web Forms, and component classes. **MessageQueue** components have no visual user interface. If you add an instance of the **MessageQueue** component to a visual designer (such as the Windows Forms Designer), the component appears in a small area below the bottom border of the form. This is known as the component tray and acts as a place to display all of the nonvisual items associated with the form.

> **Note** Nonvisual classes that inherit from **System.ComponentModel.Component** support a visual design surface similar to the component tray, on which your **MessageQueue** component and other components are arranged. The arrangement of items on this designer is not important, as the interface you see will never be displayed to the application's end users.

For instructions on how to configure **MessageQueue** components, see Message Queue Configuration Properties.

There are several ways you can create an instance of the **MessageQueue** component:

- You can drag an instance of the **MessageQueue** component from the Components Toolbox tab to a form or Component Designer.
- You can locate the queue you want in Server Explorer and add it to your designer, creating an instance of the **MessageQueue** component that is preconfigured to point to that queue.
- You can create an instance of the **MessageQueue** component in code.

To create an instance of the MessageQueue component from the Toolbox

1. Access the **Components** tab of the Toolbox.
2. Select the **MessageQueue** icon and drag it to the designer surface for your form or component.
3. Configure your component. For more information, see Message Queue Configuration Properties.

To create an instance of the MessageQueue component from Server Explorer

1. Open Server Explorer. For more information, see Accessing and Initializing Server Explorer.
2. Expand the **Servers** node and determine whether this node lists the server on which the queue you want resides.

   > **Tip** If the server you want to view is not listed, you need to add it. For more information, see Accessing and Initializing Server Explorer.

   > **Note** The Servers node of Server Explorer is not available in the Standard Edition of Visual Basic and Visual C# .NET. For more information, see Visual Basic Standard Edition Features or Visual C# Standard Edition Features.

3. Expand the node for the server you want to view, and then locate and expand the **Message Queues** node beneath it.

   > **Note** If the Message Queues node does not expand, the computer on which you are trying to view queues does not have Message Queuing installed. For information on how to install and configure Message Queuing, see your Windows 2000 or Windows NT Message Queuing documentation.

4. Locate the queue you want to add to your project, and then right-click it.
5. Click **Add to Designer**.
6. Configure your component. For more information, see Message Queue Configuration Properties.

To create an instance of the MessageQueue component programmatically

1. Create an instance of the **MessageQueue** class in code and set the Path property to determine which existing queue you want your component to reference. Your code might look like this:

```
' Visual Basic
Dim myMQ As New System.Messaging.MessageQueue()
```

```
myMQ.Path = ".\MyNewQueue"

// C#
System.Messaging.MessageQueue myMQ =
   new System.Messaging.MessageQueue();
myMQ.Path = @".\MyNewQueue";
```

Tip You can also do steps one and two in a single step, by using this format:

```
' Visual Basic
Dim myMQ2 As New System.Messaging.MessageQueue(".\MyNewQueue")

// C#
System.Messaging.MessageQueue myMQ2 =
   new System.Messaging.MessageQueue(@".\MyNewQueue");
```

Note You can refer to the queue by the path to the queue, the queue's automatically generated format name, or by the non-unique descriptive label of the queue. Each of these ways of referencing a queue has its advantages and disadvantages. For more information, see Queue Reference Recommendations.

2. Configure any other necessary properties for your queue. For more information, see Message Queue Configuration Properties.

See Also

Message Queue Configuration Properties | Accessing and Initializing Server Explorer

Message Queue Configuration Properties

There are a variety of properties you can manipulate on the queues and **MessageQueue** components you create. You can define these properties to help you set values that guide the queue's behavior, or query these properties to retrieve information about a queue with which your component interacts.

Properties That Reference a Queue

You use the **Path** property to determine what queue your component references. The path can be identified in one of three ways — by the physical path to the queue, by the queue's format name, or by the queue's descriptive label. For a full explanation of the differences between path, format name, and label, see Queue Reference Recommendations.

Queues can also be classified according to a category. You can use the **Category** property to determine the type of queue you are working with. **Category** is a GUID that is defined by the owner of the queue. You can either use a GUID-generation tool or make up your own number for the GUID. **Category** GUIDs do not have to be unique except within a single category. For example, you might define the same category for all of your order processing queues. This would allow you to retrieve all of them by that GUID.

Properties That Determine What Type of Data You Can Send

You use the **Formatter** property to determine how your message content will be serialized into a message before being sent to a queue, and how it will be extracted from a message after being received from a queue. Setting the **Formatter** property determines what type of content you can send in a message. For more information, see Message Serialization.

Properties That Control Interaction with a Queue

You can use the following properties to control how your component interacts with a queue:

- You can use the **DenySharedReceive** property to indicate that only one component at a time may look at the messages in a queue. This can be extremely useful in multi-user situations. When **DenySharedReceive** is set, the first component to peek at or receive messages off a queue is the only component that may perform these operations at that time. When the first component finishes its instance of that queue, it releases the queue either through garbage collection or by calling **Close**. For more information about garbage collection, see Reference Counting, Garbage Collection, and Object Lifetime.
- You can use the **CanRead** and **CanWrite** properties to determine if you have read and write access to a particular queue, or to set a queue to allow read and write access.
- You can use the **MaximumQueueSize** and **MaximumJournalSize** properties to determine the capacity, in kilobytes, of messages that a queue can receive. Once this maximum has been reached, messages are no longer accepted by the queue. By default, the maximum queue size defaults to the queue size set by the system administrator for Message Queuing. If no limit has been set, default queue size is infinite.
- You can use the **UseJournalQueue** property to determine whether copies of the messages retrieved from the queue are added to a journal queue.

For a full list of the properties associated with the **MessageQueue** class, see MessageQueue Members.

See Also

Creating MessageQueue Component Instances | Reference Counting, Garbage Collection, and Object Lifetime | MessageQueue Members

Sending and Serializing Messages

A message in Microsoft Windows Message Queuing consists of a body and a set of properties. Message bodies can contain text or any form of binary information, and can be encrypted. You set a message's body and properties programmatically, either by setting property values in the Properties window or by setting the required values in code. A message, including its body and all specified properties, cannot be bigger than 4 megabytes (MB) in size.

You can create and send messages with your **MessageQueue** components. Messages created in your applications can be sent to any public, private, journal, dead-letter, or transactional queue.

In This Section

Sending Simple Messages
Gives directions for sending simple messages as parameters of the **Send** method.
Sending Complex Messages
Gives directions for sending messages using the **Message** object to fine-tune the message content and processing.
Sending Messages to Disconnected Queues
Gives directions for sending a message to a queue that is not currently online.
Message Acknowledgement
Provides an overview of the types of acknowledgement messages you can have your **MessageQueue** component generate.
Requesting Acknowledgement for Messages
Gives directions for requesting acknowledgement messages to indicate whether your message successfully reached its destination.
Message Queue Journal Storage
Describes journal queues, which allow you to save copies of messages as they are processed.
Storing Messages in Journal Queues
Gives directions for storing copies of the messages you send in a journal queue.
Default Message Properties
Defines the available default properties that can be applied to all messages you send.
Message Serialization
Explains serializing objects data into and deserializing them out of a message.
Verifying Messages at Design Time
Describes how to use the Properties window to verify information about a message.

Related Sections

Introduction to Messaging
Provides an introduction to messaging in Visual Studio and the .NET Framework.
Creating Messaging Components
Lists the major topics on using messaging in your applications.
Transactional Message Processing
Describes the process of sending a series of messages together in a transaction, in which all of the messages are either successfully delivered or not sent.
Message Queues and Messaging Technology Backgrounder
Provides background explanation on message queuing technology.

Sending Simple Messages

Using an instance of the **MessageQueue** component, you can send simple messages to a queue in as little as two lines of code. When you send a simple message, you must perform the following actions:

- Create a connection to the queue to which you want to send a message.
- Specify the appropriate formatter for the data you want to send. The formatter controls what type of data can be sent in a message and how that data is persisted. In simple cases, the default formatter is acceptable. For more information, see Message Serialization.
- Call the MessageQueue.Send method, passing the object to be sent.

Creating a Connection to the Queue

After you decide which queue you want to communicate with, you need to create an instance of the **MessageQueue** component that references the queue you want to use. You can create this component either from the Toolbox, from Server Explorer to your project, or by using the MessageQueue constructor.

To create a connection to the queue you want to communicate with

1. Create an instance of the **MessageQueue** component. For more information, see Creating MessageQueue Component Instances.
2. Use the component's Path property to connect to the queue with which you want to communicate by the queue path, format name, or label.

   Note If you created your component from Server Explorer, the **Path** property is set automatically to the queue path for that queue.

Providing the Data to Be Sent

Each **MessageQueue** component contains a series of default properties that are applied to all messages sent by that component, unless you specify otherwise in an individual message. In the simplest scenario, you can send a simple message to the queue using the default properties set for the component. For more information on these properties, see Default Message Properties.

You use the **Send** method to specify a message and send it to the queue. You can send objects, primitive data types, streams, and other kinds of data in a simple message.

The **MessageQueue** component takes the data you specify in the **Send** method's argument, persists it into a message, and sends the message to the specified queue.

Note You can also use the **Message** object to send far more complex messages to a queue. In addition, you can send messages as part of a transaction. For more information, see Sending Complex Messages.

To send a simple message

1. In the Properties window for your **MessageQueue** component, accept the default value for the Formatter property, or set the property to **XMLMessageFormatter** if you have changed the default value.
2. Use the **Send** method to send a simple message to your queue, specifying the message as an argument of the method.

   Your code to send a simple message might look like this:

```
' Visual Basic
' Create a connection to the queue.
Dim MessageQueue1 As New System.Messaging.MessageQueue (".\YourQueue")
' Send an integer.
MessageQueue1.Send(1)
' Send a string.
MessageQueue1.Send("Hello world")

// C#
// Create a connection to the queue.
System.Messaging.MessageQueue mq =
   new System.Messaging.MessageQueue (@".\YourQueue");
```

```
// Send an integer.
mq.Send(1);
// Send a string.
mq.Send("Hello world");
```

See Also

Sending and Serializing Messages | Creating MessageQueue Component Instances | Sending Complex Messages | Transactional Message Processing | Verifying Messages at Design Time

Sending Complex Messages

In addition to sending a simple message by specifying your message in the Send method's argument, you can gain more control over your messages by explicitly creating a **Message** object, rather than letting the shared **Send** method create and send one for you. By manipulating the properties of the resulting **Message** object, you can create more complex messages and exert more control over the way your messages are handled.

When you send a complex message, you must first create a connection to the queue with which you want to communicate, and then specify the data to send. However, as part of specifying the data to send, you will create an instance of the Message class, set the properties you need, and fine-tune the delivery mechanisms before sending your message. As with simple messages, the system persists your message object and sends it to the queue you specified.

For more information on the properties you can use to fine-tune your **Message** object, see Message Members.

> **Note** Using the **Message** object, you can send messages as part of a transaction. For more information, see Transactional Message Processing.

To send a complex message using the Message class

1. Create an instance of the **MessageQueue** component and set its Path property to the queue to which you want to refer. For more information, see Creating MessageQueue Component Instances.

   > **Note** If you created your component from Server Explorer, the **Path** property is set automatically to the queue path for that queue.

2. Create an instance of the **Message** object.
3. Set the body of the message and specify any properties you want to change from their default values.
4. Use the **Send** method to send your object to the queue.

   When you are finished, your code might look like this:

```
' Visual Basic
Dim myMQ1 As New System.Messaging.MessageQueue(".\YourQueue")
Dim newMessage As New System.Messaging.Message("Hello again")
newMessage.Label = "This is the label"
myMQ1.Send(newMessage)

// C#
System.Messaging.MessageQueue myMQ1 =
   new System.Messaging.MessageQueue (@".\YourQueue");
System.Messaging.Message newMessage =
   new System.Messaging.Message("Hello again");
newMessage.Label = "This is the label";
myMQ1.Send(newMessage);
```

See Also

Sending and Serializing Messages | Creating MessageQueue Component Instances | Sending Simple Messages | Transactional Message Processing | Verifying Messages at Design Time

Sending Messages to Disconnected Queues

There are two situations in which messages cannot readily be delivered to their queues: when the machine on which the queue resides is not available, or when the domain controller needed to route your message is not available. Message Queuing allows you to handle these situations so that if you are disconnected from the network or a necessary machine or controller is not available, you can continue to send messages. In these cases, the messages are temporarily stored on a queue on the local machine or a machine along the delivery route until the necessary resources are back online for delivery to be completed.

For example, suppose you have a central queue that records orders sent by your on-the-road sales force. The sales force works in disconnected mode much of the day, recording order information from customer sites, and dials in once a day to transfer all of this information to the central queue. Because messages can be sent to queues when the sender is disconnected, the sales force can send their messages immediately upon recording the customer's information, and the system caches those messages until the nightly call is placed.

Sending a message to a disconnected queue is almost identical to the process of sending a message to an available queue. You do not have to perform any special configuration in order for your component to store messages in a temporary queue when the queue to which you are sending is not available. There are two main differences to be aware of:

- If the queue is unavailable but the domain controller is working, you must identify the queue you want to send to by the queue's format name, rather than the path name. When a queue is disconnected, the domain controller for the queue cannot resolve the path name correctly. For more information on format names, see Queue Reference Recommendations.
- If you set a time-out interval on your message send operation, the time-out period will begin to count down as soon as the message is sent. This means that while the message sits on your local machine waiting for the queue to become available, the time-out period is active and the expiration time is approaching. If you want to use the time-out period, you must either set it to a large enough interval to allow for delivery or request negative acknowledgement for the message so you know if expiration has occurred. For more information on requesting acknowledgement when a message cannot be delivered, see Requesting Acknowledgement for Messages.

To send a message to a disconnected queue

1. Create an instance of the **MessageQueue** component. For more information, see Creating MessageQueue Component Instances.
2. Set the Path property for the component to the format name of the queue to which you want to send a message.

   > **Note** If you create the component from Server Explorer, this property will automatically refer to the path instead of the format name. To change to the format name, click the expansion icon next to the **Path** property. In the dialog box that appears, locate and select the queue your component references; then select **Format Name** in the **Reference Queue By** field and click **OK**.

3. Send your message normally.

   - For instructions on sending simple messages, see Sending Simple Messages.

     – or –

   - For instructions on sending messages using the **Message** object, see Sending Complex Messages.

See Also

Sending and Serializing Messages | Requesting Acknowledgement for Messages | Default Message Properties | Sending Simple Messages | Sending Complex Messages | Creating MessageQueue Component Instances | Verifying Messages at Design Time

Message Acknowledgement

You can set up your **MessageQueue** component to generate acknowledgement messages that tell you whether the message was successfully delivered. There are two main types of acknowledgement you can receive:

- Acknowledgement that the message reached its destination queue.
- Acknowledgement that the message was retrieved from the queue by the destination application.

Within each of these scenarios, you can also receive positive or negative acknowledgement. In *positive acknowledgement*, you receive an acknowledgement message when your message successfully reaches its destination queue or application. In *negative acknowledgement*, you receive an acknowledgement message when your message fails to reach its destination queue or application. A message might fail to reach its destination if its time-out period expires or if the message cannot be authenticated.

As with most operations in Message Queuing, acknowledgements are handled via sending a new message to a queue. In this case, the acknowledgement message is sent from the target queue to a special type of queue called an Administration queue. Acknowledgement messages differ from standard messages in that they contain no attached body; only the information in the message header is of interest in an acknowledgement.

> **Note** Acknowledgement messages for your application are sent to whatever queue you specify in the AdministrationQueue property, either in the DefaultPropertiestoSend class for your component instance or on the **Message** object. You determine the type of acknowledgement you want to receive by setting the AcknowledgeType property to one of its predetermined values.

You can check acknowledgement messages in the same way you check any message on a queue — by peeking at the acknowledgement queue or removing messages from it. For more information, see Reading and Receiving Messages.

For instructions on setting up acknowledgement, see Requesting Acknowledgement for Messages.

See Also

Sending and Serializing Messages | Requesting Acknowledgement for Messages | Reading and Receiving Messages

Requesting Acknowledgement for Messages

You can request positive acknowledgement, negative acknowledgement, or a combination of acknowledgement types on the messages sent by your **MessageQueue** components. For an overview of acknowledgement types, see Message Acknowledgement.

To request acknowledgement of a message

1. Create an instance of the **MessageQueue** component and set its Path property to the queue you want to reference. For more information, see Creating MessageQueue Component Instances.
2. Create the message you want to send. For more information, see Sending and Serializing Messages.
3. Set the AdministrationQueue property for your **Message** object to the queue to which you want acknowledgement messages sent.
4. Set the **Message** object's AcknowledgeType property in the **DefaultPropertiestoSend** area to the appropriate value:

   - If you want acknowledgement when a message reaches or fails to reach the queue to which it was sent, set the property to **FullReachQueue**.
   - If you want acknowledgement when the message reaches or fails to be retrieved from the queue before its time-to-be-received timer expires, set the property to **FullReceive**. Note that this acknowledgement is generated by the queue manager, rather than by the destination application.
   - If you want to request negative acknowledgment when a message fails to be retrieved from the queue, set the property to **NegativeReceive**.
   - If you want acknowledgement only if a message you send fails to reach the queue, set the property to **NotAcknowledgeReachQueue**.
   - If you want acknowledgement only if a message you sent could not be retrieved by its destination application, set the property to **NotAcknowledgeReceive**.
   - If you want positive acknowledgment when the message reaches its queue, set the property to **PositiveArrival**.
   - If you want positive acknowledgment when the message is successfully retrieved from the queue, set the property to **PositiveReceive**.

     Tip For information on positive and negative acknowledgment, see Message Acknowledgement.

5. Send the message, and then check the queue you specified in the **AdministrationQueue** property for the appropriate acknowledgement. For more information, see Reading and Receiving Messages.

To prevent acknowledgement messages from being sent

- Set the **AcknowledgeType** property for your component to **None**.

See Also

Sending and Serializing Messages | Message Acknowledgement | Creating MessageQueue Component Instances | Reading and Receiving Messages | Sending and Serializing Messages

Message Queue Journal Storage

Journal queues allow you to save copies of messages as they are processed. You can store copies of the messages you send from your computer in the local machine journal queue, or you can store copies of messages removed from a queue in that queue's journal on the server.

Journal storage can be useful if you need to resend a message at a later point. For example, suppose you send a message to a queue and then receive negative acknowledgement that it could not be delivered. Acknowledgement messages do not contain the body of the original message; instead they give you some header information and a correlation ID that corresponds to the original message. If you have journal recording turned on, you can use this correlation ID to locate the original message in your machine's journal queue, re-create the message object, and resend it.

> **Note** You can use the ReceivebyCorrelationID or the PeekbyCorrelationID methods to retrieve a message by its correlation ID. Note that several messages may have the same correlation ID, and these methods will retrieve the first message in the queue with the given ID.

There are a few caveats you should note about journal queues:

- There is one global journal queue per machine. All messages sent from that machine will be recorded in that journal queue, whether they are delivered successfully or not. In addition, each queue has its own associated journal queue.
- There are two properties that control how journals are used. If you set the UseJournalQueue on the **MessageQueue** object, you enable journal storage for any messages received by that queue, not messages sent with that object. The messages sent with the queue will be placed in the queue journal. If you set the UseJournalQueue on the **Message** object, you enable system journal storage; this means that messages sent from your system will be placed in the sender's system queue.
- Journal queues have a maximum size, called a quota. This size is limited by the disk capacity of the computer on which the queue resides. After the queue capacity is reached, messages that would have been sent to the journal queue are no longer saved in the system. No errors are raised when this occurs. It is important that you periodically purge your journal queues to prevent this from occurring. For more information, see Purging Queue Contents.
- Journal queues are silent — that is, they cannot create acknowledgement messages of their own, send removed messages to dead-letter queues, or process time-outs.

For instructions on setting up journal storage for your messages, see Storing Messages in Journal Queues.

See Also

Sending and Serializing Messages | Storing Messages in Journal Queues | Purging Queue Contents

Storing Messages in Journal Queues

There are two ways you can set up the use of journal queues for sent messages — you can set a default property for your **MessageQueue** component so that all messages sent by it will use journals, or you can establish journal queue recording for individual messages you are sending.

For an overview of journal storage, see Message Queue Journal Storage.

To use journals for all messages sent by an instance of the MessageQueue component

1. Create an instance of the **MessageQueue** component and set its Path property to the queue to which you want to refer. For more information, see Creating MessageQueue Component Instances.
2. In the Properties window, set the UseJournalQueue property on the DefaultPropertiestoSend class for your component to **true**.

To set journal recording for an individual message

1. Create the message you want to send.
2. In code, set the **UseJournalQueue** property to **true**. Your code might look like this:

```
' Visual Basic
Message1.UseJournalQueue = True

// C#
Message1.UseJournalQueue = true;
```

To set journal recording for messages removed from a queue

1. Create an instance of the **MessageQueue** component and set its **Path** property to the queue to which you want to refer. For more information, see Creating MessageQueue Component Instances.
2. Set the **UseJournalQueue** property for the **MessageQueue** component that references that queue to **true**. Your code might look like this:

```
' Visual Basic
MessageQueue1.UseJournalQueue = True

// C#
MessageQueue1.UseJournalQueue = true;
```

To set the maximum size for a journal queue

1. Create an instance of the **MessageQueue** component and set its **Path** property to the journal queue you want to refer to. For more information, see Creating MessageQueue Component Instances.
2. Set the MaximumJournalSize property to the capacity, in kilobytes, that you want to use for your journal queue.

See Also

Sending and Serializing Messages | Message Queue Journal Storage | Purging Queue Contents | Default Message Properties | Requesting Acknowledgement for Messages | Creating MessageQueue Component Instances

Default Message Properties

When you create an instance of the **MessageQueue** component, you have the option of setting a series of default properties that messages sent by that component will use. This allows you to save time when defining your messages, and to have more control over the Send method without the need to create and configure **Message** objects.

> **Note** Only messages that are not sent using the **Message** object use the **DefaultPropertiestoSend** values. When you send a message using the **Message** object, the properties of the **Message** object are used.

You set default message properties by setting values in the **DefaultPropertiestoSend** section of the Properties window, or by setting values for the properties in the DefaultPropertiestoSend class. These default properties control features such as whether acknowledgement messages are generated and where they are sent, how your messages are encrypted and authenticated, and the time-out period for your messages.

This topic provides an overview of how you might set some of the more critical properties in this area. For a full description of the default properties, see DefaultPropertiesToSend Members.

Message Priority

Setting a default priority for your messages affects the order in which they are sent to their destination queues. You set the Priority property to determine the default priority for messages sent by your component. Priority ratings default to normal, with settings for lower and higher priorities.

The priority setting for a message determines where the message is placed in the queue. Messages sent with a higher priority are placed higher in the queue, and messages with a lower priority are placed lower in the queue. When the queue receives a set of messages with an identical priority setting, the messages are arranged in the queue according to the time when they were sent.

> **Note** You cannot set priority on messages that are being sent to transactional queues, because the order in which messages are processed on a transactional queues are determined by the transaction itself.

Acknowledgement and Response Properties

There are several default properties you can set that determine whether responses and acknowledgements will be generated for your component's messages and how these responses will be handled. The properties you can set as a default for all messaging operations include:

- The AcknowledgeType property and the AdministrationQueue properties, which you can use to determine whether acknowledgement messages should be generated when your messages reach or fail to reach their destination queue, or when the messages are retrieved from the queue by the target application. For more information, see Requesting Acknowledgement for Messages.
- The ResponseQueue property, which you can use to indicate the return path for any messages that will be sent by the application retrieving your messages from the queue. These messages differ from acknowledgement messages in that they are generated by the destination application, rather than by the queue manager.
- You can determine whether copies of your outgoing messages should be stored in a journal queue by setting the UseJournalQueue property and, similarly, whether undeliverable messages should be sent to a dead-letter queue by setting the UseDeadLetterQueue property.

Time-Out Properties

There are two time-related properties you can use to help maintain better control of your messages. Both of these determine how long a message can exist in the system before it is discarded. You use the TimeToReachQueue property to specify how long a message has to try to reach its destination queue. You use the TimeToBeReceived property to specify how long the message should remain in the system, from the time it is sent until the time it is removed from the queue by the destination application. If either timer interval expires, Message Queuing discards the message.

When a message is discarded due to an expired timer, the queue manager might take additional steps:

- If you set the **UseDeadLetterQueue** property to **true**, a copy of the discarded message will be sent to the specified dead-letter queue.
- If you set the **AcknowledgeType** property to send negative acknowledgements, an acknowledgement message will be sent to the specified Administration queue.

By default, no time-out interval is set for either property. If you set a value for both properties, the value in the

TimeToBeReceived property takes precedence.

See Also

Sending and Serializing Messages | Creating MessageQueue Component Instances

Message Serialization

Serialization is the process of taking objects and converting their state information into a form that can be stored or transported. The basic idea of serialization is that an object writes its current state, usually indicated by the value of its member variables, to persistent storage. Later, the object can be re-created by reading, or deserializing, the object's state from the storage. Serialization handles all the details of object pointers and circular object references that are used when you serialize an object.

In the message-queuing feature, serialization refers specifically to the process of converting an object or set of data into a message that can be sent to a queue, and then converting messages retrieved from the queue back into objects or data that your application can process.

A formatter object handles message serialization in your Visual Studio .NET or .NET framework applications. When a message is sent to the queue, the formatter serializes an object into a stream that can be sent to the message queue. When reading from a queue, the formatter deserializes the message data into the Body property.

You choose the appropriate type of formatter for the kind of data you want to send and receive. Visual Studio and the .NET Framework ship three predefined formatters, each designed to persist and de-persist a different type of data:

- The **XMLMessageFormatter** object persists objects and primitive data types into and out of messages using human-readable XML strings. This is the default formatter setting for **MessageQueue** components.
- The **BinaryMessageFormatter** object persists one or more connected objects into serialized streams. The result is very compact and fast to parse, but not human-readable.
- The **ActiveXMessageFormatter** object persists primitive data types, allowing for interoperability with components that use previous versions of Message Queuing. The resulting serialization is very compact. This formatter is designed with Windows in mind and does not produce human-readable results. However, it is an extremely fast method of serialization.

By default, an XMLMessageFormatter is created for you when you create a **MessageQueue** component instance and is associated with the instance. When the Send method is called on the **MessageQueue** instance, the body of your message is serialized using this formatter. You do not have to write any additional code to use the formatter during a send operation.

Reading a message is slightly more complex. To read from the queue using a formatter, you must set properties indicating how the body of the message should be handled. If you are using the **XMLMessageFormatter** object, you set the TargetTypes or TargetTypeNames property. If you are using the **BinaryMessageFormatter** object, you set a different series of properties. The **ActiveXMessageFormatter** object has no associated properties. For more information, see Reading and Receiving Messages.

In addition to using the formatters shipped with Visual Studio and the .NET framework, you can create your own formatters if you need to work with different types of data. You can import your formatter into a project and access it in code.

You choose the formatter you want to use by setting the **Formatter** property for the queue, message, or **MessageQueue** component with which you are working.

Advantages of XML Message Formatting

One advantage of the XML formatter is that you can read the strings that are created when the object or data is serialized. This means that if something happens and the message cannot be de-persisted from the message queue, you can take a look at the message itself and likely fix the problem.

Another advantage is that messages serialized with this formatter do not have to be deserialized by the same formatter object. That is, you do not need to have the same formatter class in the receiver as you had in the sender, as long as both parties know the data schema. In addition, messages serialized by the XML formatter do not necessarily have to be deserialized at all. Most browsers and other XML viewers can parse XML messages.

See Also

Creating MessageQueue Component Instances | Sending and Serializing Messages

Verifying Messages at Design Time

You can use the Properties window to verify that your message-sending mechanism works correctly. Working with the Message Queues node in Server Explorer, you can select and expand queues to see the messages they contain, then select specific messages and view information about them in the Properties window. The Properties window displays information about the metadata attached to the message — its arrival time, authentication status, correlation ID, and so on — and also allows you to view the body of the message by accessing the BodyStream property.

To verify information about a message at design time

1. From design view, access Server Explorer. For more information, see Accessing and Initializing Server Explorer.

   > **Note** The Servers node of the Server Explorer is not available in the Standard Edition of Visual Basic and Visual C# .NET. For more information, see Visual Basic Standard Edition Features or Visual C# Standard Edition Features.

2. Expand the **Message Queues** node, and locate the queue you want to view information about under **Public Queues**, **Private Queues**, or **System Queues**.
3. Expand the queue you need and then locate the message you want under either **Journal Messages** or **Queue Messages**.
4. Click on a message.

   Information about that message appears in the Properties window.

5. To view the message body, click the **Ellipsis** button ( ) next to the **BodyStream** property.

   The Byte Array Property Editor dialog box appears.

6. Click on **Auto** to view the message's body in XML format, or choose one of the other view options as appropriate.

See Also

Sending and Serializing Messages | Requesting Acknowledgement for Messages | Default Message Properties | Sending Simple Messages | Sending Complex Messages | Creating MessageQueue Component Instances | Verifying Messages at Design Time

Reading and Receiving Messages

There are several ways you can read and receive messages from the queues in your system. First, you can either receive messages from the queue or you can peek at the messages on a queue without removing them. You can also create an event handler that will watch for a new message to arrive and automatically raise an event when the message arrives at the queue. The following topics explain each of these methods of working with queue messages.

In This Section

Introduction to Reading and Retrieving Messages
Provides an overview of formatters and their interaction with reading and receiving messages.
Specifying Properties to Retrieve with a Message
Gives directions for specifying the properties to retrieve when reading messages.
Specifying the Formatter for Retrieved Messages
Gives directions on how to specify the formatter you want to use to retrieve messages.
Receiving Messages Programmatically
Gives directions for writing code to read and receive messages.
Peeking at Messages
Gives directions for looking at messages without removing them from the queue.
Asynchronous Message Processing
Describes the process of retrieving messages asynchronously.
Transactional Message Processing
Explains how messages can be processed as part of a transaction.
Queue and Message Collections
Provides basic information on retrieving collections of queues and messages.
Retrieving Messages
Provides instructions on retrieving lists of messages.
Retrieving Queues
Provides instructions on retrieving lists of message queues.

Related Sections

Message Serialization
Provides an introduction to formatters and the process of serializing and deserializing messages.
Introduction to Messaging
Provides an introduction to messaging in Visual Studio and the .NET Framework.
Creating Messaging Components
Lists the major topics on using messaging in your applications.
Message Queues and Messaging Technology Backgrounder
Provides background explanation on message queuing technology.
Walkthrough: Creating a Message Queue Transaction with COM+ Services
Creates a message queue transaction hosted by Enterprise Services (COM+ Services).

Introduction to Reading and Retrieving Messages

There are several considerations to keep in mind when reading and retrieving messages.

Locking Access to a Queue

When you perform a receive operation, you may want to temporarily prevent other users from also removing messages from the queue with which you are working. You can do this by setting the DenySharedReceive property on that queue to **true**. Setting **DenySharedReceive** prevents any other users from removing messages from the same queue until you release your usage of it, either through garbage collection (For more information, see Reference Counting, Garbage Collection, and Object Lifetime) or by calling the Close method.

Properties to Retrieve

Messages have a large number of properties, and filtering the properties you want to receive can enable you to focus on only the information that is relevant to you. For example, if you are not working with transactional messages, you might not want to retrieve the properties associated with transactions. In this case, you would set the MessagePropertyFilter properties IsFirstInTransaction, IsLastInTransaction, TransactionID, and TransactionStatusQueue to **false**. Your **MessageQueue** component instance would not retrieve any of the property values corresponding to these properties whenever it gets a message from a queue. For more information, see Specifying Properties to Retrieve with a Message.

Specifying the Format for Received Messages

When you read messages from a queue, a formatter object is used to serialize and deserialize the message's contents as you manipulate the message. By default, an XMLMessageFormatter object is associated with any instance of the **MessageQueue** component you create, and you can use this to receive messages by setting a few necessary properties on it. You can also use various other types of formatter objects that are available to you; for more information on the formatters available, see Message Serialization.

The **XMLMessageFormatter** persists objects and primitive data types into and out of messages using human-readable XML strings. When you want to use this formatter to retrieve a message, you set a **TargetTypeNames** or **TargetTypes** property to indicate how the resulting message should be handled. For more information, see TargetTypeNames Property. If you are using another formatter object, the properties differ.

See Also

Specifying the Formatter for Retrieved Messages | Receiving Messages Programmatically | Peeking at Messages | Asynchronous Message Processing | Message Serialization

Specifying Properties to Retrieve with a Message

When you create an instance of the **MessageQueue** component, you can set a series of properties to indicate what properties you want to retrieve when the component gets a message from a queue. These properties exist in a class called MessagePropertyFilter Class and correspond to actual properties on the Message class. When you set the value for one of these properties to **true**, the component will retrieve the corresponding property each time a message is removed from the queue.

> **Tip** If you do not need access to some properties in your retrieved message and want to improve performance, you can set the **MessagePropertyFilter** to false.

For a full list of all of the properties you can set, see MessagePropertyFilter Class. By default, the properties retrieved with a message include properties indicating the time it was sent and arrived, several identifiers for the message, the message's label, the response queue, and the body of the message.

To specify the properties to retrieve with a message

1. Create an instance of the **MessageQueue** component and set its Path property to the queue to which you want to refer. For more information, see Creating MessageQueue Component Instances.
2. In the Properties window, expand the **MessageReadPropertyFilter** node.
3. Set the value for the properties you want to receive to **true**.

> **Note** You can set these in code, as well. When doing so, you can use the SetAll method to set all the filter properties to **true**, or the ClearAll method to set all the filter properties to **false**.

See Also

Reading and Receiving Messages | Receiving Messages Programmatically | Peeking at Messages | Receiving Messages Asynchronously | Creating MessageQueue Component Instances

Specifying the Formatter for Retrieved Messages

You must indicate the formatter object you want to use to retrieve a message from a queue. A formatter indicates how a message will be deserialized when it is removed from the queue. Depending on the type of formatter you use, you might need to set additional properties to specify how the formatter should proceed.

By default, you use the **XMLMessageFormatter** for most operations. For more information on the formatters available to you, see Message Serialization. For information on the properties unique to the **XMLMessageFormatter** object, see Introduction to Reading and Retrieving Messages.

You can specify the formatter in code or in the Properties window. In addition, you can either set the formatter on the queue or on the message. You set the formatter on the queue when you plan to access the queue's messages directly, as in the following code:

```
' Visual Basic
Console.WriteLine(CStr(MessageQueue1.Receive().Body))

// C#
Console.WriteLine(MessageQueue1.Receive().Body.ToString());
```

You set the formatter on the message when you plan to retrieve a queue's contents through the message object:

```
' Visual Basic
Message1 = MessageQueue1.Receive()

// C#
message1 = messageQueue1.Receive();
```

To specify the formatter in the Properties window

1. Create and configure your **MessageQueue** component instance. For more information, see Creating MessageQueue Component Instances.
2. In the designer, click on the **MessageQueue** instance, and then access the Properties window.
3. Set the **Formatter** property to the appropriate value.
4. If you chose either the XMLMessageFormatter object or the BinaryMessageFormatter object, expand the property and set the necessary subproperties to configure your formatter.

To specify the formatter programmatically

1. Create and configure your **MessageQueue** component instance. For more information, see Creating MessageQueue Component Instances.
2. In the Code Editor, create a formatter object of type XMLMessageFormatter, ActiveXMessageFormatter, or BinaryMessageFormatter.
3. Set any necessary properties for your formatter.
4. Retrieve your message.

For example, the following code shows how to create and configure a component that uses the **XMLMessageFormatter** object to retrieve a message from a public queue called MyQueue.

```
' Visual Basic
Dim mq As New System.Messaging.MessageQueue(".\MyQueue")
Dim formatter As System.Messaging.XmlMessageFormatter = _
   CType(mq.Formatter, System.Messaging.XmlMessageFormatter)
formatter.TargetTypeNames = New String() {"System.String"}
Dim m As System.Messaging.Message = mq.Receive(New TimeSpan(0, 0, 3))


// C#
System.Messaging.MessageQueue mq =
   new System.Messaging.MessageQueue(".\\MyQueue");
string[] types = { "System.String" };
((System.Messaging.XmlMessageFormatter)mq.Formatter).TargetTypeNames =
   types;
```

```
System.Messaging.Message m = mq.Receive(new TimeSpan(0,0,3));
```

See Also

Receiving Messages Programmatically | Introduction to Reading and Retrieving Messages

Receiving Messages Programmatically

You can use a synchronous method called Receive to look at the contents of a queue. When you call the **Receive** method on a queue, the system removes the first message from the queue and returns it to you. This message is no longer available to other components looking at the queue.

> **Note** You can also look at the first message on a queue without removing it from the queue. For more information, see Peeking at Messages. You can also receive messages asynchronously. For more information, see Receiving Messages Asynchronously.

If there are no messages available on the queue when you call the **Receive** method, the method will wait until a message arrives on the queue. You can specify a time-out period if you want the method to wait for only a specified interval. The time-out period is specified as a TimeSpan object.

When you read messages from a queue, a formatter object is used to serialize and deserialize the message's contents as you manipulate the message. For more information, see Introduction to Reading and Retrieving Messages.

There are several forms of the **Receive** method — the basic method with several overloads that allow you to specify things such as time-out periods, a method called MessageQueue.ReceiveById that allows you to retrieve a particular message by its identifier, and a method called ReceiveByCorrelationID that allows you to retrieve a message by correlation ID. For an explanation of correlation IDs, see Message Queue Journal Storage.

To receive a message programmatically

1. Create an instance of the **MessageQueue** component and set its Path property to the queue to which you want to refer. For more information, see Creating MessageQueue Component Instances.
2. Specify the formatter to use to retrieve your message. For more information, see Specifying the Formatter for Retrieved Messages.
3. Create an instance of the **Message** object to hold the retrieved message.
4. Call the **Receive** method to remove the message from the queue.

   > **Tip** To specify a time-out for the **Receive** method, use a TimeSpan object to specify the length of time that you want the method to wait.

   When you are finished, your code might look like this:

```
' Visual Basic
Dim mq As New System.Messaging.MessageQueue(".\MyQueue")
mq.Send("1", "1")
Dim m As System.Messaging.Message
m = mq.Receive(New TimeSpan(0, 0, 3))
m.Formatter = New _
   System.Messaging.XmlMessageFormatter(New String()
   {"System.String,mscorlib"})
Console.WriteLine(m.Body)

// C#
System.Messaging.MessageQueue mq =
   new System.Messaging.MessageQueue(".\\MyQueue");
mq.Send("1", "1");
System.Messaging.Message m = mq.Receive(new TimeSpan(0,0,3));
m.Formatter = new System.Messaging.XmlMessageFormatter(new string[]
   {"System.String,mscorlib"});
Console.WriteLine(m.Body);
```

See Also

Reading and Receiving Messages | Peeking at Messages | Receiving Messages Asynchronously | Creating MessageQueue Component Instances | Specifying the Formatter for Retrieved Messages

Peeking at Messages

You can use the Peek method to look at the first message on any queue without removing that message from the queue. This allows your component to get information from the queue without preventing other applications or components from retrieving messages they were intended to process.

> **Note** **Peek** allows you to see only the first message on the queue. Because that message is not removed from the queue when you peek at it, you cannot then peek at subsequent messages. If you want to see all of the messages in a queue without removing them from the queue, you can use the GetAllMessages method or the GetMessageEnumerator method. For more information, see Queue and Message Collections.

If there are no messages in the queue when you call the **Peek** method, the method waits until a message arrives. You can specify a time-out period if you want the method to wait only a specified period of time. The time-out period is specified as a TimeSpan object. Most commonly, the time-out period will be set to either zero, in which case it checks for a message and does not wait at all, or to the default infinite setting, which waits indefinitely. You would set this in code using the following syntax:

```
' Visual Basic
msg.TimeToBeReceived = System.Messaging.Message.InfiniteTimeout

// C#
msg.TimeToBeReceived = System.Messaging.Message.InfiniteTimeout;
```

You can also set this in the Properties window by setting the TimeToBeReceived property for **DefaultMessagestoSend** to Infinity.

> **Tip** When using dependent clients, be sure the clock on the client computer is synchronized with the clock on the server that is running Message Queuing. Otherwise, unpredictable behavior might result when sending a message whose **TimeToBeReceived** property is not **InfiniteTimeout**.

Peeking can be synchronous or asynchronous. For more information, see Receiving Messages Asynchronously.

To peek at messages synchronously

1. Create an instance of the **MessageQueue** component and set its Path property to the queue to which you want to refer. For more information, see Creating MessageQueue Component Instances.
2. Create an instance of the **Message** object to hold the data the **Peek** method copies from the queue.
3. Call the **Peek** method to get data about the first message on the queue. Optionally, to specify a time-out for the **Peek** method, enter the length of time (as a **TimeSpan** object) that you want the method to wait as an argument of the method.

   For example, the following code shows how you can use the **Peek** method to return and display information about the first message on a queue. This example assumes that you are working on a Windows Forms application that contains a label called Label1, and that you are working with a queue called MyQueue.

```
' Visual Basic
Public Sub LookFirstMessage()
   Dim NewQueue As New System.Messaging.MessageQueue(".\MyQueue")
   Dim FirstMessage As System.Messaging.Message
   FirstMessage = NewQueue.Peek
   Dim targetNames() As String = {"System.String,mscorlib"}
   FirstMessage.Formatter = _
      New System.Messaging.XmlMessageFormatter(targetNames)
   Dim label As String = CStr(FirstMessage.Label)
End Sub

// C#
public void LookFirstMessage()
{
   System.Messaging.MessageQueue newQueue =
      new System.Messaging.MessageQueue(".\\MyQueue");
   System.Messaging.Message firstMessage;
   string[] types = { "System.String,mscorlib" };
```

```
    firstMessage = newQueue.Peek();
    firstMessage.Formatter =
       new System.Messaging.XmlMessageFormatter(types);
    string label = firstMessage.Label.ToString();
 }
```

See Also

Reading and Receiving Messages | Receiving Messages Programmatically | Creating MessageQueue Component Instances

Asynchronous Message Processing

You can retrieve messages asynchronously if you want to retrieve messages without tying up your application's processing. In asynchronous message processing, the method that starts a task is returned immediately without waiting for a result. The application can continue to do what it was originally doing while the task is completed. When the task completes, the server can notify the application that the message was successfully processed.

There are two types of asynchronous messaging operations — receiving messages asynchronously and peeking at messages asynchronously. When you retrieve a message asynchronously, you use the BeginReceive method and the EndReceive method to mark the beginning and end of the operation. The actions that occur are as follows:

- The **BeginReceive** method returns immediately and raises an event called ReceiveCompleted when a message becomes available on the queue or if the message you are peeking at or receiving already exists.
- The **ReceiveCompleted** method returns an object of type IAsyncResult that contains information about the asynchronous operation.
- After the completed event is received, you call the **EndReceive** method to complete the operation. Within the end call, you might access the message or retrieve it by accessing the ReceiveCompletedEventArgs class.

You can access the **IAsyncResult** object throughout the lifetime of the operation, but typically you will not use it until **EndReceive** is called. However, if you start several asynchronous operations, you can place their **IAsyncResult** values in an array and specify whether to wait for all operations or any individual operation to complete. In this case, you use the AsyncWaitHandle property of the **IAsyncResult** object to identify completed operations.

Peek, like **Receive**, uses two methods called BeginPeek and EndPeek to bracket the beginning and end of the asynchronous operation. **BeginPeek** returns immediately and raises an event called PeekCompleted when a message becomes available. Like **ReceiveCompleted**, this event returns an **IAsyncResult** object you can manipulate for information about the operation.

In addition, both asynchronous receive and peek operations can use a time-out period to specify how long you want to wait for a message to become available. To do so, you use an overloaded form of either method to pass a TimeSpan object that indicates the time to wait. The **ReceiveCompleted** or **PeekCompleted** event is raised if the time-out period expires, but the IsCompleted property on the **IAsyncResult** object is set to false to indicate that a message was not dealt with.

For more information on asynchronous processing, see the following topics:

For	See
Step-by-step instructions on receiving a message asynchronously	Receiving Messages Asynchronously
An overview of the **BeginPeek** method and its members	MessageQueue.BeginPeek Method
An overview of the **BeginReceive** method and its members	MessageQueue.BeginReceive Method

Receiving Notification of Your Asynchronous Operation

There are two ways you can receive notification when your asynchronous receive or peek operation completes successfully:

- You can create an event handler that will handle the **ReceiveCompleted** or **PeekCompleted** events when they occur.
- You can use a callback to automatically watch for incoming messages and pass processing to your application when a message arrives.

When you use event notification, you create a method that handles your message processing and returns a notification when the processing completes. You then call the method that begins the asynchronous processing. The system creates the event handlers for you automatically when you double-click your **MessageQueue** component in the designer.

> **Note** In the event-notification scenario, **BeginPeek** or **BeginReceive** returns a single message and then stops processing. You must call **BeginPeek** or **BeginReceive** again for each message you want to retrieve.

An alternate way to process messages asynchronously is to use a callback. A callback identifies a delegate you want to associate with your **BeginPeek** or **BeginReceive** operations. In this scenario, the delegate continues to watch for new event notification after each message is processed.

See Also

Receiving Messages Asynchronously | Reading and Receiving Messages | Peeking at Messages | Receiving Messages Programmatically

Receiving Messages Asynchronously

You can retrieve messages asynchronously in two ways: by using an event handler to receive notification when the operation finishes processing, or by using a callback. For an overview of asynchronous messaging, see Asynchronous Message Processing.

In event notification, you first bind an event handler to the procedure you want to run when the asynchronous call completes. You then call the BeginReceive method in your code. This starts the asynchronous processing and returns processing to your component when a message becomes available or when the time-out parameter expires. When the call returns, the system runs the delegate you have defined and processes the results of the retrieval. You then call the EndReceive method to indicate the end of the operation.

> **Note** **BeginReceive** will retrieve only a single message. If you want to continue processing messages asynchronously, you must either call the **BeginReceive** method again or use the callback parameter on **BeginReceive** to call a delegate that will continue to watch for new messages arriving on the queue.

In addition to receiving messages asynchronously, you can peek at messages asynchronously. The pattern for both processes is very similar, except that to peek asynchronously, you use the BeginPeek method.

To create the MessageQueue object programmatically

1. Add a reference to System.Messaging.dll to your project.
2. Create an instance of the **MessageQueue** object in the class you are implementing, and set its Path property (in the constructor) and the Formatter property.

```
' Visual Basic
' Add this to the class declarations.
Dim WithEvents mq As New System.Messaging.MessageQueue(".\MyQueue")
' Add this to the constructor
Dim targetTypeNames() As String = _
   New String() {"System.String,mscorlib"}
mq.Formatter = _
   New System.Messaging.XmlMessageFormatter(targetTypeNames)

// C#
// Add this to the class declarations.
System.Messaging.MessageQueue mq =
   new System.Messaging.MessageQueue(".\\MyQueue");
// Add this to the constructor.
mq.Formatter = new System.Messaging.XmlMessageFormatter(
   new string[]{"System.String,mscorlib"});
```

To create the MessageQueue object in the Designer

1. Add a reference to System.Messaging.dll to your project.
2. From the Toolbox, drag a **MessageQueue** component onto the Designer. Set the **Name** property to **mq**. Set the **Formatter** property to **XmlMessageFormatter**. Set the **Path** property to **".\MyQueue"**.

To receive a message asynchronously using event notification

1. Create an event handler for the ReceiveCompleted event. In the designer, double-click the **MessageQueue** component, and add the code shown below.

```
' Visual Basic
Private Sub mq_ReceiveCompleted(ByVal sender As System.Object, _
ByVal e As System.Messaging.ReceiveCompletedEventArgs) _
Handles mq.ReceiveCompleted
   ' Add code here to respond to message.
End Sub

// C#
```

```
private void mq_ReceiveCompleted(object sender,
System.Messaging.ReceiveCompletedEventArgs e)
{
   // Add code here to respond to message.
}
```

2. Write code within the event handler to retrieve the message, using the **Message** object to retrieve the results of the asynchronous call. The following code retrieves the message and displays it on the console.

```
' Visual Basic
Private Sub mq_ReceiveCompleted(ByVal sender As System.Object, _
ByVal e As System.Messaging.ReceiveCompletedEventArgs) _
Handles mq.ReceiveCompleted
   Dim m As System.Messaging.Message = mq.EndReceive(e.AsyncResult)
   m.Formatter = New System.Messaging.XmlMessageFormatter( _
      New String() {"System.String,mscorlib"})
   Console.WriteLine("Message: " + m.Body.ToString())
End Sub

// C#
private void mq_ReceiveCompleted(object sender,
System.Messaging.ReceiveCompletedEventArgs e)
{
   System.Messaging.Message m = mq.EndReceive(e.AsyncResult);
   m.Formatter = new System.Messaging.XmlMessageFormatter(
      new string[]{"System.String,mscorlib"});
   Console.WriteLine("Message: " + (string)m.Body);
}
```

3. Call the **BeginReceive** method somewhere in your code to start the asynchronous operation. For example, the following code calls the method when the user clicks a button.

```
' Visual Basic
Private Sub Button1_Click(ByVal sender As System.Object, _
ByVal e As System.EventArgs) Handles Button1.Click
   mq.BeginReceive()
End Sub

// C#
private void button1_Click(object sender, System.EventArgs e)
{
   mq.BeginReceive();
}
```

4. If you want to continue receiving messages asynchronously, call the **BeginReceive** method again in the **ReceiveCompleted** event handler, shown below. This makes your component continue processing new messages as they are received on the queue.

```
' Visual Basic
Private Sub mq_ReceiveCompleted(ByVal sender As System.Object, _
ByVal e As System.Messaging.ReceiveCompletedEventArgs) _
Handles mq.ReceiveCompleted
   Dim m As System.Messaging.Message = mq.EndReceive(e.AsyncResult)
   m.Formatter = New System.Messaging.XmlMessageFormatter( _
      New String() {"System.String,mscorlib"})
      Console.WriteLine("Message: " + m.Body.ToString())
   mq.BeginReceive()
End Sub
```

```
// C#
private void mq_ReceiveCompleted(object sender,
System.Messaging.ReceiveCompletedEventArgs e)
{
   System.Messaging.Message m = mq.EndReceive(e.AsyncResult);
   m.Formatter = new System.Messaging.XmlMessageFormatter(
      new string[]{"System.String,mscorlib"});
   Console.WriteLine("Message: " + (string)m.Body);
   mq.BeginReceive();
}
```

To receive a message asynchronously using a callback

1. Create a class that defines information relevant to the message task. In this case, define a `Customer` class.

```
' Visual Basic
Public Class Customer
    Public Name As String = ""
    Public Sub New(ByVal newName As String)
        Name = newName
    End Sub
End Class

// C#
public class Customer
{
    public string Name = "";
    public Customer(string name)
    {
        Name = name;
    }
}
```

2. Create an instance of the class. This object will be passed to the callback method.

```
' Visual Basic
Dim george As New Customer("George")

// C#
Customer george = new Customer("George");
```

3. Create a callback method according to the AsyncCallback delegate. This method will contain the processing to be done after the message is received. The IAsyncResult.AsyncState property of the **result** parameter will contain the object you created to pass information about the message task. In this case, the **AsyncState** is a `Customer` object.

```
' Visual Basic
Private Sub ReceiveCallback(ByVal result As System.IAsyncResult)
    Dim buyer As Customer = CType(result.AsyncState, Customer)
    Dim buyerName As String = buyer.Name
End Sub

// C#
private void ReceiveCallback(System.IAsyncResult result)
{
    Customer buyer = (Customer)result.AsyncState;
    string buyerName = buyer.Name;
}
```

4. Call the MessageQueue.BeginReceive (TimeSpan, Object, AsyncCallback) method somewhere in your code to start the

asynchronous operation. For example, the following code calls the method when the user clicks a button. A message is sent to the message queue and then read from the queue. The **ReceiveCallback** method, defined in step 3 above, is called when the message is received.

```
' Visual Basic
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles Button1.Click
    MessageQueue1.Send("Buy six eggs.", george.Name)
    MessageQueue1.BeginReceive(New System.TimeSpan(0, 0, 5), george, _
        AddressOf ReceiveCallback)
End Sub

// C#
private void button1_Click(object sender, System.EventArgs e)
{
    messageQueue1.Send("Buy six eggs.", george.Name);
    messageQueue1.BeginReceive(new System.TimeSpan(0, 0, 5), george,
        new System.AsyncCallback(this.ReceiveCallback));
}
```

See Also

Asynchronous Message Processing | Reading and Receiving Messages | Peeking at Messages | Receiving Messages Programmatically | Creating MessageQueue Component Instances

Transactional Message Processing

Transactional processing allows you to ensure that the messages in a transaction are delivered in order, are delivered only once, and are successfully retrieved from the destination queue. The **MessageQueue** component can be used to send and receive messages transactionally. When you send messages inside a transaction, you group a set of related messages. All the messages contained in the transaction are either delivered together in the order in which they were sent (a *committed* transaction), or automatically not sent or received if any problems occur (an *aborted* transaction).

There are two main types of transactions you can create from your applications:

- *Internal transactions* are used to send messages between two or more Message Queuing resources — that is, between two queues belonging to a Message Queuing enterprise.
- *External transactions* are used to send messages between queues and other resources, such as databases.

Internal and external transactions differ in their programming model, in the resource manager used to govern the transaction, and in their usage patterns.

Internal Transactions

Internal transactions are executed by creating an instance of the MessageQueueTransaction class and associating it with an instance of the **MessageQueue** component. An internal transaction is the simplest type of transaction. Messages are sent between one or more message queues, and Message Queuing's transaction coordinator governs the actions that occur. Because you control these steps in the process manually, internal transactions can be thought of as manual or explicit transactions, whereas external transactions are automatic or implicit.

The programming model for internal transactions are very simple — you call the Begin method of the **MessageQueueTransaction** class and pass an instance of this class to the sending or receiving method. You then call Commit to save the transaction's changes to the destination queue. For more information, see Sending Messages Within Internal Transactions.

Internal transactions are somewhat limited in functionality, in that you cannot perform transactions with non-message queuing resources such as databases. If you want to interact with a database in a transaction, you must use external transactions. However, in spite of these limitations, internal transactions have better performance than external transactions.

Both internal and external transactions require that you send the messages to a queue that has been marked as transactional. For more information, see Creating Transactional Queues.

External Transactions

External transactions are typically used when you want to send messages between queues and another type of resource. For example, you might want to retrieve a message from a queue and send it to a database, or vice versa. External transactions differ from internal transactions in several ways:

- External transactions rely on a coordinator that is not part of the Message Queuing system. In most cases, this will be the Microsoft Distributed Transaction Coordinator (DTC). The DTC enlists the necessary resources in the transaction and governs the process of determining whether all of the actions in the transaction succeed or fail as a unit.
- An external transaction has a more complex programming model. Instead of simply calling **Begin**, **Commit**, and **Abort**, you specify attributes that indicate an external transaction and register the component with COM+ 1.0 Services. You also must use a special form of the Send and Receive methods that takes a parameter for the type of transaction, and set this field to Automatic.

There are also other types of transactions that can be sent without using the **MessageQueue** component. For more information on transactions in Visual Studio .NET or the .NET Framework, see the following topics:

For more information on	See
An overview of the types of transactions supported by the .NET Framework	Transaction Types
An example demonstrating the attributes used with external transactions	See MSDN online.
An explanation of the Message Queuing features relating to transactions	Manual Transactions and Message Queuing
Factors to keep in mind when working with transactions	An Introduction to Microsoft Transaction Server
Transactions in C++	Transactions

See Also

Reading and Receiving Messages | Creating Transactional Queues | Sending Messages Within Internal Transactions | Walkthrough: Creating a Message Queue Transaction with COM+ Services

Creating Transactional Queues

A transactional queue is created for the specific purpose of receiving and processing messages that are sent as part of a transactional activity. Nontransactional queues are unable to process messages that have been included in a transactional activity.

To send messages as part of a transactional activity, the receiving queue must be set to process transactions. This can be done programmatically by setting its Transactional property to **true**.

To create a transactional queue

1. Add a reference to the **System.Messaging.dll**. For more information, see Adding and Removing References.
2. Use the overloaded Create constructor to create a public, transactional queue, setting the second parameter to **true** to indicate that the queue should be transactional.

```
' Visual Basic
Dim mq2 As New System.Messaging.MessageQueue
mq2 = System.Messaging.MessageQueue.Create(".\MyTransQueue", True)

// C#
System.Messaging.MessageQueue mq2 =
   new System.Messaging.MessageQueue();
mq2 = System.Messaging.MessageQueue.Create(".\\MyTransQueue",true);
```

Note This syntax creates a public transactional queue. The exact syntax you use in the first parameter depends on the kind of queue you are trying to create. For more information, see Queue Reference Recommendations.

See Also

Reading and Receiving Messages | Transactional Message Processing | Sending Messages Within Internal Transactions | Creating Queues | Adding and Removing References | Queue Reference Recommendations | Walkthrough: Creating a Message Queue Transaction with COM+ Services

Sending Messages Within Internal Transactions

Internal transactions involve only Message Queuing resources and are managed entirely from within Message Queuing with the MessageQueueTransaction Class. You instantiate an instance of **MessageQueueTransaction** to handle the transaction process and pass it to the message you are choosing to send as a transaction.

The following methods control the transactional process:

Method	Used to
Begin	Indicate the start of a transaction that consists of one or more messages and operations.
Commit	Used to commit the transaction if all of the message operations it contained were performed successfully.
Abort	Used during error-checking to roll back a transaction.

Note A disconnected or full queue is not considered a failure.

In most cases, your transactional code should be enclosed within a Try...Catch statement. This allows you to specify the transaction in the **Try** portion of the statement, and then use the **Catch** statement to issue the **Abort** if any errors occur. See the code below for an example of this pattern.

To send messages within an internal transaction

1. Create an instance of the **MessageQueue** component and set its Path property to the queue to which you want to refer. For more information, see Creating MessageQueue Component Instances.
2. Create an instance of the MessageQueueTransaction class to control the transaction.
3. Call the **Begin** method of the **MessageQueueTransaction** class.
4. Define each message you want to send after calling **Begin**. For more information, see Creating MessageQueue Component Instances.
5. After the last message is sent, call **Commit** on the **MessageQueueTransaction** instance to close the transaction.
6. Enclose the entire transaction in **Try...Catch** error-checking code to handle any errors that arise.
7. If you detect an error, roll back the transaction by calling the **Abort** method.

   Your code might look like this:

```
' Visual Basic
Dim transaction As New System.Messaging.MessageQueueTransaction
Try
    transaction.Begin()
    MessageQueue1.Send("Message1", "Label1", transaction)
    MessageQueue1.Send("Message2", "Label2", transaction)
    MessageQueue1.Send("Message3", "Label3", transaction)
    transaction.Commit()
Catch
    transaction.Abort()
End Try

// C#
System.Messaging.MessageQueueTransaction transaction =
    new System.Messaging.MessageQueueTransaction();
try
{
    transaction.Begin();
    messageQueue1.Send("Message1", "Label1", transaction);
    messageQueue1.Send("Message2", "Label2", transaction);
    messageQueue1.Send("Message3", "Label3", transaction);
    transaction.Commit();
}
catch
{
    transaction.Abort();
}
```

See Also

Transactional Message Processing | Creating Transactional Queues | Creating MessageQueue Component Instances | Sending and Serializing Messages

Walkthrough: Creating a Message Queue Transaction with COM+ Services

This walkthrough creates a transaction supported by COM+ Services by using the **ServicedComponent** class. The walkthrough creates two transactional queues and moves data between them in a transaction hosted by Enterprise Services (COM+ Services). For an overview of Enterprise Services see Writing Serviced Components.

> **Note** You must install the Message Queuing components of your operating system to complete this walkthrough.

The first step in using a COM+ transaction service is to create a class that derives from the **ServicedComponent** class, found in the System.EnterpriseServices namespace. The **ServicedComponent** class is the base class used to create components that are hosted by COM+.

To create the ServicedComponent Class

1. Create a class library project. Name it TransactionComponent.
2. Add references to System.Messaging.dll and System.EnterpriseServices.
3. Replace Class1 with the class below.

   - The TransactionAttribute on the class directs the Distributed Transaction Coordinator to create a context for the object.
   - Classes derived from **ServicedComponent** must have a public default (parameterless) constructor.
   - Fields are added for the source and destination queues. These are exposed as properties, because the constructor cannot take parameters.
   - The AutoComplete attribute on the **Move** method indicates that at the end of the method, the component's vote towards the transaction is based on failure or success of this method. The method fails if an exception is thrown. If the method succeeds, the ContextUtil.SetComplete Method is called. For more information, see AutoCompleteAttribute Class and Voting in an Automatic Transaction.

```
' Visual Basic
<System.EnterpriseServices.TransactionAttribute( _
System.EnterpriseServices.TransactionOption.Required)> _
Public Class MessageMover
    Inherits System.EnterpriseServices.ServicedComponent

    Private sourceQueue As System.Messaging.MessageQueue
    Private destinationQueue As System.Messaging.MessageQueue

    Public Sub New()
    End Sub

    Public Property Source() As System.Messaging.MessageQueue
        Get
            Return sourceQueue
        End Get
        Set(ByVal Value As System.Messaging.MessageQueue)
            sourceQueue = Value
        End Set
    End Property

    Public Property Destination() As System.Messaging.MessageQueue
        Get
            Return destinationQueue
        End Get
        Set(ByVal Value As System.Messaging.MessageQueue)
            destinationQueue = Value
        End Set
    End Property

    <System.EnterpriseServices.AutoComplete()> _
```

```
    Public Sub Move()
        Dim sourceMessage As System.Messaging.Message
        sourceMessage = sourceQueue.Receive( _
            System.Messaging.MessageQueueTransactionType.Automatic)
        destinationQueue.Send(sourceMessage, _
            System.Messaging.MessageQueueTransactionType.Automatic)
    End Sub
End Class

// C#
[System.EnterpriseServices.TransactionAttribute(
System.EnterpriseServices.TransactionOption.Required)]
public class MessageMover : System.EnterpriseServices.ServicedComponent
{
   private System.Messaging.MessageQueue sourceQueue;
   private System.Messaging.MessageQueue destinationQueue;
   public MessageMover()
   {
   }

   public System.Messaging.MessageQueue Source
   {
      get
      {
         return sourceQueue;
      }
      set
      {
         sourceQueue = value;
      }
   }
   public System.Messaging.MessageQueue Destination
   {
      get
      {
         return destinationQueue;
      }
      set
      {
         destinationQueue = value;
      }
   }
   [System.EnterpriseServices.AutoComplete()]
   public void Move()
   {
      System.Messaging.Message sourceMessage;
      sourceMessage = sourceQueue.Receive(
         System.Messaging.MessageQueueTransactionType.Automatic);
      destinationQueue.Send(sourceMessage,
         System.Messaging.MessageQueueTransactionType.Automatic);
   }
}
```

To be hosted by COM+, components must have a strong name. The following procedure describes how to create a key file and use it to apply a strong name to the class library.

To give the class a strong name

1. Locate the Strong Name Tool (Sn.exe) and run it to create a key file, key.snk, in the project directory. The command looks

something like this:

```
sn -k ProjectDirectory\key.snk
```

2. In Solution Explorer, double-click AssemblyInfo.vb or AssemblyInfo.cs, depending on the language you are using. Add or replace the attributes for AssemblyDelaySign, AssemblyKeyFile, and AssemblyKeyName. The file location for the key file is given relative to the build location. You will need to confirm that the path is correct for your project configuration. For more information on the interaction of these three attributes, see Global Attributes.

```
' Visual Basic
<Assembly: AssemblyDelaySign(False)>
<Assembly: AssemblyKeyFile("..\..\key.snk")>
<Assembly: AssemblyKeyName("")>
// C#
[assembly: AssemblyDelaySign(false)]
[assembly: AssemblyKeyFile("..\\..\\key.snk")]
[assembly: AssemblyKeyName("")]
```

3. Build the application to create the DLL.
4. Now that the assembly has a strong name, it can be registered with COM+. The .NET Framework includes a tool for registering serviced components.

To register the component with COM+

1. Locate the .NET Services Installation Tool (Regsvcs.exe), located in the .NET Framework installation folder. The .NET Framework is installed in the operating system folder, not in the **Program Files** folder.
2. Run the tool to register TransactionComponent.dll. The command looks something like this:

```
regsvcs c:\DllPath\TransactionComponent.dll
```

> **Note** The registration tool may report that "The class TransactionComponent.MessageMover has no class interface, which means that unmanaged late bound calls cannot take advantage of AutoComplete methods." The message is a warning that because the component does not implement an interface, you will not be able to use the component in a late-bound call, such as if you were using VBScript. The component will work in this walkthrough.

You can use the Component Services tool to see the COM+ component you created.

To verify the results with Component Services

1. Open Control Panel.
2. Double-click the **Administrative Tools** icon.
3. Open **Component Services**.
4. Double-click the Component Services node under the Console Root to expand it.
5. Expand the **Computers** node.
6. Expand the **My Computer** node.
7. Expand the **COM+ Applications** node.
8. Find the TransactionComponent entry. Its presence verifies that the component is registered with Component Services. When you create an instance of TransactionComponent, it will be hosted in a COM+ Services context. The code executed (the **ProcessOrder** method) will run in the managed space, but the context (which tracks the success or failure of the transaction) will be maintained by COM+ Services.

The client application will be a Windows Form with code that is able to create and delete a message queue. The application will include two methods that use the COM+ transaction. The first will be a successful transaction. The second will be coded so that the transaction fails. This will allow you to experiment with the exception-handling features of COM+.

To create the client application

1. Add a new Windows Application project to the solution. Name it MessageMoverClient.
2. Add a reference to the TransactionComponent project. To use the .NET serviced component, you add a reference to the DLL

and instantiate the class.

3. Add references to System.Messaging.dll and System.EnterpriseServices.
4. Add a button to the form. Name the button **moveSucceeds** and set the Text property to **Succeed**. This button will execute code that successfully moves a new message from one queue to another in a complete transaction.
5. Create the Click event handler for the **moveSucceeds** button and add the code below to the method.

> **Note** This code creates and deletes message queues named SourceQueue and DestinationQueue. If these queues already exist on your computer, they may be deleted in the course of this walkthrough. Use the Computer Management application in the Control Panel item named **Administrative Tools** to look for these queues. See Step 7 below, for information on using the Computer Management application to examine message queues.

```
' Visual Basic
Dim mover As New TransactionComponent.MessageMover
Dim sourcePath As String = ".\SourceQueue"
Dim destinationPath As String = ".\DestinationQueue"

If System.Messaging.MessageQueue.Exists(sourcePath) Then
    Dim queue As New System.Messaging.MessageQueue(sourcePath)
    If (Not queue.Transactional) Then
        System.Messaging.MessageQueue.Delete(sourcePath)
        System.Messaging.MessageQueue.Create(sourcePath, True)
    End If
Else
    System.Messaging.MessageQueue.Create(sourcePath, True)
End If

If System.Messaging.MessageQueue.Exists(destinationPath) Then
    Dim queue As New System.Messaging.MessageQueue(destinationPath)
    If (Not queue.Transactional) Then
        System.Messaging.MessageQueue.Delete(destinationPath)
        System.Messaging.MessageQueue.Create(destinationPath, True)
    End If

Else
    System.Messaging.MessageQueue.Create(destinationPath, True)
End If

mover.Source = New System.Messaging.MessageQueue(sourcePath)
mover.Destination = New System.Messaging.MessageQueue(destinationPath)

mover.Source.Send("some message", System.Messaging.MessageQueueTransactionType.Single)
mover.Move()

MessageBox.Show("Move complete.")

// C#
TransactionComponent.MessageMover mover;
mover = new TransactionComponent.MessageMover();
string sourcePath = ".\\SourceQueue";
string destinationPath = ".\\DestinationQueue";

if (System.Messaging.MessageQueue.Exists(sourcePath))
{
   System.Messaging.MessageQueue queue = new System.Messaging.MessageQueue(sourcePath);
   if (!queue.Transactional)
   {
      System.Messaging.MessageQueue.Delete(sourcePath);
      System.Messaging.MessageQueue.Create(sourcePath, true);
```

```
    }
}
else
{
    System.Messaging.MessageQueue.Create(sourcePath, true);
}

if (System.Messaging.MessageQueue.Exists(destinationPath))
{
    System.Messaging.MessageQueue queue = new System.Messaging.MessageQueue(destinationPat
h);
    if (!queue.Transactional)
    {
        System.Messaging.MessageQueue.Delete(destinationPath);
        System.Messaging.MessageQueue.Create(destinationPath, true);
    }
}
else
{
    System.Messaging.MessageQueue.Create(destinationPath, true);
}

mover.Source = new System.Messaging.MessageQueue(sourcePath);
mover.Destination = new System.Messaging.MessageQueue(destinationPath);

mover.Source.Send("some message", System.Messaging.MessageQueueTransactionType.Single);
mover.Move();

MessageBox.Show("Move complete.");
```

6. In Solution Explorer, right-click the MessageMoverClient project and select "Set as Startup Project."
7. Run the application and click the **Succeed** button.

   Note If the computer is on a domain, there may be a delay between when the queue is created programmatically and when the queue is available to move messages. This delay may result in a failure of the **Move** method. You can add a call to the Thread.Sleep method to allow the queue to be created, or you can eliminate the code to create the queue and use Server Explorer to create the queue.

8. In the Control Panel item named **Administrative Tools**, open **Computer Management**. Expand the **Services and Applications** node and then the **Message Queuing** and **Public Queues** nodes. Locate the SourceQueue and DestinationQueue. Confirm that a new message was created and exists in DestinationQueue.
9. Add a button to the form. Name the button **moveFails** and set the Text property to **Fail**. This button will execute code that is not able to move a new message from one queue to another, because the destination queue does not exist.
10. Create the Click event handler for the **moveFails** button and add the code below to the method:

```
' Visual Basic
Try
    Dim mover As New TransactionComponent.MessageMover
    Dim sourcePath As String = ".\SourceQueue"
    Dim destinationPath As String = ".\DestinationQueue"

    If System.Messaging.MessageQueue.Exists(sourcePath) Then
        Dim queue As New System.Messaging.MessageQueue(sourcePath)
        If (Not queue.Transactional) Then
            System.Messaging.MessageQueue.Delete(sourcePath)
            System.Messaging.MessageQueue.Create(sourcePath, True)
        End If
    Else
        System.Messaging.MessageQueue.Create(sourcePath, True)
```

```
    End If

    If System.Messaging.MessageQueue.Exists(destinationPath) Then
        Dim queue As New System.Messaging.MessageQueue(destinationPath)
        If (Not queue.Transactional) Then
            System.Messaging.MessageQueue.Delete(destinationPath)
            System.Messaging.MessageQueue.Create(destinationPath, True)
        End If

    Else
        System.Messaging.MessageQueue.Create(destinationPath, True)
    End If

    mover.Source = New System.Messaging.MessageQueue(sourcePath)
    mover.Destination = _
        New System.Messaging.MessageQueue(destinationPath)
    System.Messaging.MessageQueue.Delete(destinationPath)

    mover.Source.Send("some message", _
        System.Messaging.MessageQueueTransactionType.Single)

    mover.Move()

    MessageBox.Show("Move complete.")
Catch
    MessageBox.Show("Didn't move the message.")
End Try

// C#
try
{
    TransactionComponent.MessageMover mover;
    mover = new TransactionComponent.MessageMover();
    string sourcePath = ".\\SourceQueue";
    string destinationPath = ".\\DestinationQueue";

    if (System.Messaging.MessageQueue.Exists(sourcePath))
    {
        System.Messaging.MessageQueue queue =
            new System.Messaging.MessageQueue(sourcePath);
        if (!queue.Transactional)
        {
            System.Messaging.MessageQueue.Delete(sourcePath);
            System.Messaging.MessageQueue.Create(sourcePath, true);
        }
    }
    else
    {
        System.Messaging.MessageQueue.Create(sourcePath, true);
    }

    if (System.Messaging.MessageQueue.Exists(destinationPath))
    {
        System.Messaging.MessageQueue queue =
            new System.Messaging.MessageQueue(destinationPath);
        if (!queue.Transactional)
        {
            System.Messaging.MessageQueue.Delete(destinationPath);
            System.Messaging.MessageQueue.Create(destinationPath,
```

```
                true);
        }
    }
    else
    {
        System.Messaging.MessageQueue.Create(destinationPath, true);
    }

    mover.Source = new System.Messaging.MessageQueue(sourcePath);
    mover.Destination =
        new System.Messaging.MessageQueue(destinationPath);
    System.Messaging.MessageQueue.Delete(destinationPath);

    mover.Source.Send("some message",
        System.Messaging.MessageQueueTransactionType.Single);
    mover.Move();

    MessageBox.Show("Move complete.");
}
catch
{
    MessageBox.Show("Didn't move the message.");
}
```

11. Run the application and click the **Fail** button.
12. In the Control Panel item named **Administrative Tools**, open the C**omputer Management** application. Expand the **Services and Applications** node and then the **Message Queuing** and **Public Queues** nodes. Locate the SourceQueue and DestinationQueue. Confirm that a new message was created and exists in SourceQueue. Even though the message was removed in the **Move** method, the message was returned to the queue when the transaction failed.

Next Steps

The message queues involved in your application may not all reside on the same server. For information on using queues in a domain, see Message Queues and Messaging Technology Backgrounder.

In this example, the information in the queue was a simple string. You may want to move more complicated objects between queues. For more information, see Message Serialization and Introduction to Reading and Retrieving Messages.

See Also

Transactional Message Processing | Creating Queues | MessageQueue Class | Writing Serviced Components

Queue and Message Collections

You can retrieve lists of the queues on your Message Queuing network and the messages they contain by using any of a variety of retrieval methods available in the **MessageQueue** class. You might retrieve lists of queues and messages if you wanted to perform administrative tasks, generate reports about queue contents, or search through the contents of a queue for messages that meet specific criteria.

There are two main ways you can retrieve both messages and message queues: you can retrieve a static snapshot of the messages or queues on the network, or you can iterate through messages and queues using an enumerator. Retrieving a static list of queues or messages is faster than retrieving an enumeration but does not give you as much accuracy in or control over the resulting set of information. You might use a static snapshot of retrieval if you need to perform an administrative task on all public queues in your network.

Enumerators give you finer control over how you interact with the retrieved results. Using an enumerator, you can locate a queue or message that meets certain criteria, instead of evaluating each and every item in the returned collection. For example, you might use an enumerator if you want to review the contents of a queue and work with only those messages for which a particular property has been set to **true**.

Message Queues Retrieval

You can retrieve information about message queues in the following ways:

- You can retrieve a static array of all of the public or private queues on the network.
- You can retrieve a static subset of all the public queues on the network by specifying certain criteria such as the date and time the queue was created or modified, or by specifying a category, label, or machine by which to retrieve queues.
- You can iterate queues on the network using an enumerator, or you can iterate through a subset of all the queues on the network by specifying criteria such as machine name, date and time created, and category.

You can also use the Exists method to determine whether a queue you are interested in exists. The **Exists** method searches for a specific queue by a path you define.

For instructions on retrieving lists of message queues, see Retrieving Queues.

Message Retrieval

When you retrieve messages in an array or enumerator, you do not actually pull those messages off the queue as you do when you read or receive messages. Instead, you retrieve information about the items in the queue, while the actual messages remain available to the users who were intended to receive them.

You can retrieve messages in the following ways:

- You can retrieve a static array of all the messages in a queue.
- You can iterate messages in a queue using an enumerator.

For instructions on retrieving lists of messages, see Retrieving Messages.

Working with Items in an Enumerator

After you create an enumeration of messages or queues, the enumerator is conceptually positioned before the first item of the enumeration. You can call the **GetNext** method to move to the first item in the list and to move through subsequent items. You can also use the **Current** method to return the item you are currently positioned on in the enumeration.

See Also

Retrieving Queues | Retrieving Messages

Retrieving Messages

You can use either static or dynamic retrieval mechanisms to retrieve a list of messages. When you retrieve a static list of messages, the system returns an array of Message objects representing all messages in the queue.

When you retrieve a dynamic list of messages, the system returns a MessageEnumerator object representing all messages in the queue. A **MessageEnumerator** object is a cursor, initialized to the head of a dynamic list. The list order is the same as the order of the messages in the queue, according to message priority. You can move the cursor to the first message in the queue by calling the MoveNext method. After the enumerator has been initialized, you can use **MoveNext** to step forward through the remaining messages. You can specify whether to wait for a message to become available by passing a time-out into the **MoveNext** method.

Because the enumerator is dynamic, it can access a message that is appended beyond the cursor's current position (for example, due to low priority). The enumerator cannot access a message that is inserted before the cursor's current position. It is not possible to step backward with a **MessageEnumerator**. A cursor allows forward-only movement. The Reset method enables you to place the cursor back at the beginning of the queue.

> **Note** If you instantiate a **MessageQueue** with DenySharedReceive set to **true**, no other application can modify the messages in your enumerator while you have the connection to the queue.

To retrieve a static list of messages

1. Create an instance of the **MessageQueue** component and set its Path property to the queue to which you want to refer. For more information, see Creating MessageQueue Component Instances.

   > **Note** If you created your component from Server Explorer, the **Path** property is set automatically to the queue path for that queue.

2. Create an array that references the **Message** class to hold the results of your query.
3. Call the GetAllMessages method.
4. Assign the results to the array you created.

   The following code shows how you might use the **GetAllMessages** function to retrieve the messages from a queue and display the message labels in a list box. This example assumes that a **MessageQueue** component instance has been added to the application.

```
' Visual Basic
Private Sub button1_Click(ByVal sender As System.Object, _
   ByVal e As System.EventArgs) Handles button1.Click
   Me.MessageQueue1.Path = ".\MyQueue"
   Dim msg() As System.Messaging.Message
   Dim i As Integer
   ' Retrieve the messages.
   msg = MessageQueue1.GetAllMessages()
   ' Clear the current contents of the list.
   Me.ListBox1.Items.Clear()
   ' Display the results.
   For i = 0 To msg.Length - 1
      Me.ListBox1.Items.Add(msg(i).Label)
   Next
End Sub

// C#
private void button1_Click(System.Object sender, System.EventArgs e)
{
   messageQueue1.Path = ".\\MyQueue";
   System.Messaging.Message[] msg;
   // Retrieve the messages.
   msg = messageQueue1.GetAllMessages();
   // Clear the current contents of the list.
   this.listBox1.Items.Clear();
   // Display the results.
```

```
    for (int i = 0; i < msg.Length; i++)
    {
       this.listBox1.Items.Add(msg[i].Label);
    }
}
```

To retrieve a dynamic list of messages

1. Create an instance of the **MessageQueue** component and set its **Path** property to the queue to which you want to refer. For more information, see Creating MessageQueue Component Instances.

   Note If you created your component from Server Explorer, the **Path** property is set automatically to the queue path for that queue.

2. Create a MessageEnumerator object to hold the results of your query.
3. Call the GetEnumerator method on the **MessageQueue** class.
4. Assign the results to the **MessageEnumerator** object. Your code might look like this:

```
' Visual Basic
Dim mq As New System.Messaging.MessageQueue(".\MyQueue")
Dim msgEnum As System.Messaging.MessageEnumerator
msgEnum = CType(mq.GetEnumerator, System.Messaging.MessageEnumerator)

// C#
System.Messaging.MessageQueue mq = new System.Messaging.MessageQueue();
System.Messaging.MessageEnumerator msgEnum;
mq.Path = @".\MyQueue";
msgEnum = (System.Messaging.MessageEnumerator)(mq.GetEnumerator());
```

See Also

Queue and Message Collections | Retrieving Queues

Retrieving Queues

You can use either static or dynamic retrieval mechanisms to retrieve a list of queues. When you retrieve a static list of queues, the system returns a MessageQueue object with the results of your query. When you retrieve a dynamic list of queues, the system returns a MessageQueueEnumerator object with the results of the query.

Unlike the MessageEnumerator, which returns messages according to their order in a queue, the **MessageQueueEnumerator** does not return queues in any order. This is because queues on a network are not defined in order — they are not ordered, that is, by computer, label, or other user-accessible criteria. You can move the cursor to the first queue of the enumeration by calling the MoveNext method. After the enumerator has been initialized, you can use **MoveNext** to step forward through the remaining queues.

It is not possible to step backward with a **MessageQueueEnumerator**. A cursor only allows forward movement through the queue enumeration. However, you can call the Reset method to reset the enumeration and put the cursor at the beginning of the list again. Because the enumerator is dynamic, it can access a queue that is appended beyond the cursor's current position. A queue that is inserted before the cursor's current position cannot be accessed without first calling **Reset**.

Because the **GetPublicQueues**, **GetPrivateQueuesByMachine**, and **GetMessageQueuesEnumerator** methods are static, you do not need to create an instance of the **MessageQueue** class before calling the method.

To retrieve a static list of public or private queues

1. Create an array of type **MessageQueue** to hold the results of your query.
2. Call the appropriate method on the **MessageQueue** class:
   - To retrieve all public queues without criteria, call the GetPublicQueues method.
   - To retrieve public queues with criteria, set the **Criteria** parameter to the appropriate value and call the GetPrivateQueuesByMachine method.
   - To retrieve only those public queues that share a category GUID, call the GetPublicQueuesByCategory method and specify the category GUID as the parameter.
   - To retrieve only those public queues that share a label, call the GetPublicQueuesByLabel method and specify the label as a parameter.
   - To retrieve only those public queues on a specific machine, call the GetPublicQueuesByMachine method and specify the machine name as a parameter.
3. Assign the results to the array.

   For example, the following code shows how you might use a list box to display the label of all retrieved public queues on the local machine:

```
' Visual Basic
Private Sub button1_Click(ByVal sender As System.Object, _
   ByVal e As System.EventArgs) Handles button1.Click
   Dim mqlist() As System.Messaging.MessageQueue
   Dim i As Integer
   ' Retrieve queues on the local machine.
   mqlist = System.Messaging.MessageQueue.GetPublicQueuesByMachine(".")
   ' Clear the current contents of the list.
   Me.ListBox1.Items.Clear()
   ' Display the results.
   For i = 0 To mqlist.Length - 1
      Me.ListBox1.Items.Add(mqlist(i).Path)
   Next
End Sub

// C#
private void button1_Click(System.Object sender, System.EventArgs e)
{
   System.Messaging.MessageQueue[] mqlist;
   // Retrieve public queues.
   mqlist = System.Messaging.MessageQueue.GetPublicQueuesByMachine(
```

```
        ".");
    // Clear the current contents of the list.
    this.listBox1.Items.Clear();
    // Display the results.
    for (int i = 0; i < mqlist.Length; i++)
    {
        this.listBox1.Items.Add(mqlist[i].Path);
    }
}
```

To retrieve a dynamic list of queues

1. Create a MessageQueueEnumerator object to hold the results of your query.
2. Call the GetMessageQueueEnumerator method on the **MessageQueue** class.
3. To retrieve a subset of the queues on the network, set the Criteria parameter to the appropriate value.
4. Set the results to the **MessageQueueEnumerator** object you created. Your code might look like this:

```
' Visual Basic
Dim mqEnum As System.Messaging.MessageQueueEnumerator
mqEnum = System.Messaging.MessageQueue.GetMessageQueueEnumerator()

// C#
System.Messaging.MessageQueueEnumerator mqEnum;
mqEnum = System.Messaging.MessageQueue.GetMessageQueueEnumerator();
```

See Also

Queue and Message Collections | Retrieving Messages

Message Queue Security

Microsoft Windows Message Queuing takes advantage of the various built-in security features of the Windows 2000 operating system. Specifically, Message Queuing uses access control, authentication, encryption, and auditing for security:

- Access control is used to restrict user access to Message Queuing objects, and is implemented by assigning security descriptors to objects. Message Queuing objects include computer (MSMQ), queue, routing link, and Message Queuing Settings objects. A security descriptor lists the users and groups that are granted or denied access to an object, and the specific permissions assigned to those users and groups.
- Authentication is implemented using public key certificates, the Kerberos V5 security protocol, and Windows NTLM (for compatibility with Message Queuing 1.0 running on Windows NT 4.0). Public key certificates are used for message authentication, which verifies the sender of a message (the client) to a Message Queuing server. Kerberos V5 and NTLM are used for server authentication, which verifies a Message Queuing server to a client.
- Encryption is implemented using both public key (asymmetric) and secret key (symmetric) algorithms. Encryption is used by Message Queuing applications to encrypt messages sent between Message Queuing computers.
- Auditing is used to record which users attempt to access Message Queuing objects in Active Directory. The security descriptor for an object specifies the various access events to be audited for the object.

By managing security properties for objects, you can set permissions, assign ownership, and monitor user access. For more information on the specifics of security in Message Queuing, see "Security Message Queuing" in the Windows 2000 or Windows NT Message Queuing documentation, or search MSDN Online for articles on configuring security in your **MessageQueue** components.

See Also

Creating Messaging Components | Introduction to Messaging

Troubleshooting MessageQueue Components

You may encounter the following situation while working with the **MessageQueue** component:

Client is Not Recognized by Microsoft MessageQueue Service

If you are sending messages from a server configured as a Windows 2000 dependent client to a Message Queue enterprise and the enterprise server is a Windows NT version 4.0 server with Service Pack 4, you might receive the following error when you send a message to the queue:

> "Service not installed."

To solve this problem

- Install Service Pack 6 or later on the Windows NT 4.0 server. For more information, see the Downloads section of the Microsoft Web site (http://www.microsoft.com/).

See Also

Introduction to Messaging | Troubleshooting Problems Portal

Creating File System and Timer Components

File system and scheduling components allow you to interact with file system and directory-level resources and raise events at scheduled intervals. **FileSystemWatcher** component instances allow you to watch for changes to directories and files and react when they occur. The Timer component allows you to set up simple event-raising schedules and carry out associated processing.

In This Section

Reacting to File System Events
Provides information about responding to events on a file system.

Creating Timer-Based Server Tasks
Describes how to use server-based **Timer** components to raise events on a simple, periodic recurrence pattern.

Related Sections

Programming with Components
Presents a high-level overview of component programming and authoring.

Introduction to the Windows Forms Timer Component
Describes the Windows-based timer and how you can use it.

Reacting to File System Events

Using the **FileSystemWatcher** component and language features in Visual Studio .NET or the Microsoft .NET Framework SDK, you can easily monitor actions occurring within directories and files on your system or on any computer to which you have network access.

In This Section

Introduction to Monitoring File System Events
Provides basic information about file- and directory-watching features in Visual Studio .NET and the Microsoft .NET Framework SDK.
Creating FileSystemWatcher Component Instances
Gives directions for creating instances of **FileSystemWatcher** components in your projects.
Configuring FileSystemWatcher Component Instances
Gives directions for setting properties to determine the behavior of **FileSystemWatcher** components.
Creating Handlers for File System Events
Provides recommendations for creating event handlers that will automatically take action when file system events are raised.
Considerations for File Changes on High-Volume Systems
Provides information about controlling the flow of file system events on systems with high activity.
Waiting Until Specified File System Events Occur
Gives directions for using the **WaitForChanged** method to monitor directories.
Walkthrough: Reacting to File System Events
Provides step-by-step instructions for how to respond to events with the **FileSystemWatcher** component.
Troubleshooting FileSystemWatcher Components
Explains how to remedy some common problems.

Related Sections

Programming with Components
Presents a high-level overview of component programming and authoring.
FileSystemWatcher Class
Describes the major features of the FileSystemWatcher class.
Visual Basic and Visual C# Projects
Describes the projects types used in this chapter and how to choose between them.
Deploying Applications and Components
Lists the major deployment topics and the information they contain.

Introduction to Monitoring File System Events

You use the **FileSystemWatcher** component to monitor a file system and react when changes to the directories or files it contains occur. This makes it possible for you to quickly and easily launch business processes when certain files or directories are created, modified, or deleted. For example, suppose you and a group of coworkers are collaborating on a document that is stored on a shared directory on your server. Using an instance of the **FileSystemWatcher** component, you can program your application to watch for changes to the contents of that shared directory. When a change is detected, the component can run procedures that notify each of the group members via e-mail.

> **Note** The **FileSystemWatcher** component is designed to watch for changes within a directory, not to changes to the root directory's attributes themselves. For example, if you are watching a directory called c:\MyProjects, the component will monitor changes within the directory but not changes to the root directory itself.

You can configure the component to watch either an entire directory and its contents or a specific file or set of files within a given directory. The **FileSystemWatcher** component raises an event whenever a file or subdirectory within the specified root directory is created, deleted, renamed, or changed in some other way. The types of changes that the component monitors include changes in the file's or subdirectory's attributes, size, last write time, last access time, and security settings.

> **Security Note** A denial-of-service attack is possible if a malicious program gains access to a directory your **FileSystemWatcher** component is monitoring and generates so many changes that the component cannot manage them. Follow the recommendations in Considerations for File Changes on High-Volume Systems and use access control lists properly to reduce the risk of a denial-of-service attack. For more information, see Access Control.

Directory-Level Events

When changes occur within a root directory or to a file your application is monitoring, one or more events are raised, stored in a buffer, and ultimately passed to your component for processing. The **FileSystemWatcher** component gives you access to the following events:

- Created — raised whenever a directory or file is created.
- Deleted — raised whenever a directory or file is deleted.
- Renamed — raised whenever the name of a directory or file is changed.
- Changed — raised whenever changes are made to the size, system attributes, last write time, last access time, or security permissions of a directory or file.

> **Note** Although some common occurrences, such as copying or moving a file, do not correspond directly to an event, these occurrences do cause events to be raised. When you copy a file, the system raises a **Created** event in the directory to which the file was copied but does not raise any events in the original directory. When you move a file, the server raises two events: a **Deleted** event in the source directory, followed by a **Created** event in the target directory.

Each change to a file in a directory raises a separate event. If a user makes multiple changes to a single file, the application raises a separate event for each change. For example, if you are watching a directory that contains five files and the security permissions for all of the files suddenly change, you will receive five **Changed** events. If the user renames and assigns new security permissions to each of those files, you will receive 10 event notifications — a **Changed** event and a **Renamed** event for each of the five files.

See Also

Creating FileSystemWatcher Component Instances | Configuring FileSystemWatcher Component Instances | Creating Handlers for File System Events | Considerations for File Changes on High-Volume Systems | Waiting Until Specified File System Events Occur

Creating FileSystemWatcher Component Instances

You create an instance of the **FileSystemWatcher** component when you want to build file system monitoring into your application. **FileSystemWatcher** components allow you to connect to directories and watch for specific changes within them, such as the creation of new files, addition of subdirectories, and renaming of files or subdirectories.

You can add an instance of the **FileSystemWatcher** component to Windows Forms, Web Forms, and component classes. **FileSystemWatcher** components have no visual user interface. If you add an instance of the **FileSystemWatcher** component to a visual designer (such as the Windows Forms Designer), the component appears in a small area below the bottom border of the form. This area is known as the component tray and acts as a place to display all of the nonvisual items associated with the form.

> **Note** Nonvisual classes that inherit from **System.ComponentModel.Component** support a visual design surface similar to the component tray, on which your **FileSystemWatcher** component and other components are arranged. The arrangement of items on this designer is not important, as the interface you see will never be displayed to the application's end users.

There are several ways you can create an instance of the **FileSystemWatcher** component:

- You can drag an instance of the **FileSystemWatcher** component from the **Components** tab of the Toolbox to a form or other designer.
- You can create an instance of the **FileSystemWatcher** component in code.

> **Note** Unlike many of the other components on the Components tab of the Toolbox, **FileSystemWatcher** components are not represented in Server Explorer, so you cannot create instances of the component from that window.

To create an instance of the FileSystemWatcher component from the Toolbox

1. Access the **Components** tab of the Toolbox.
2. Select the **FileSystemWatcher** icon and drag it to the designer surface for your form or component.
3. Configure your component. For more information, see Configuring FileSystemWatcher Component Instances.

To create an instance of the FileSystemWatcher component programmatically

1. Create an instance of the **FileSystemWatcher** base class in code. Your code might look like this:

```
' Visual Basic
Dim myWatcher As New System.IO.FileSystemWatcher()

// C#
System.IO.FileSystemWatcher myWatcher =
   new System.IO.FileSystemWatcher();
```

2. Configure any other necessary properties for your component. For more information, see Configuring FileSystemWatcher Component Instances.

See Also

Configuring FileSystemWatcher Component Instances | Introduction to Monitoring File System Events

Configuring FileSystemWatcher Component Instances

There are several properties you set for your **FileSystemWatcher** component instances to determine how they behave. These properties determine what directories and subdirectories the component instance will monitor and the exact occurrences within those directories that will raise events.

Specifying the Directories to Watch

You use two properties to determine what directories the **FileSystemWatcher** component should watch: Path and IncludeSubdirectories.

Path indicates the fully qualified path of the root directory you want to watch. This can be in standard directory path notation (c:\directory) or in UNC format (\\server\directory). You can set the **Path** property on an existing component instance, as shown in the following code:

```
' Visual Basic
Dim MyWatcher As New System.IO.FileSystemWatcher()
MyWatcher.Path = "c:\"

// C#
System.IO.FileSystemWatcher MyWatcher = new System.IO.FileSystemWatcher();
MyWatcher.Path = "c:\\";
```

The **IncludeSubdirectories** property indicates whether subdirectories within the root directory should be monitored. If the property is set to **true**, the component watches for the same changes in the subdirectories as it does within the main directory the component is watching.

Specifying the Changes to Watch

In addition to specifying the directories and subdirectories to watch, you can indicate what specific files and types of changes you are interested in watching for. In part, you determine the changes to watch for by defining handlers for only those events in which you are interested. For example, if you only want to be notified when files are created, you would only handle the Created event. However, you can set a series of properties that restrict the component's activities even further.

You can use the Filter property to indicate a wildcard matching pattern that restricts the component to watching only certain files or directories within the root directory. For example, if you want to watch for the creation of text files within the root, you could set the Filter property to "*.txt" and then create a handler for the **Created** event. You can set the filter to any value.

> **Note** The **FileSystemWatcher** component is designed to watch for changes within a directory, not to changes to the directory's attributes themselves. For example, if you are watching a directory called c:\MyProjects, the component will monitor changes within the directory but not changes to the directory itself.

You can also use the NotifyFilter property to further restrict the changes that are reacted to. You can set the **NotifyFilter** property to watch for changes to directory names, files names, or both. In addition, the **NotifyFilter** property contains a series of values that restrict the **Changed** event. The **Changed** event has the potential to raise a large number of events, because it will raise an event each time any change is made to the file in question; some standard operations like copying and moving files will actually cause several file change events to be raised as various attributes such as size, last write, and last access are altered.

The **NotifyFilter** property limits the number of file or directory changes for which the component instance will raise events. You can set up your component so that it will raise events only when a file or directory name changes, when the attributes of an item in the main directory change, when a file's size changes, when its last write or last access time changes, or when security access to a file or directory rights change. These values are all part of the **NotifyFilters** enumeration. You can set multiple changes to watch for by using a **BitOr** (for Visual Basic) or **|** (for Visual C#) operator, as shown in the example below.

> **Note** You must set the EnableRaisingEvents property to **true** in order for the **FileSystemWatcher** component to start monitoring for changes.

To configure an instance of the FileSystemWatcher component

1. Create an instance of the **FileSystemWatcher** component. For more information, see Creating FileSystemWatcher Component Instances.
2. Set the **Path** property to the fully qualified path of the root directory you want to watch.

   > **Tip** You can enter this as a standard file path or a UNC path.

3. Set any of the optional properties for your instance.

Scenario	Property	Value
To watch for changes to a specific file or subdirectory name within the root	**Filter**	A wildcard filter expression that restricts the monitoring activities of the instance
To watch for changes within any subdirectories the root directory contains	**IncludeSubdirectories**	**true** or **false**
To watch for only specific changes to a file or subdirectory when handling the **Changed, Renamed, Deleted,** or **Created** events	**NotifyFilter**	Any of the available values in the NotifyFilters enumeration

4. Enable your component instance by setting the EnableRaisingEvents property to **true**.

For example, suppose you want to create a **FileSystemWatcher** instance that watches a drop directory called "reports" for the creation of new .txt files. You also want to monitor changes to existing reports that rerun dynamically when data changes. You would set up your component as follows:

```
' Visual Basic
' This needs to be declared in a place where it will not go out of scope.
' For example, it would be a class variable in a form class.
Dim MyWatcher As New System.IO.FileSystemWatcher()
' This code would go in one of the initialization methods of the class.
MyWatcher.Path = "c:\"
' Watch only for changes to *.txt files.
MyWatcher.Filter = "*.txt"
MyWatcher.IncludeSubdirectories = False
' Filter for Last Write changes.
MyWatcher.NotifyFilter = System.IO.NotifyFilters.LastWrite
' Example of watching more than one type of change.
MyWatcher.NotifyFilter = _
   System.IO.NotifyFilters.LastAccess Or System.IO.NotifyFilters.Size
' Enable the component to begin watching for changes.
MyWatcher.EnableRaisingEvents = True

// C#
// This needs to be declared in a place where it will not go out of scope.
// For example, it would be a class variable in a form class.
System.IO.FileSystemWatcher MyWatcher = new System.IO.FileSystemWatcher();
// This code would go in one of the initialization methods of the class.
MyWatcher.Path = "c:\\";
// Watch only for changes to *.txt files.
MyWatcher.Filter = "*.txt";
MyWatcher.IncludeSubdirectories = false;
// Enable the component to begin watching for changes.
MyWatcher.EnableRaisingEvents = true;
// Filter for Last Write changes.
MyWatcher.NotifyFilter = System.IO.NotifyFilters.LastWrite;
// Example of watching more than one type of change.
MyWatcher.NotifyFilter =
   System.IO.NotifyFilters.LastWrite | System.IO.NotifyFilters.Size;
```

See Also

Creating FileSystemWatcher Component Instances | Introduction to Monitoring File System Events

Creating Handlers for File System Events

The **FileSystemWatcher** component raises four events depending on the types of changes that occur in the directory it is watching. These events are:

- Created — raised whenever a directory or file is created.
- Deleted — raised whenever a directory or file is deleted.
- Renamed — raised whenever the name of a directory or file is changed.
- Changed — raised whenever changes are made to the size, system attributes, last write time, last access time, or NTFS security permissions of a directory or file.

> **Note** You can use the NotifyFilter property to limit the amount of events the **Changed** event raises. For more information, see Configuring FileSystemWatcher Component Instances.

For each of these events, you can define handlers that automatically call methods in your code when a change occurs. An event handler is a method that is bound to an event for a component, form, or control. Each event handler provides two parameters that allow you to handle the event properly — the *sender*, which provides an object reference to the object responsible for the event, and the *e* parameter, which provides an object for representing the event and its information.

There are several ways you can create an event handler for an instance of the **FileSystemWatcher** component, but the simplest way is to let Visual Studio .NET create most of the code for you automatically. When you double-click your **FileSystemWatcher** component instance in the designer, Visual Studio opens the Code Editor and creates an event handler for both the **Changed** event and the blank procedure it calls. You can then fill in the processing that you want to occur in the procedure. You can also create your own handlers to handle the other events.

For more information on event handlers, see Event Handlers in Visual Basic and Visual C#.

To create a default handler for the Changed event

1. Create an instance of the **FileSystemWatcher** component. For more information, see Creating FileSystemWatcher Component Instances.
2. Set the necessary properties for the component. For more information, see Configuring FileSystemWatcher Component Instances.
3. From the designer, double-click the **FileSystemWatcher** component for which you want to create a handler.

   > **Note** The Code Editor appears and two items are added to your code: the handler that creates and registers a delegate and calls the procedure, and a blank procedure for the **Changed** event.

4. In the blank procedure for the **Changed** event, define code to take action when this event is called.

To create a handler for a FileSystemWatcher event

1. Create an instance of the **FileSystemWatcher** component. For more information, see Creating FileSystemWatcher Component Instances.
2. Set the necessary properties for the component. For more information, see Configuring FileSystemWatcher Component Instances.
3. Create the appropriate procedure that will be called and define the code you want to process the entries. The following example shows an event handler for the **Changed** event.

```
' Visual Basic
Private Sub myWatcher_Changed(ByVal sender As System.Object, _
   ByVal e As System.IO.FileSystemEventArgs)
   Dim pathChanged As String
   pathChanged = e.FullPath
End Sub

// C#
private void myWatcher_Changed(object sender,
System.IO.FileSystemEventArgs e)
{
   string pathChanged = e.FullPath;
}
```

4. Assign an instance of FileSystemEventHandler to the event for your component. This example adds the **Changed** event handler defined in step 3.

```
' Visual Basic
AddHandler myWatcher.Changed, _
   New System.IO.FileSystemEventHandler(AddressOf Me.myWatcher_Changed)

// C#
myWatcher.Changed +=
   new System.IO.FileSystemEventHandler(this.myWatcher_Changed);
```

See Also

Creating FileSystemWatcher Component Instances | Configuring FileSystemWatcher Component Instances | Introduction to Monitoring File System Events

Considerations for File Changes on High-Volume Systems

Your **FileSystemWatcher** component has the potential to receive an enormous number of events, particularly if you have set it to watch a very high-traffic directory. This can cause problems, as the **FileSystemWatcher** component can only receive event notifications at a manageable rate. If the rate at which events are coming in exceeds the rate at which the component can receive them, the excess events are stored in a buffer that is then passed to the component. The component translates each entry in the buffer to an event to be raised to the user's code. This buffer can cause a problem in that it can overflow. If the number of changes in a directory is so large that they cannot all fit in the buffer, the operating system will issue a "blanket" notification, and the component will raise an exception.

> **Security Note** A denial-of-service attack is possible if a malicious program gains access to a directory your **FileSystemWatcher** component is monitoring and generates so many changes that the component cannot manage them. Follow the recommendations in this Help topic and use access control lists properly to reduce the risk of a denial-of-service attack. For more information, see Access Control.

To avoid this problem, you should do several things. First, you should set your buffer to an appropriate size for the approximate number of events you expect to receive. By default, the buffer is set to a size of 4 KB. A 4 KB buffer can track changes on approximately 80 files in a directory. Each event takes up 16 bytes in the buffer, plus enough bytes to store the name of the file, in Unicode (2 bytes per character), that the event occurred on. You can use this information to approximate the buffer size you will need.

You reset the buffer size by setting the InternalBufferSize property in the Properties window. If you are using Microsoft Windows 2000, you should increase the buffer size in increments of 4 KB, because this corresponds to the operating system's default page size. With any other operating system, you should increase the buffer size in increments that correspond to the operating system's default page size.

> **Tip** If you are unsure of the default page size for the operating system you are using, the safest way to proceed is to just double the original size of the buffer. This will maintain the original interval needed for your operating system.

In addition to setting an appropriate buffer size, you can use the following three properties to control the amount of information that the buffer receives:

- Use the NotifyFilter property to determine whether the **FileSystemWatcher** component looks for all changes or only specified changes in the watched directory.
- Use the IncludeSubdirectories property to indicate whether or not you want to include the subdirectories of the directory you are watching. If you turn this off when you do not need it, you will receive fewer events than when it is turned on.
- If you have set the **FileSystemWatcher** component to watch for the **Changed** event, you can use the **NotifyFilter** property to specify only those events in which you are interested.

  > **Note** Although you can also use the Filter property to specify files or subdirectories that you want watched, this property is applied after changes are added to your buffer and will not reduce the buffer size at all. Instead, use the **NotifyFilter** property to control the amount of information written to the buffer.

See Also

Configuring FileSystemWatcher Component Instances | Introduction to Monitoring File System Events

Waiting Until Specified File System Events Occur

In addition to using the **FileSystemWatcher** component to immediately monitor a specific directory, you can use the WaitForChanged method to wait until a specific event occurs and then continue with execution of the thread. For example, if you are working with a Web-based news application, you might create an admin portion of the site where users upload their news stories. You could use the **WaitForChanged** method to watch that directory until the last access date changes, and then begin processing the news directory for new articles.

You specify the type of change to watch for by setting the value of a WatcherChangeType enumeration. The possible values are **Created**, **Changed**, **Renamed**, **Deleted**, or **All**. Setting the value to **All** watches for any change to the directory. You can also specify a timeout period if you only want to wait a certain period of time for the event to occur.

The **WaitForChanged** method returns an object of type WaitForChangedResult. This class contains specific information on the type of change that occurred in the directory. You can access information such as Name, OldName, and TimedOut on this object to find out more about the change.

WaitForChanged is a synchronous method. If you use it in a Windows Form application, the application would stop responding if the method was used on the UI thread instead of the worker thread. For more information on threading issues in Visual Basic .NET, see Multithreading in Visual Basic .NET.

To wait for changes

1. Create and configure your **FileSystemWatcher** component instance to indicate the directory you want to watch. For more information, see Creating FileSystemWatcher Component Instances or Configuring FileSystemWatcher Component Instances.
2. In the Code Editor, create an instance of the **WaitForChangedResult** class.

```
' Visual Basic
Dim result As System.IO.WaitForChangedResult

// C#
System.IO.WaitForChangedResult result;
```

3. On your **FileSystemWatcher** component instance, specify the type of event it is waiting for by setting the value of the **WaitForChanged** property to one of the values of the WatcherChangeTypes enumeration.

```
' Visual Basic
fileSystemWatcher1.WaitForChanged(System.IO.WatcherChangeTypes.All)

// C#
fileSystemWatcher1.WaitForChanged(System.IO.WatcherChangeTypes.All);
```

   Tip You can use an overloaded form of the WaitForChanged method to specify a timeout period as a second parameter.

4. Specify the actions you want to take when the change for which you are watching occurs. For more information, see Creating Handlers for File System Events.

The following code shows how to notify a user that a file needs to be deleted, then wait for the **Deleted** event to be raised. This code watches the personal folder of the user.

```
' Visual Basic
Console.WriteLine("Please delete OldNote.txt.")
Dim directory As String
directory = Environment.GetFolderPath(Environment.SpecialFolder.Personal)
Dim watcher As New System.IO.FileSystemWatcher(directory, "OldNote.txt")
result = watcher.WaitForChanged(System.IO.WatcherChangeTypes.Deleted)
Console.WriteLine("Thanks for deleting OldNote.txt.")

// C#
Console.WriteLine("Please delete OldNote.txt.");
string directory =
   Environment.GetFolderPath(Environment.SpecialFolder.Personal);
```

```
System.IO.FileSystemWatcher watcher =
   new System.IO.FileSystemWatcher(directory,"OldNote.txt");
result = watcher.WaitForChanged(System.IO.WatcherChangeTypes.Deleted);
Console.WriteLine("Thanks for deleting OldNote.txt.");
```

See Also

Creating FileSystemWatcher Component Instances | Configuring FileSystemWatcher Component Instances | Introduction to Monitoring File System Events | Creating Handlers for File System Events

Walkthrough: Reacting to File System Events

The procedures in this topic walk you through the process of creating a **FileSystemWatcher** component, pointing it to a directory on your local computer, and then using the FileSystemWatcher.Filter property to watch only for changes to text files. You will create an event handler that responds when the FileSystemWatcher.Created event and FileSystemWatcher.Changed event are raised, and you will use a form to display the notifications that result from these events.

Create the form and components you need for the application

In this procedure, you create a Windows Form and a **FileSystemWatcher** component instance that will work together to react to directory-level events.

1. From the **New Project** dialog box, create a Visual Basic or C# Windows Application, and name it **MyWatcher**.
2. From the **Windows Forms** tab in the Toolbox, drag two **Label** controls to the designer surface.
3. Click the **Components** tab in the Toolbox, and then drag the **FileSystemWatcher** icon to the designer surface for your component. By default, this component is named **FileSystemWatcher1**.

To set properties for your FileSystemWatcher component

In this procedure, you set several properties for your component to determine what it will watch. These settings cause the component to watch the specified directory on your local computer and to watch for the creation of files ending in the .txt file name extension.

1. Click the **FileSystemWatcher** component you created in the previous procedure, and view its properties in the Properties window.
2. Set the component's name to **myWatcher**.
3. Use the Path property to set the **FileSystemWatcher** component to watch a directory on your local computer. For example, on a computer running Microsoft® Windows® 2000, you might enter the following into the **Path** property to set the component to watch your My Documents directory:

   ```
   C:\Documents and Settings\yourusername\My Documents\
   ```

   Tip For the purposes of this example, you can use any directory you want on your local computer.

4. Set the **Filter** property to ***.txt** to have the component watch only those files ending in the .txt file name extension.

Write the necessary code for your component

In this procedure, you will define two event handlers for your component that define the processing that should occur whenever the **Changed** and **Created** events are raised.

1. Double-click the **FileSystemWatcher** component. The Code Editor appears and a default event handler for the **Changed** event appears.

   Note For more information, see Event Handling in Windows Forms.

2. Use the following code to display a simple text string that will verify that your event was raised:

   ```
   ' Visual Basic
   Private Sub myWatcher_Changed(ByVal sender As System.Object, _
      ByVal e As System.IO.FileSystemEventArgs) Handles myWatcher.Changed
      Me.Label1.Text = "Changes made to: " & e.FullPath
   End Sub

   // C#
   private void myWatcher_Changed(object sender,
      System.IO.FileSystemEventArgs e)
   {
      label1.Text = "Changes made to: " + e.FullPath;
   }
   ```

3. Create the event handler for the **Created** event that specifies what your application should do whenever the component

raises this event. For information on creating event handlers, see Creating Event Handlers on the Windows Forms Designer. Add code as shown to display the full path of the newly created file.

```
' Visual Basic
Private Sub myWatcher_Created(ByVal sender As System.Object, _
   ByVal e As System.IO.FileSystemEventArgs) Handles myWatcher.Created
   Me.Label2.Text = "The file: " & e.FullPath & _
      " has been added to your directory"
End Sub

//C#
private void myWatcher_Created(object sender,
   System.IO.FileSystemEventArgs e)
{
   label2.Text = "The file: " + e.FullPath +
      " has been added to your directory.";
}
```

4. Save all files, and then build and run your application.

To test your FileSystemWatcher component

In this procedure, you will manually make changes in the directory your component is watching so that you can force the event handler to raise the **Changed** and **Created** events.

1. Run the application you created in the previous procedure.
2. Use Windows Explorer to find the directory that you set your **FileSystemWatcher** component to watch.
3. Open Notepad, and then create a new text file. Save this file to the directory you located in step 2 and close the file.

   Note This should raise the **Created** and **Changed** events and run the handler you defined.

4. Return to the form. You should see the created message in the label.
5. Open the text file you created, type a few lines of text, and save it again.

   Note This should raise the **Changed** event and run the handler you defined.

6. Return to the form. You should see the change message in the label.

See Also

Creating FileSystemWatcher Component Instances | Introduction to Monitoring File System Events | Label Control (Windows Forms)

Troubleshooting FileSystemWatcher Components

You may encounter the following situations while working with the **FileSystemWatcher** component:

UNC Path Names Not Accepted on Windows NT 4.0 Computers

If you are working with a **FileSystemWatcher** component on a Windows NT version 4.0 computer and trying to set its path to monitor file system activity on a different Windows NT version 4.0 computer, you will not be able to specify a UNC-based path value in the Path property to point to the computer in question. You can only set UNC-based values when working on Windows 2000 computers.

Cannot Watch Windows 95 or Windows 98 Directories

If you set your **FileSystemWatcher** component to reference a directory on a Windows 95 or Windows 98 computer, you will receive an error about an invalid directory path when the project runs. When using **FileSystemWatcher**, you cannot watch directories on computers running Windows 95 or Windows 98.

Multiple Created Events Generated for a Single Action

You may notice in certain situations that a single creation event generates multiple Created events that are handled by your component. For example, if you use a **FileSystemWatcher** component to monitor the creation of new files in a directory, and then test it by using Notepad to create a file, you may see two **Created** events generated even though only a single file was created. This is because Notepad performs multiple file system actions during the writing process. Notepad writes to the disk in batches that create the content of the file and then the file attributes. Other applications may perform in the same manner. Because **FileSystemWatcher** monitors the operating system activities, all events that these applications fire will be picked up.

> **Note** Notepad may also cause other interesting event generations. For example, if you use the **ChangeEventFilter** to specify that you want to watch only for attribute changes, and then you write to a file in the directory you are watching using Notepad, you will raise an event . This is because Notepad updates the **Archived** attribute for the file during this operation.

Unexpected Events Generated on Directories

Changing a file within a directory you are monitoring with a **FileSystemWatcher** component generates not only a Changed event on the file but also a similar event for the directory itself. This is because the directory maintains several types of information for each file it contains — the names and sizes of files, their modification dates, attributes, and so on. Whenever one of these attributes changes, a change is associated with the directory as well.

See Also

Creating FileSystemWatcher Component Instances | Introduction to Monitoring File System Events | Troubleshooting Problems Portal

Creating Timer-Based Server Tasks

Using the **Timer** component and language features in Visual Studio .NET or the Microsoft .NET Framework, you can easily create events that are raised on recurring intervals.

In This Section

Introduction to Server-Based Timers
Explains the server-based **Timer** component and how you can use it to raise events at simple intervals.
Timer Programming Architecture
Gives an overview of the basic language elements of the server-based **Timer** component.
Creating Instances of Server-Based Timers
Describes how to create **Timer** components to build interval-based events into your multi-threaded application.
Monitoring Server-Based Timers
Explains how to create a handler for the **Elapsed** event.

Related Sections

System.Timers Namespace
Describes the major features of the **Timers** class.
Introduction to the Windows Forms Timer Component
Describes the Windows-based timer and how you can use it.
Programming with Components
Presents a high-level overview of component programming and authoring.

Introduction to Server-Based Timers

Server-based timers allow you to specify a recurring interval at which an event will be raised in your application. You can then handle this event to provide regular processing. For example, suppose you have a critical server that must be kept running 24 hours a day, seven days a week. You could create a service that uses a timer to periodically check the critical server to ensure that the system is up and running. If the system is not responding, the service can attempt to restart the server or notify an administrator.

> **Note** The interval for server timers is specified in milliseconds.

Server Timers, Windows Timers, and Thread Timers

There are three timer controls in Visual Studio .NET and the .NET Framework — the server-based timer you can see on the Components tab of the Toolbox, the standard Windows-based timer you can see on the Windows Forms tab of the Toolbox, and the thread timer that is only available programmatically. The Windows-based timer has existed in Visual Basic since version 1.0, and has remained essentially unchanged. This timer is optimized for use in Windows Forms applications. The server-based timer is an update of the traditional timer that has been optimized to run in a server environment. The thread timer is a simple, lightweight timer that uses callback methods instead of events and is served by threadpool threads.

There are two types of threads in Win32 architecture: UI threads, and worker threads. UI threads sit idle most of the time and wait for messages to arrive in their message loops. Once they receive a message, they handle it and then wait for the next message to arrive. Alternatively, worker threads are used to perform background processing and do not use message loops. Both the Windows timer and the server-based timer run using an Interval property. The interval of the thread timer is set in the **Timer** constructor. The timers are designed for different purposes, as evidenced by their handling of threads:

- The Windows timer is designed for a single-threaded environment where UI threads are used to perform processing. The accuracy of Windows timers is limited to 55 milliseconds. These traditional timers require that the user code have a UI message pump available and always operate from the same thread, or marshal the call onto another thread. For a COM component, this would be detrimental to performance.
- The server-based timer is designed for use with worker threads in a multi-threaded environment. Because they use a different architecture, server-based timers have the potential to be much more accurate than Windows timers. Server timers can move among threads to handle the raised events.
- The thread timer is useful in scenarios where messages are not pumped on the thread. For example, the Windows-based timer relies on operating-system timer support, and if you are not pumping messages on the thread, the event associated with your timer will not occur. The thread timer is more useful in this case.

The Windows timer is located in the **System.Windows.Forms** namespace, the server timer is located in the **System.Timers** namespace, and the thread timer is located in the **System.Threading** namespace.

See Also

Timer Programming Architecture | Creating Instances of Server-based Timers | Monitoring Server-Based Timers | Introduction to the Windows Forms Timer Component | Timer (thread timer) | Thread Pooling

Timer Programming Architecture

The **Timer** component uses the System.Timers namespace in the Microsoft .NET Framework. The major class within this namespace pertaining to the **Timer** component is System.Timers.Timer.

Key Programming Elements of Server-Based Timers

The **Timer** component raises an event called Timer.Elapsed. You can create handlers for this event to perform whatever processing needs to occur.

Some of the more important properties and methods of a **Timer** component include the following:

- The Interval property is used to set the time span, in milliseconds, at which events should be raised. For example, an interval of 1000 will raise an event once a second.
- The AutoReset property determines whether the timer continues raising events after the given interval elapses. If set to **true**, the timer continues to recount the interval and raise events. If **false**, it raises one event after the interval elapses and then stops.
- The Start method sets the timer's **Enabled** property to **true**, which allows the timer to begin raising events. If the timer is already enabled, calling the **Start** method resets the timer.
- The Stop method sets the timer's Enabled property to **false**, preventing the timer from raising any more events.

See Also

Introduction to Server-Based Timers | Creating Instances of Server-based Timers | Monitoring Server-Based Timers

Creating Instances of Server-Based Timers

You create **Timer** components when you want to build interval-based events into your multi-threaded application. You can add an instance of the **Timer** component to Web Forms, Windows Services, and component classes. If you add an instance of the **Timer** component to a visual designer (such as the Web Forms Designer), the component appears in a small area below the bottom border of the form. This is known as the component tray and acts as a place to display all of the non-visual items associated with the form.

There are two ways you can create an instance of the **Timer** component:

- You can drag an instance of the **Timer** component from the Components tab of the Toolbox to a form or other designer.
- You can create an instance of the **Timer** component in code.

To create a Timer component from the Toolbox

1. Access the **Components** tab of the Toolbox.
2. Select the **Timer** icon, and drag it to the designer surface for your form or component.
3. Set the Interval property for your component instance to the length of time, in milliseconds, after which you want the component to raise an event.
4. Set the Enabled property for your component instance to **true**.

To create a Timer component programmatically

1. Add a reference to the **System.Timers** DLL and import the **System.Timers** namespace into your project.
2. Create an instance of the **Timer** class in code.
3. Set the Interval property for your component instance to the length of time, in milliseconds, after which you want the component to raise an event.
4. Set the Enabled property for your component instance to **true**.

   The following code shows how to create and configure a **Timer** component programmatically.

```
' Visual Basic
Dim myTimer As New System.Timers.Timer()
myTimer.Enabled = True
myTimer.Interval = 3000

// C#
System.Timers.Timer myTimer = new System.Timers.Timer();
myTimer.Enabled = true;
myTimer.Interval = 3000;
```

See Also

Introduction to Server-Based Timers | Monitoring Server-Based Timers | Introduction to the Windows Forms Timer Component

Monitoring Server-Based Timers

The **Timer** component raises an event called Elapsed periodically, based on the interval you specify. You can handle this event to perform the processing you need. For example, suppose that you have an online sales application that continuously posts sales orders to a database. The service that compiles the instructions for shipping operates on a batch of orders rather than processing each order individually. You could use a timer to start the batch processing every 30 minutes.

To create a default handler for the Elapsed event

1. From the designer, double click on the **Timer** component for which you want to create a handler.

   Note The Code Editor appears and two items are added to your code: the handler that creates and registers a delegate and calls the procedure, and a blank procedure for the **Timer.Elapsed** event.

2. In the blank procedure for the **Elapsed** event, define code to run when this event is called.

To create a handler for the Elapsed event programmatically

1. Create an instance of the **Timer** component and set the necessary properties for it to function. For more information, see Creating Instances of Server-Based Timers.
2. Use the AddHandler method to create an event handler for your component that will call the **MyTimer_Elapsed** procedure when the Elapsed event is raised. Your code should look like this:

```
' Visual Basic
AddHandler myTimer.Elapsed, New _
   System.Timers.ElapsedEventHandler(AddressOf Me.myTimer_Elapsed)

// C#
myTimer.Elapsed +=
   new System.Timers.ElapsedEventHandler(myTimer_Elapsed);
```

   Note For more information on this syntax, see Event Handling in Windows Forms.

3. Create the **myTimer_Elapsed** procedure and define the code you want to process the event.

```
' Visual Basic
Private Sub myTimer_Elapsed(ByVal sender As System.Object, ByVal e As System.Timers.Elaps
edEventArgs)
   System.Windows.Forms.MessageBox.Show("Elapsed!", _
      "Timer Event Raised!")
End Sub

// C#
private void myTimer_Elapsed(object sender, System.Timers.ElapsedEventArgs e)
{
   System.Windows.Forms.MessageBox.Show("Elapsed!",
      "Timer Event Raised!");
}
```

The following code shows how you can create a **Timer** component that raises an event every five seconds and displays a message box that alerts you to the event:

```
' Visual Basic
Private Sub CreateTimer()
   Dim Timer1 As New System.Timers.Timer()
   Timer1.Interval = 5000
   Timer1.Enabled = True
   AddHandler Timer1.Elapsed, _
      New System.Timers.ElapsedEventHandler(AddressOf Me.Timer1_Elapsed)
End Sub
```

```
Private Sub Timer1_Elapsed(ByVal sender As System.Object, _
   ByVal e As System.Timers.ElapsedEventArgs)
   System.Windows.Forms.MessageBox.Show("Elapsed!", "Timer Event Raised!")
End Sub

// C#
private void CreateTimer()
{
   System.Timers.Timer Timer1 = new System.Timers.Timer();
   Timer1.Enabled = true;
   Timer1.Interval = 5000;
   Timer1.Elapsed +=
      new System.Timers.ElapsedEventHandler(Timer1_Elapsed);
}

private void Timer1_Elapsed(object sender,
System.Timers.ElapsedEventArgs e)
{
   System.Windows.Forms.MessageBox.Show("Elapsed!",
      "Timer Event Raised!");
}
```

See Also

Introduction to Server-Based Timers | Creating Instances of Server-based Timers | Event Handling in Windows Forms | Introduction to the Windows Forms Timer Component

Creating System Monitoring Components

System monitoring components such as the **PerformanceCounter** component, the **EventLog** component, and the **ServiceController** component allow you to monitor and interact with a variety of system resources. You can use these component instances to interact with Windows performance counters, event logs, and long-running services.

In This Section

System Monitoring Walkthroughs
Lists walkthroughs you can use to learn how to work with **EventLog**, **PerformanceCounter**, and **ServiceController** components.

Monitoring Performance Thresholds
Provides basic information on monitoring Microsoft Windows performance thresholds.

Logging Application, Server, and Security Events
Provides additional information on monitoring Windows event logs by reading information from and writing information to them.

Monitoring Windows Services
Provides information on how to monitor existing Windows services.

Monitoring and Managing Windows Processes
Explains the process of creating a component to automate administrative tasks for Windows processes.

Related Sections

Programming with Components
Presents a high-level overview of component programming and authoring.

System Monitoring Walkthroughs

The following walkthroughs introduce you to the use of the **PerformanceCounter** component, the **EventLog** component, and the Windows Service application and **ServiceController** component. These components allow you to work with various built-in pieces of Windows such as event logs, performance counters, and services.

Walkthrough: Retrieving Categories and Counters
: Presents step-by-step instructions on how to retrieve lists of performance counter categories and the counters they contain.

Walkthrough: Changing and Retrieving Performance Counter Values
: Presents step-by-step instructions on retrieving raw and calculated values from a counter using the **PerformanceCounter** component.

Walkthrough: Exploring Event Logs, Event Sources, and Entries
: Presents step-by-step instructions on how to retrieve event logs and their entries and write additional entries to them.

Walkthrough: Creating a Windows Service Application in the Component Designer
: Presents step-by-step instructions on how to create and interact with a Windows service.

Walkthrough: Installing an Event Log Component
: Describes how to use an installation component to configure server resources needed for your **EventLog** component instance.

Walkthrough: Managing a Windows Process
: Presents step-by-step instructions on how to create a **Process** component instance and use it to interact with a Windows process.

See Also

Component Programming Walkthroughs | Performance Counter Walkthroughs | Walkthroughs in Visual Basic and Visual C# | Visual Basic Code Example Topics

Monitoring Performance Thresholds

Windows performance counters allow your applications and components to publish, capture, and analyze the performance data that applications, services, and drivers provide. You can use this information to determine system bottlenecks and fine-tune system and application performance. For example, you might use a performance counter to track the amount of time required to process an order or query a database, or you might monitor the size of a message queue and write code that performs a specific action whenever the queue reaches a preset limit.

Using the **PerformanceCounter** component and language features in Visual Studio .NET and the Microsoft .NET Framework SDK, you can easily connect to performance counters on both local and remote computers and retrieve values from these counters. You can also write values to existing custom counters (those created with .NET) on the local computer and create your own custom counters on computers that have Windows installed.

In This Section

Introduction to Monitoring Performance Thresholds
Provides basic information on performance counter features.
Creating PerformanceCounter Component Instances
Gives directions for creating instances of and configuring **PerformanceCounter** components in your projects.
Configuring PerformanceCounter Component Instances
Describes how to set properties for the **PerformanceCounter** components you create.
Category and Counter Management
Describes the processes for creating, deleting, and querying performance counters and the categories that classify them.
Writing Values to Performance Counters
Gives instructions for incrementing or decrementing a performance counter's value.
Performance Counter Value Retrieval
Explains the process of reading values from and writing values to a counter.
Working with Performance Counters in Server Explorer
Gives directions for using Server Explorer to investigate the performance counters on a computer.
Performance Counter Walkthroughs
Lists walkthroughs that show you how to use both system performance counters and custom performance counters.

Related Sections

PerformanceCounter Class
Describes the major features of the **PerformanceCounter** class.
PerformanceCounterCategory Class
Describes the major features of the **PerformanceCounterCategory** class.
Performance Counter Programming Architecture
Explains the language elements used in **PerformanceCounter** component programming.
Performance Counter Types
Describes the types of performance counters that you might work with.
Creating Installation Components
Describes how to use the Add Installer link in the Properties window to add an installation component to applications with **PerformanceCounter** component instances.
Visual Basic and Visual C# Projects
Describes the projects types used in this chapter and how to choose between them.
Deploying Applications and Components
Lists the major deployment topics and the information they contain.
Manageability
Discusses best practices for managing distributed applications.
Programming with Components
Presents a high-level overview of component programming and the topics available on it.

Introduction to Monitoring Performance Thresholds

The mechanism by which Windows collects performance data on various system resources is the *performance counter*. Windows contains a pre-defined set of performance counters with which you can interact; some of these are found on all Windows 2000 computers and some are custom to specific applications and are found only on certain computers. Each counter is related to a specific area of system functionality. Examples include counters that monitor a processor's busy time, memory usage, or the number of bytes received over a network connection. You use an instance of the **PerformanceCounter** component to connect directly to existing performance counters and interact with their data in dynamic ways.

A performance counter monitors the behavior of *performance objects* on a computer. These include physical components, such as processors, disks, and memory, and system objects, such as processes and threads. In Visual Studio and the .NET SDK, these performance objects are shown as categories; system counters related to the same performance object are grouped into categories that indicate their common focus. When you create an instance of the **PerformanceCounter** component, you first indicate the category with which the component will interact and then choose a counter to interact with from within that category.

For example, a category of counters on Windows is the Memory category. System counters within this category track data such as bytes available and bytes cached. To deal with bytes cached in your application, you create an instance of the **PerformanceCounter** component and connect it to the Memory category, then pick the appropriate counter in that category (Cached Bytes, in this case).

In some situations, categories are further subdivided into instances. Instances track data about multiple occurrences of the object that a category relates to. It is important to note that instances apply to the category as whole, rather than to individual counters. All counters within a category have each instance defined for the category. For example, the Process category contains instances called "Idle" and "System." Each counter within the Process category specifies data in these two ways, showing information about either idle processes or system processes. The structure of the category and counters looks like this:

Categories, Instances, and Counters

	Category: Process	
	Instance1:Idle	Instance2:System
Counter: Elapsed Time	✓	✓
Counter: Thread Count	✓	✓

As you can see, the Process category contains two instances, so both counters in the category also contain instances of Idle and System. For more information on instances, see Adding and Removing Performance Counter Instances.

Although your system makes available many more categories of counters, the default categories you'll probably interact with most frequently include the Cache, Memory, Objects, PhysicalDisk, Processor, Server, System, and Thread categories.

Performance Counter Data

Performance counters record values about various parts of the system. These values are not stored as entries, but are persisted for as long as a handle remains open for the given category in memory. The process of retrieving data from a performance counter is called *sampling*. When you sample, you either retrieve the immediate value of a counter or a calculated value.

Depending on how a counter is defined, its value might be the most recent aspect of resource utilization, also called the *instantaneous value*, or it might be the average of the last two measurements over the period of time between samples. For example, when you retrieve a value from the Process category's Thread Count counter, you retrieve the number of threads for a particular process as of the last time this was measured. This is an instantaneous value. However, if you retrieve the Memory category's Pages/Sec counter, you retrieve a rate per second based on the average number of memory pages retrieved during the last two samples.

Resource usage can vary dramatically based on the work being done at various times of day. Because of this, performance counters that show usage ratios over an interval are a more informative measurement than averages of instantaneous counter values. Averages can include data for service startup or other events that might cause the numbers to go far out of range for a brief period, thereby skewing results.

The **PerformanceCounter** component provides facilities for the most common Windows performance monitoring requirement, namely, connecting to an existing counter on the server and reading and writing values to it. Additional functionality, such as complex data modeling, is available directly through Windows Performance Monitor. For example, you can use Performance Monitor to chart the data a counter contains, run reports on the data, set alerts, and save data to a log.

Working with Performance Counters in .NET Applications

Using the **PerformanceCounter** component, you can connect to the existing counters installed on the servers to which you have access, or you can create your own custom performance counters. You can read data from any counter, and you can write to your own custom counters.

When you connect to an existing performance counter, you do so by specifying the computer on which the counter exists, the category for the counter, and the name of the counter itself. Additionally, you have the option of specifying the instance of the counter you want to use, if the counter contains more than one instance. You can then read any and all data from the counter. You can also enumerate the existing categories, counters, and instances on the computer by using code, or you can use Server Explorer to see a list of existing counters on the computer.

When you create a new counter, you must specify a completely new category for your local computer. You cannot add a new counter to an existing category or create new counters on a remote computer. However, you can write data to and read data from any custom counters you create.

You can expand the Performance Counters node in Server Explorer to see a list of the counters that exist on that server (by category) and the instances by which they are classified. You can also drag a specific performance counter or instance from Server Explorer to your Component Designer in Microsoft Visual Studio .NET to create an instance of the **PerformanceCounter** component that automatically has its properties set to point to the selected counter.

If you are using a **PerformanceCounter** component in an ASP.NET application, the default settings of the ASPNET user account restrict access to performance counters. The ASPNET user account, by default, can write to but not read from performance counters, and it cannot create new categories. You can use impersonation with the ASPNET account to allow creation of new categories. The impersonation identity must have sufficient privileges to create categories. If your application needs performance counters that can be specified before deployment, they can be created by the deployment project. For more information, see ASP.NET Web Application Security.

> **Note** You can create a new counter category in Server Explorer by right-clicking the Performance Counters node and choosing the appropriate menu command. You can also edit an existing category by right-clicking its specific node.
>
> **Note** The Servers node of Server Explorer is not available in the Standard Edition of Visual Basic and Visual C# .NET. For more information, see Visual Basic Standard Edition Features or Visual C# Standard Edition Features.

For more information, see Introduction to Server Explorer.

Restrictions

In this release, you may have to restart the Performance Monitor (Perfmon.exe) that is installed with Windows 2000 when you create custom performance counters before you can see the custom counter in that application.

The **PerformanceCounter** class is not fully supported on Microsoft Windows NT version 4.0. You can read from the system counters, but you cannot create, write to, or delete custom counters.

See Also

Performance Counter Data Analysis | Performance Counter Programming Architecture | Performance Counter Lifetime | Performance Counter Types

Performance Counter Data Analysis

Analyzing performance counter data consists of examining counter values that are reported while your system is performing various operations. During this analysis you could determine which processes are most active and which programs or threads, if any, are monopolizing a resource. Using this type of performance data analysis, you can understand how your system is responding to workload demands.

As a result of this analysis, you might find that your system performs satisfactorily at times and unsatisfactorily at others. Depending on the causes of these variations and the degree of difference, you might choose to take corrective action or to accept these variations and delay tuning or upgrading resources to a later time.

The level of system performance that you consider acceptable when your system is handling a typical workload and running all required services is its baseline. The baseline performance is a subjective standard that the administrator determines based on the work environment. It might correspond to a range of counter values, including some that are temporarily unacceptable, but which generally indicate the best possible performance under the administrator's specific conditions.

Determining Acceptable Values for Counters

A large set of factors determines the normal operating range for each counter in a typical system. You must drive your system through a large number of conditions and develop a sense for normal operating ranges for your equipment. You should record these typical values in log files for future reference. Then, as you make changes to your workload or your hardware, you can refer to your earlier experience as a baseline.

In general, deciding whether or not performance values are acceptable is a subjective judgment that varies significantly with variations in user environments. The values you establish as the baseline for your organization are the best basis for comparison. You should stress test your applications before releasing them and determine the performance thresholds that cause problems. After releasing the application to a live site, you can monitor the performance and take appropriate action if these thresholds are reached.

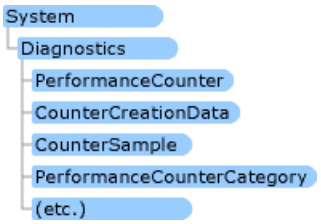
See Also

Introduction to Monitoring Performance Thresholds | Performance Counter Value Retrieval

Performance Counter Programming Architecture

The **PerformanceCounter** component uses these portions of the Microsoft .NET Framework namespaces:

PerformanceCounter Namespace



When you add an instance of the **PerformanceCounter** component to your application, Visual Studio .NET automatically creates the references and import statements you need to access the namespace and classes. However, if you are creating your **PerformanceCounter** instances in code, you might need to create a reference to the System.Diagnostics namespace and add a statement to your project importing (in Visual Basic) or using (in C#) the appropriate namespace.

For more information on adding and removing project references, see Adding and Removing References.

Key Performance Counter Classes

The following are the major classes associated with **PerformanceCounter** components:

- **System.Diagnostics.**PerformanceCounter — the base class used to create **PerformanceCounter** components and interact with their values.
- **System.Diagnostics.**PerformanceCounterCategory — provides access to the categories for which performance counters exist on a server and all of their corresponding properties and methods. You use methods in this class to create new categories and custom counters, to verify that categories and counters exist, and to read the existing counters in a category.
- **System.Diagnostics.**CounterCreationData — used to create multiple counters for a category; allows you to explicitly choose the counter type for each counter you create.
- **System.Diagnostics.**InstanceData — used to retrieve multiple instances of a counter and all of their corresponding properties and methods.

Key Programming Elements in the PerformanceCounter and PerformanceCounterCategory Classes

You can use the following methods to interact with performance counters:

- Use the Create method on the PerformanceCounterCategory class to define a new category containing one or more counters. Similarly, the Delete method removes a user-defined category and the counters it contains.
- Use the Exists method on the **PerformanceCounterCategory** class to determine if a category of counters already exists.
- Use the Increment or IncrementBy methods on the PerformanceCounter class to increment a counter value by one or by the specified integer and to return the new value of the counter. Similarly, you can use the Decrement method to reduce a counter's value.
- Use the NextValue method on the **PerformanceCounter** class to return the next calculated value for the sample, or the NextSample method to return the next sampling.
- Use the GetCounters method on the **PerformanceCounterCategory** class to return an array of counters in a category.

See Also

Adding and Removing References | Introduction to Monitoring Performance Thresholds

Performance Counter Lifetime

It is important to understand that performance counter values are values rather than entries — when you write a value to a performance counter, you are not making a permanent entry in the counter as you do when you write an entry to an event log. Rather, performance counter values are transitory. When the last **PerformanceCounter** component to reference a particular system counter is disposed of, the counter resets itself to zero.

There are several ways you can manage the lifetime of a performance counter if you want to retain values for a longer period of time than the default counter behavior allows:

- You can run the Performance Monitor application on the server where the counter lives. As long as Performance Monitor is open and actively monitoring the category for your counter, the reference to the counter is maintained and counter values continue to accumulate.
- You can make sure that there is always an instance of the **PerformanceCounter** component connected to the particular counter for which you want to maintain values.

See Also

Introduction to Monitoring Performance Thresholds | Performance Counter Types

Performance Counter Types

Performance counters have a type definition that determines how they interact with monitoring applications and the counter values they contain. For example, suppose you are creating an instance of the **PerformanceCounter** component to count the number of orders processed per second in your e-commerce application. You create a performance counter that increments by one after processing each order. In this case, you would set the CounterType to RateOfCountsPerSecond32. This type tells the counter to calculate the number of orders processed per second by taking two samplings and dividing by the timing of the samples. The more samplings taken, the more relevant the data becomes.

Choosing the Right Counter Type

There are many types of performance counters you can choose from when you create a counter. Some of the counters have specific calculations associated with them, and some contain uncalculated values only. In general, there are a few counter types that you will use most frequently. The following table lists these types by common usage patterns.

If You Need To	Use This Type of Counter	Example
Maintain a simple count of items, operations, and so on.	**NumberofItems32**	You might use this counter type to track the number of orders received as a 32-bit number.
Maintain a simple count with a higher capacity	**NumberofItems64**	You might use this counter type to track orders for a site that experiences very high volume; stored as a 64-bit number.
Track the amount per second of an item or operation	**RateOfCountsPerSecond32**	You might use this counter type to track the orders received per second on a retail site; stored as a 32-bit number.
Track the amount per second with a higher capacity	**RateOfCountsPerSecond64**	You might use this counter type to track the orders per second for a site that experiences very high volume; stored as a 64-bit number.
Calculate average time to perform a process or to process an item	**AverageTimer32**	You might use this counter type to calculate the average time an order takes to be processed; stored as a 32-bit number.

A few caveats on counter type:

- Counter type affects whether the Calculate method will work or raise an error. You cannot perform a calculation on counters of type **NumberofItems32**, **NumberofItems64**, **NumberofItemsHEX32**, **NumberofItemsHEX64**, or **RawBase**.
- If you are comparing two samples, they must be of the same counter type. For more information on samples, see Performance Counter Value Retrieval.

For a full list of the types available, see PerformanceCounterType Enumeration.

See Also

Introduction to Monitoring Performance Thresholds | Creating Custom Performance Counters | Creating Performance Counter Categories

Creating PerformanceCounter Component Instances

You create an instance of the **PerformanceCounter** component when you want to make available performance data about your application or monitor the performance of resources on your system. **PerformanceCounter** components allow you to connect to existing performance counters, write values to custom counters, and react to values the counters receive.

PerformanceCounter components have no visual user interface. If you add an instance of the **PerformanceCounter** component to a visual designer (such as the Windows Forms Designer), the component appears in a small area below the bottom border of the form. This is known as the component tray and acts as a place to display all of the non-visual items associated with the form.

> **Note** Nonvisual classes that inherit from **System.ComponentModel.Component** support a visual design surface similar to the component tray, on which your **PerformanceCounter** component and other components are arranged. The arrangement of items on this designer is not important, as the interface you see will never be displayed to the application's end users.

There are several ways you can create an instance of the **PerformanceCounter** component:

- You can locate the counter you want in Server Explorer and add it to your designer, creating an instance of the **PerformanceCounter** component that is preconfigured to point to that counter.
- You can drag an instance of the **PerformanceCounter** component from the **Components** tab of the Toolbox to a form or other designer.
- You can create an instance of the **PerformanceCounter** component in code.

To create an instance of the PerformanceCounter components from Server Explorer

1. In Server Explorer, expand the **Servers** node, and then expand the node for the server you want to view.

   > **Note** If the server you want is not listed, you need to add it. For more information, see Accessing and Initializing Server Explorer.

   > **Note** The Servers node of Server Explorer is not available in the Standard Edition of Visual Basic and Visual C# .NET. For more information, see Visual Basic Standard Edition Features or Visual C# Standard Edition Features.

2. Locate and expand the **Performance Counters** node beneath it.
3. Locate the counter you want to add to your project, and then right-click it.
4. Click **Add to Designer**.
5. Configure your component. For more information, see Configuring PerformanceCounter Component Instances.

To create an instance of the PerformanceCounter component from the Toolbox

1. Access the **Components** tab of the Toolbox.
2. Select the **PerformanceCounter** icon and drag it to the designer surface for your form or component.
3. Configure your component. For more information, see Configuring PerformanceCounter Component Instances.

To create an instance of the PerformanceCounter component programmatically

1. Create an instance of the **PerformanceCounter** class in code. You code might look like this:

```
' Visual Basic
Dim myCounter As New System.Diagnostics.PerformanceCounter()

// C#
System.Diagnostics.PerformanceCounter myCounter =
   new System.Diagnostics.PerformanceCounter();
```

2. Configure any other necessary properties for your component. For more information, see Configuring PerformanceCounter Component Instances.

See Also

Configuring PerformanceCounter Component Instances | Accessing and Initializing Server Explorer

Configuring PerformanceCounter Component Instances

There are a variety of properties you must set on the **PerformanceCounter** components you create. These determine the counter with which the component interacts. The properties you must set include:

- CategoryName, which determines the performance object with which the component will interact. All performance counters in Windows are grouped by categories, such as Memory, Processes, and Processor.
- CounterName, which identifies the counter within the specified category with which the component will interact.
- InstanceName, which identifies the instance of the category with which the component will interact. Note that many categories do not contain instances, so this property might be left blank, indicating a single instance counter.

  > **Note** If the Category has been set to a category that was not defined by the user, you must choose from existing counters and instances. You can only create new counters and instances in custom categories. For example, you cannot create a new counter in the Memory category, but if you create a new category called Orders, you can define as many counters and instances as you need within that category.

- MachineName, which identifies the server on which the performance counter with which you want your component to interact resides. You can set this property to "." to indicate the local machine or leave it blank to default to this value.
- ReadOnly, which determines whether any custom counter you create can be written to. By default, any **PerformanceCounter** component instances you configure to work with existing Windows counters function in read-only mode and will not allow you to change values in these counters. When you configure your component instance to work with a custom counter, however, you have a choice to accept the default of working in read-only mode, or resetting the property value to **false** to allow values to be written.

  > **Note** You can write counters only on the local machine. If you intend to read counter values, you can choose any computer to which you have access.

To configure an instance of the PerformanceCounter component

1. Create an instance of the **PerformanceCounter** component. For more information, see Creating PerformanceCounter Component Instances.
2. Indicate the counter with which your component should interact by setting the following properties.

Property	Setting
MachineName	Any server to which you have access.
CategoryName	Any category that exists on the server.
CounterName	Any counter within the selected category.

3. If the counter with which you are interacting has several instances, set the appropriate one as the value of the **InstanceName** property.
4. If you are working with a custom counter and want write access, set the **ReadOnly** property to **false**.

   > **Note** You can specify these properties either in the Properties window, in code, or by using special forms of the constructor for the **PerformanceCounter** component instance.

   The following example shows how you would set these values programmatically to connect to an existing Windows performance counter called "aborted transactions" on the local server.

```
' Visual Basic
' Connect to an existing Windows counter and category
Dim abortedTransactions As New PerformanceCounter()
abortedTransactions.CategoryName = _
   "distributed transaction coordinator"
abortedTransactions.CounterName = "aborted transactions"
abortedTransactions.MachineName = "."
' Connect to a custom counter and category in writable mode
PerformanceCounterCategory.Create("orders", "desc", "milk", "desc")
Dim myCounter2 As New PerformanceCounter("orders", "milk", False)

// C#
// Connect to an existing Windows counter and category
```

```
System.Diagnostics.PerformanceCounter abortedTransactions =
   new System.Diagnostics.PerformanceCounter();
abortedTransactions.CategoryName =
   "distributed transaction coordinator";
abortedTransactions.CounterName = "aborted transactions";
abortedTransactions.MachineName = ".";
// Connect to a custom counter and category in writable mode
System.Diagnostics.PerformanceCounterCategory.Create(
   "orders", "desc", "milk", "desc");
System.Diagnostics.PerformanceCounter myCounter2 =
   new System.Diagnostics.PerformanceCounter("orders", "milk", false);
```

Note In this code, "." represents the local machine.

See Also

Creating PerformanceCounter Component Instances | Creating Custom Performance Counters | Category and Counter Management

Category and Counter Management

You create categories and custom performance counters if you want to track data that is not captured by the standard counters provided with Microsoft Windows. For example, you might create a custom counter to track the total number of users logged on to your Web site, or to track the orders processed per second by your site. When you create a counter, you add it to a performance counter category, and you assign it a type that governs how it will behave. For information on choosing the correct type, see Performance Counter Types.

There are several ways you can create custom counters:

- You can create a single counter within a category using the Create method on the **PerformanceCounter** class. For more information, see Creating Performance Counter Categories.
- You can create multiple counters within a single category using the CounterCreationData class. For more information, see Creating Custom Performance Counters.

You must create counters in a user-defined category rather than in the performance objects (called categories in Visual Studio .NET and the .NET Framework) defined by Windows. That is, you cannot create a new counter within the Processor category or any other system-defined categories. Additionally, you must create a counter in a new category; adding a counter to an existing user-defined category will raise an exception.

The following restrictions apply to the process of creating new counters:

- You can create custom counters on the local computer, provided that you have administrator access.
- If you do not specify a counter type when creating the counter and its category, it will default to NumberofItems32.
- You cannot create new counters within existing categories. If you need to add counters to categories that already exist, the only way you can do so is to delete the category and recreate it with all of its contents, including the new counters you want to add.
- There are security restrictions that affect your ability to create counters and categories. For more information, see Introduction to Monitoring Performance Thresholds.
- The **PerformanceCounter** class is not fully supported on Microsoft Windows NT version 4.0. You can read from the system counters, but you cannot create, write to, or delete custom counters.

See Also

Creating Custom Performance Counters | Creating Performance Counter Categories | Deleting Performance Counter Categories | Determining If Counters and Categories Exist | Adding and Removing Performance Counter Instances | Performance Counter Types

Creating Custom Performance Counters

When you create a new counter, you first create a category, and then specify one or more counters to be placed in it. You can do this in one of the following ways:

- You can use Server Explorer to create one or more counters at design time.
- You can use the CounterCreationData class to create one or more counters programmatically, then add them to a collection of type CounterCreationDataCollection and pass the collection as a parameter of a special form of the Create method.
- You can use the **Create** method on the **PerformanceCounterCategory** class to create a single counter within a new category. For more information, see Creating Performance Counter Categories.

There are two special issues to be aware of when creating counters and categories. First, you cannot create custom categories and counters on remote machines. Second, your interaction with custom counters and categories is restricted to read-only mode unless you explicitly specify otherwise. In read-only mode, you cannot increment or decrement them or set the raw value or other values within them. You can use the **ReadOnly** property to set a custom counter to writable mode.

It is important to note the difference between creating a counter and creating an instance of the **PerformanceCounter** component. When you create a counter, you are creating a new category and its associated counters in the Windows operating system, rather than a component in your project or application. When you create an instance of the **PerformanceCounter** component, you create a component inside your Visual Studio .NET project that references an external counter.

> **Note** There are security restrictions that affect your ability to use performance counters. For more information, see Introduction to Monitoring Performance Thresholds.
>
> **Security Note** When you create a performance counter, be aware that the resource may already exist. Another process, perhaps a malicious one, may have already created the resource and have access to it. When you put data in the performance counter, the data is available to the other process.
>
> **Note** The **PerformanceCounter** class is not fully supported on Microsoft Windows NT version 4.0. You can read from the system counters, but you cannot create, write to, or delete custom counters.

To create a new category and custom performance counter at design time

1. Open Server Explorer and expand the node for the server you want to view.

   > **Note** If the server you want is not listed, you need to add it. For more information, see Accessing and Initializing Server Explorer.
   >
   > **Note** The Servers node of Server Explorer is not available in the Standard Edition of Visual Basic and Visual C# .NET. For more information, see Visual Basic Standard Edition Features or Visual C# Standard Edition Features.

2. Right-click the **Performance Counters** node and select **Create New Category**.

   The Performance Counter Builder dialog box appears.

3. Enter a name and description for the category you want to create.

   > **Note** If you specify the name of an existing category, an error will be raised. To overwrite an existing counter category, you must first delete the category using the **Delete** method, and then add a new category.

4. In the **New Counter** frame, do the following:
   a. Specify a name for the counter you want to create within the category.
   b. Choose a type from the **Type** drop-down list.
   c. Enter a description for the counter.
5. Click **Add to List**.
6. Repeat steps 4 and 5 for each counter you want to create in this category.

   > **Tip** Before you exit the dialog box, you can select any of the counters in the Counters list and edit their values, or delete the counters.
   >
   > **Note** Counters and categories you create in the dialog box are read-write enabled by default, but your interaction with them via an instance of the **PerformanceCounter** component will be restricted to read-only unless you specify otherwise.

To create a new category and set of performance counters programmatically

1. Create a collection of type CounterCreationDataCollection.
2. Create the counters you want to create as objects of type CounterCreationData and set their necessary properties.
3. Add the **CounterCreationData** objects to the collection by calling the collection's Add method.
4. Call the Create method on the **PerformanceCounterCategory** class and pass the collection to it.

   The following example shows how to create a series of counters and pass them to the category when you create it:

```
' Visual Basic
' Create a collection of type CounterCreationDataCollection.
Dim CounterDatas As New CounterCreationDataCollection()
' Create the counters and set their properties.
Dim cdCounter1 As New CounterCreationData()
Dim cdCounter2 As New CounterCreationData()
cdCounter1.CounterName = "MyCounter1"
cdCounter1.CounterHelp = "help string"
cdCounter1.CounterType = PerformanceCounterType.NumberOfItems64
cdCounter2.CounterName = "MyCounter2"
cdCounter2.CounterHelp = "help string 2"
cdCounter2.CounterType = PerformanceCounterType.NumberOfItems64
' Add both counters to the collection.
CounterDatas.Add(cdCounter1)
CounterDatas.Add(cdCounter2)
' Create the category and pass the collection to it.
PerformanceCounterCategory.Create( _
   "Multi Counter Category", "Category help", CounterDatas)

// C#
// Create a collection of type CounterCreationDataCollection.
System.Diagnostics.CounterCreationDataCollection CounterDatas =
   new System.Diagnostics.CounterCreationDataCollection();
// Create the counters and set their properties.
System.Diagnostics.CounterCreationData cdCounter1 =
   new System.Diagnostics.CounterCreationData();
System.Diagnostics.CounterCreationData cdCounter2 =
   new System.Diagnostics.CounterCreationData();
cdCounter1.CounterName = "Counter1";
cdCounter1.CounterHelp = "help string1";
cdCounter1.CounterType = System.Diagnostics.PerformanceCounterType.NumberOfItems64;
cdCounter2.CounterName = "Counter2";
cdCounter2.CounterHelp = "help string 2";
cdCounter2.CounterType = System.Diagnostics.PerformanceCounterType.NumberOfItems64;
// Add both counters to the collection.
CounterDatas.Add(cdCounter1);
CounterDatas.Add(cdCounter2);
// Create the category and pass the collection to it.
System.Diagnostics.PerformanceCounterCategory.Create(
   "Multi Counter Category", "Category help", CounterDatas);
```

See Also

Accessing and Initializing Server Explorer | Creating Performance Counter Categories | Category and Counter Management

Creating Performance Counter Categories

You can create a new category to contain custom counters. For example, if you plan to create a series of counters to track various data about the orders processed on your Web site, you might create a category called OrderData on your server and then create the counters you need within it.

Creating a category is not a distinct process from creating the counters that are placed within it; counters can only be created at the point when you create the category itself. You cannot create categories and counters on or remove them from remote computers.

There are several ways you can create counters and categories:

- You can use the Create method on the PerformanceCounterCategory class to create a new category and simultaneously create a single performance counter within it.
- You can create an array of CounterCreationData objects and pass the array as a parameter of the **Create** methods, creating a set of counters within the category. For more information on this approach, see Creating Custom Performance Counters.
- You can use the Performance Counter dialog box, launched from Server Explorer, to create multiple counters while simultaneously creating a new category. For more information on this approach, see Creating Custom Performance Counters.

> **Note** There are security restrictions that affect your ability to use performance counters. For more information, see Introduction to Monitoring Performance Thresholds.

> **Note** The **PerformanceCounter** class is not fully supported on Microsoft Windows NT version 4.0. You can read from the system counters, but you cannot create, write to, or delete custom counters.

To create a category and a single counter within it

- Call the **Create** method on the **PerformanceCounterCategory** class and specify the following parameters:

Parameter	Value
CategoryName	Any category name that is not already in use on this server.
CategoryHelp	A description of the category.
CounterName	A name for the counter.
CounterHelp	A description of the counter. This text is displayed in Windows Performance Monitor when a user selects a counter and clicks the **Explain** button.

The following example shows how you might create a simple category with the **Create** method:

```
' Visual Basic
Sub CreateCustomCounter()
   PerformanceCounterCategory.Create("MyCategory", "Desc", _
      "MyCounter", "Counterdesc")
End Sub

// C#
void CreateCustomCounter()
{
   System.Diagnostics.PerformanceCounterCategory.Create(
      "MyCategory", "Desc", "MyCounter", "Counterdesc");
}
```

> **Note** By default, the counters created with this code will be read-write enabled, but your interaction with them via a **PerformanceCounter** component instance will be read-only unless you specify otherwise. You can change the value of the **ReadOnly** property for a component instance to **false** if you want to make modifications to a counter.

See Also

Deleting Performance Counter Categories

You can delete performance counter categories when they are no longer needed. You can only delete categories that are not Windows-standard performance objects; that is, you can only delete categories that have been user-defined. In addition, you cannot create or delete categories on remote computers. You might delete a category if you have created a custom category associated with an application and you are removing that application from the system. For example, if you have an order entry application that creates counters in an OrderProcessing category, you would delete that category as part of uninstalling your application.

You use the Delete method on the PerformanceCounterCategory class to remove a category from the system. When you delete a category, all of the counters and instances associated with it are also removed.

It is important to note the difference between deleting a counter and deleting an instance of the **PerformanceCounter** component. When you use the **Delete** method, you are deleting a category and its associated counters from the Windows operating system, rather than removing an instance of a counter from your project or application. When you delete an instance of the **PerformanceCounter** component, you remove a component inside your project that references an external counter.

> **Note** The **PerformanceCounter** class is not fully supported on Microsoft Windows NT version 4.0. You can read from the system counters, but you cannot create, write to, or delete custom counters.

To delete a performance counter category from Server Explorer

1. In Server Explorer, locate the performance counter category you want to remove. For more information, see Accessing and Initializing Server Explorer.

   > **Note** The Servers node of Server Explorer is not available in the Standard Edition of Visual Basic and Visual C# .NET. For more information, see Visual Basic Standard Edition Features or Visual C# Standard Edition Features.

2. Right-click the category and select **Delete**.

To delete a performance counter category programmatically

- Call the **Delete** method on the **PerformanceCounterCategory** class, specifying the category to remove as a parameter.

See Also

Category and Counter Management | Adding and Removing Performance Counter Instances

Determining If Counters and Categories Exist

You can use the CounterExists method on the **PerformanceCounterCategory** class to determine if a given performance counter exists within a particular category on either the local or a remote computer. You might do this before creating a new counter to prevent an error from occurring if another counter with that name exists.

In addition to determining whether counters exist, you can determine whether a given category exists by using the Exists method on the **PerformanceCounterCategory** class. You might do this if you are creating a custom counter and want to determine if the category for it already exists. The **Create** method will raise an error if the category you specify has already been created.

Both the **Exists** and the **CounterExists** methods return **true** if the item is found and **false** if it is not.

To determine if a counter exists

- Call the **CounterExists** method on the **PerformanceCounterCategory** class, specifying the following parameters.

Parameter	Value
CounterName	The name of the counter you want to query.
CategoryName	Any category of performance objects on the server.
MachineName	The server on which to locate the category and counter.

  Note In Visual Basic, the **MachineName** parameter is optional; if left blank it defaults to the local machine. In C#, you can use an overload of the **CounterExists** method if you do not want to specify the machine name.

To determine if a category exists

- Call the **Exists** method on the **PerformanceCounterCategory** class, specifying the following parameters.

Parameter	Value
CategoryName	Any category you want to query.
MachineName	The server on which to locate the category.

  The following example shows how to use an **If** statement to determine whether a category exists before creating a category and counter:

```
' Visual Basic
If Not (PerformanceCounterCategory.Exists("MyCat")) Then
   PerformanceCounterCategory.Create( _
      "MyCat", "Description", "MyCounter", "Description")
End If

// C#
if (!System.Diagnostics.PerformanceCounterCategory.Exists("MyCat"))
{
   System.Diagnostics.PerformanceCounterCategory.Create(
      "MyCat", "Description", "MyCounter", "Description");
}
```

See Also

Category and Counter Management | Writing Values to Performance Counters | Performance Counter Value Retrieval

Adding and Removing Performance Counter Instances

You can add and remove instances dynamically as you work with counters and categories. For example, you might add an instance for each user of your Web-based retail application so that you could track information about their actions, and then remove the instance when the user's session expires.

You add instances by setting a raw value for a counter. If no instance exists for the counter, an instance is created the first time you set the **RawValue** property, and all subsequent actions on the raw value are assumed to impact that instance if no other is specified. You can create additional instances by specifying a new instance name and then setting a value after it.

> **Note** The process of setting the instance's value, rather than specifying the name of a new instance, creates the instance.

Unlike counters, which cannot be added to an existing category except as part of category creation, instances can be added to and removed from user-defined categories at any time. You can use the **Instance** property to switch between one instance and another.

You can use the RemoveInstance method to remove an instance of a custom performance counter from memory. For example, suppose you have a Web-based retail application that uses a category called OrderInProgress, and within that you maintain instances for each user's current shopping cart. When a user first adds an item to their cart, your application creates a new instance for that user. When the user completes the order, your application deletes the instance. During the order, you update the instance with counters such as NumberofItemsinCart, TimeSinceCreation, and NumberofItemsAddedPerSecond.

You cannot remove instances from the performance counters that are a default part of Windows. If your **PerformanceCounter** component does not refer to a valid instance, this method will throw an exception.

> **Note** The **PerformanceCounter** class is not fully supported on Microsoft Windows NT version 4.0. You can read from the system counters, but you cannot create, write to, or delete custom counters.

To add a performance counter instance

1. Create your categories and counters normally. For more information, see Creating Performance Counter Categories.
2. Set the **Instance** property to a unique name for the instance, and then set the instance's RawValue property.

   The following code shows how to create several instances of an existing performance counter category:

```
' Visual Basic
' Assumes the category and counter have already been created.
Dim myCounter As New System.Diagnostics.PerformanceCounter( _
   "cat", "counter", "instance1", False)
' Set the raw value to automatically create instance1.
myCounter.RawValue = 100
' State that you will now be working with a different instance.
myCounter.InstanceName = "instance2"
' Setting the value actually creates instance2.
myCounter.RawValue = 200

// C#
// Assumes category and counter have been created.
System.Diagnostics.PerformanceCounter myCounter =
   new System.Diagnostics.PerformanceCounter(
   "cat","counter","instance1", false);
// Set the raw value to automatically create instance1.
myCounter.RawValue = 100;
// State that you will now be working with a different instance.
myCounter.InstanceName = "instance2";
// Setting the value actually creates instance2.
myCounter.RawValue = 200;
```

To remove a performance counter instance

1. Create an instance of the **PerformanceCounter** component that is connected to the counter from which you want to

remove an instance. For more information, see Creating PerformanceCounter Component Instances.

2. Set the InstanceName property to the instance you want to delete.
3. Call the **RemoveInstance** method on your component.

   The following example shows how to remove an instance called Reference from a counter:

```
' Visual Basic
' Assumes that you have configured PerformanceCounter1 to
' interact with the appropriate counter.
PerformanceCounter1.InstanceName = "Reference"
PerformanceCounter1.RemoveInstance()

// C#
// Assumes that you have configured PerformanceCounter1 to
// interact with the appropriate counter.
PerformanceCounter1.InstanceName = "Reference";
PerformanceCounter1.RemoveInstance();
```

See Also

Creating PerformanceCounter Component Instances | Category and Counter Management

Writing Values to Performance Counters

You write a value to a counter by incrementing the counter's current raw value by a positive or negative number. You do this using the **IncrementBy** method on the **PerformanceCounter** class.

> **Note** Incrementing by a negative number decrements the counter by the absolute value of the number. For example, incrementing with a value of 3 will increase the counter's raw value by three. Incrementing with a value of –3 will decrease the counter's raw value by three.

In addition, you can use the **Increment** and **Decrement** methods to increase or decrease a counter's values by one. These methods are processed much more quickly than **IncrementBy**.

You can only increment values on custom counters; by default, your interactions with system counters via a **PerformanceCounter** component instance are restricted to read-only mode. Before you can increment a custom counter, you must set the **ReadOnly** property on the component instance with which you are accessing it to **false**.

> **Note** There are security restrictions that affect your ability to use performance counters. For more information, see Introduction to Monitoring Performance Thresholds.

> **Note** The **PerformanceCounter** class is not fully supported on Microsoft Windows NT version 4.0. You can read from the system counters, but you cannot create, write to, or delete custom counters.

To write values to performance counters

1. Create a **PerformanceCounter** instance and configure it to interact with the desired category and counter. For more information, see Creating PerformanceCounter Component Instances or Configuring PerformanceCounter Component Instances.
2. Write the value using one of the following methods:

To	Call this method	Parameter
Increase the raw value by one	Increment	None
Decrease the raw value by one	Decrement	None
Increase the raw value by greater than one	IncrementBy	A positive integer
Decrease the raw value by greater than one	IncrementBy	A negative integer
Reset the raw value to any integer, rather than incrementing it	RawValue	A positive or negative integer

The following code shows how to set values for a counter in various ways. This code assumes that you are working on a Windows Form that contains a text box named txtValue and three buttons: one that increments the raw value by the number entered in the text box, one that decrements the raw value by one, and one that sets the raw value of the counter to the value set in the text box.

```
' Visual Basic
Private Sub btnIncrement_Click(ByVal sender As System.Object, _
   ByVal e As System.EventArgs) Handles btnIncrement.Click
   PerformanceCounter1.ReadOnly = False
   PerformanceCounter1.IncrementBy(CLng(txtValue.Text))
End Sub

Private Sub btnDecrement_Click(ByVal sender As System.Object, _
   ByVal e As System.EventArgs) Handles btnDecrement.Click
   PerformanceCounter1.ReadOnly = False
   PerformanceCounter1.Decrement()
End Sub

Private Sub btnSetValue_Click(ByVal sender As System.Object, _
   ByVal e As System.EventArgs) Handles btnSetValue.Click
   PerformanceCounter1.ReadOnly = False
   PerformanceCounter1.RawValue = CLng(txtValue.Text)
End Sub

// C#
```

```
protected void btnIncrement_Click(System.Object sender,
   System.EventArgs e)
{
   performanceCounter1.ReadOnly = false;
   performanceCounter1.IncrementBy(long.Parse(txtValue.Text));
}

protected void btnDecrement_Click(System.Object sender,
   System.EventArgs e)
{
   performanceCounter1.ReadOnly = false;
   performanceCounter1.Decrement();
}

protected void btnSetValue_Click(System.Object sender,
   System.EventArgs e)
{
   performanceCounter1.ReadOnly = false;
   performanceCounter1.RawValue = long.Parse(txtValue.Text);
}
```

See Also

Introduction to Monitoring Performance Thresholds | Performance Counter Value Retrieval

Performance Counter Value Retrieval

There are several ways you can read performance counter values:

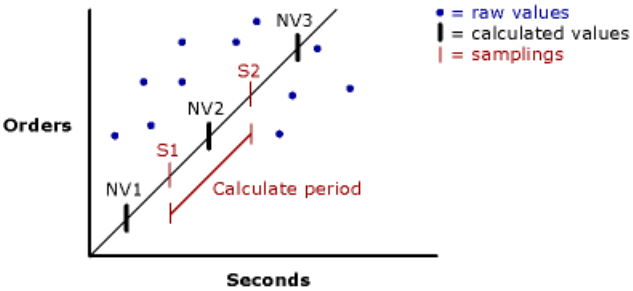
- You can retrieve a raw value from a counter using the RawValue property on the **PerformanceCounter** class.
- You can retrieve the current calculated value for a counter using the NextValue method on the **PerformanceCounter** class.
- You can retrieve a set of samples using the NextSample method on the **PerformanceCounter** class, and compare their values using the CounterSample.Calculate method.

> **Note** There are security restrictions that affect your ability to use performance counters. For more information, see Introduction to Monitoring Performance Thresholds.

Raw, Calculated, and Sampled Data

The interaction between raw values, next (or calculated) values, and samples is fairly straightforward once you understand that raw and calculated values shift constantly, whereas samples allow you to retrieve a static snapshot of the counter at a particular point in time. The following diagram illustrates the relationship between raw value, next value, and samples.

Performance Counter Values: Raw, Calculated, and Sampled



This picture shows a representation of the data contained in a counter called Orders Per Second. The raw values for this counter are individual data points that vary by second, where the calculated average is represented by the line showing an increasing order receipt over time. In this chart, the following data points have been taken:

- The user has used the **NextValue** method to retrieve the calculated value at three different times, represented by NV1, NV2, and NV3. Because the next value is constantly changing, a different value is retrieved each time without specifying any additional parameters.
- The user has used the **NextSample** method to take two samples, indicated by S1 and S2. Samples freeze a value in time, so the user can then compare the two sample values and perform calculations on them.

Comparing Retrieval Methods

Retrieving a raw value with the **RawValue** property is very quick, because no calculations or comparisons are performed. For example, if you are using a counter simply to count the number of orders processed in a system, you can retrieve the counter's raw value. For more information on retrieving raw values, see Retrieving Raw Performance Counter Values.

Retrieving a calculated value with the **NextValue** method is often more useful than retrieving the raw value, but this value may also give you an unrealistic view of the data because it can reflect unusual fluctuations in the data at the moment when the value is calculated. For example, if you have a counter that calculates the orders processed per second, an unusually high or low amount of orders processed at one particular moment will result in an average that is not realistic over time. This may provide a distorted view of the actual performance of your system. For more information on retrieving a calculated value, see Retrieving Calculated Performance Counter Values.

Samples provide the most realistic views of the data in your system by allowing you to retrieve, retain, and compare various values over time. You would retrieve a sample, using the **NextSample** method, if you needed to compare values in different counters or calculate a value based on raw data. This may be slightly more resource-intensive, however, than a **NextValue** call. For more information on retrieving a sample, see Retrieving Performance Counter Samples.

The **NextSample** method returns an object of type CounterSample. When you retrieve a sample, you have access to properties on the **CounterSample** class such as **RawValue**, **BaseValue**, **TimeStamp**, and **SystemFrequency**. These properties let you get a very detailed look at the data that makes up the sample data.

See Also

Retrieving Raw Performance Counter Values | Retrieving Calculated Performance Counter Values | Retrieving Performance Counter Samples | Retrieving Lists of Counters and Categories | Introduction to Monitoring Performance Thresholds

Retrieving Raw Performance Counter Values

You retrieve raw performance counter values by retrieving the current value of the **RawValue** property on the **PerformanceCounter** class.

To retrieve a counter's raw value

1. Create a **PerformanceCounter** instance and configure it to interact with the desired category and counter. For more information, see Creating PerformanceCounter Component Instances or Configuring PerformanceCounter Component Instances.
2. Retrieve the current value of the **RawValue** property.

   The following example illustrates how to use the **RawValue** property to retrieve the current value of the % Processor Time counter:

```
' Visual Basic
Dim MyCounter As New PerformanceCounter()
MyCounter.CategoryName = "Processor"
MyCounter.CounterName = "% Processor Time"
MyCounter.InstanceName = "_Total"
Dim raw As Int64
raw = MyCounter.RawValue

// C#
System.Diagnostics.PerformanceCounter myCounter =
   new System.Diagnostics.PerformanceCounter();
myCounter.CategoryName = "Processor";
myCounter.CounterName = "% Processor Time";
myCounter.InstanceName = "_Total";
long raw = myCounter.RawValue;
```

See Also

Performance Counter Value Retrieval | Retrieving Calculated Performance Counter Values | Retrieving Performance Counter Samples | Retrieving Lists of Counters and Categories

Retrieving Calculated Performance Counter Values

You retrieve calculated performance counter values by calling the NextValue method on the current counter. The counter's calculated value at the instant that the method executes is calculated and returned to you. Calling **NextValue** a second time will retrieve a different value, because the calculated value changes as the counter increments.

> **Note** To retrieve more complex views of the data in the counter, you can use samples to retrieve static snapshots of counter data at two points in time, and perform comparisons on those returns. For more information on retrieving a sample, see Retrieving Performance Counter Samples.

To retrieve a counter's calculated value

1. Create a **PerformanceCounter** instance and configure it to interact with the desired category and counter. For more information, see Creating PerformanceCounter Component Instances or Configuring PerformanceCounter Component Instances.
2. Call the **NextValue** method and set the results to an integer variable.

   The following example illustrates how to use the **NextValue** property to retrieve the current value of the Total counter in a custom category called Orders:

```
' Visual Basic
Dim MyCtr As New PerformanceCounter()
MyCtr.CategoryName = "Orders"
MyCtr.CounterName = "Total"
Dim retvalue As Single
retvalue = MyCtr.NextValue()

// C#
System.Diagnostics.PerformanceCounter MyCtr =
   new System.Diagnostics.PerformanceCounter();
MyCtr.CategoryName= "Orders";
MyCtr.CounterName= "Total";
float retvalue = MyCtr.NextValue();
```

> **Note** The first time you run this code, it will return zero. Subsequent queries for the value will return a figure.

See Also

Performance Counter Value Retrieval | Retrieving Performance Counter Samples | Creating PerformanceCounter Component Instances | Configuring PerformanceCounter Component Instances

Retrieving Performance Counter Samples

You use the CounterSample class to create samples and perform calculations on their contents. A sample class performs a "sampling" of a performance counter based on criteria you define. These criteria include the values from one or more counters and the frequency with which values should be provided. In addition, the class records the time at which samples were taken. You can collect all of this data in a single instantiated class and then use the Calculate method to perform a calculation. You can also use the **Calculate** method to perform a calculation that compares values in two different samples.

The calculation performed depends on the type of the counters; certain counter types have specific calculations associated with them. For example, counters of the type **ElapsedTime** compare the time stamp on two different samples and determine how much time has elapsed. Many counters perform an averaging calculation based on the data retrieved.

These are the steps you go through to define a sample:

- Create one or more instances of the **CounterSample** class.
- Retrieve a current sample for each instance.
- Call the **Calculate** method, passing each sample to be included in the calculation as a parameter.

To retrieve performance counter samples and perform a calculation

1. Create a **PerformanceCounter** instance and configure it to interact with the desired category and counter. For more information, see Creating PerformanceCounter Component Instances or Configuring PerformanceCounter Component Instances.
2. Create an instance of the **CounterSample** class to hold the results of the sampling.
3. Call the **NextSample** method on the **PerformanceCounter** component instance to retrieve the calculated value, and assign the result to the **CounterSample** class.

   Tip You must retrieve two samples to perform a calculation.

4. Create a variable to hold the results of the calculation, and give it a data type of Single .
5. For each of the two samples that you need for your calculation, assign a return value of **NextSample** to a variable of type **CounterSample**.
6. Call the **Calculate** method on the **CounterSample** class, and do one of two things:

   - If you have retrieved two samples, pass both of the samples (which are stored as **CounterSample** objects) to the **Calculate** method as parameters.

     - or -

   - If you have retrieved only one sample, pass the first sample to the **Calculate** method, and then use the second parameter to retrieve another sample.

7. Set the result of the calculation equal to the variable you created to contain the result.

   Note The two samples must be from counters of the same type or the method will throw an exception. The type of counter determines what kind of calculation performed. For more information, see Performance Counter Types.

The following code illustrates how to retrieve two samples and compare them using the **Calculate** method:

```
' Visual Basic
' Dim variables of types CounterSample for each sample and a
' variable of type single for the results.
Dim sample1 As CounterSample
Dim sample2 As CounterSample
Dim result As Single
' Retrieve a sample.
sample1 = PerformanceCounter1.NextSample()
' Wait some interval of time here and retrieve
' a second sample.
System.Threading.Thread.Sleep(1000)
sample2 = PerformanceCounter1.NextSample()
```

```
' Pass both samples to the Calculate method.
result = CounterSample.Calculate(sample1, sample2)

// C#
System.Diagnostics.CounterSample sample1;
System.Diagnostics.CounterSample sample2;
float result;
// Retrieve a sample.
sample1 = PerformanceCounter1.NextSample();
// Wait some interval of time here and retrieve
// a second sample.
System.Threading.Thread.Sleep(1000);
sample2 = PerformanceCounter1.NextSample();
// Pass both samples to the Calculate method.
result = System.Diagnostics.CounterSample.Calculate(sample1, sample2);
```

Alternatively, you can call the **NextSample** method to provide the values for the second sample. The following example illustrates using this approach:

```
' Visual Basic
Dim sample1 As CounterSample
Dim result As Single
' Retrieve a single sample.
sample1 = PerformanceCounter1.NextSample()
' Pass the retrieved sample to the calculate method
' and retrieve another sample in the second parameter.
result = CounterSample.Calculate(sample1, PerformanceCounter1.NextSample())

// C#
System.Diagnostics.CounterSample sample1;
float result;
// Retrieve a single sample.
sample1 = PerformanceCounter1.NextSample();
// Pass the retrieved sample to the calculate method
// and retrieve another sample in the second parameter.
result = System.Diagnostics.CounterSample.Calculate(sample1, PerformanceCounter1.NextSamp
le());
```

See Also

Performance Counter Value Retrieval | Retrieving Raw Performance Counter Values | Retrieving Calculated Performance Counter Values | Retrieving Lists of Counters and Categories

Retrieving Lists of Counters and Categories

You can use the GetCounters method on the PerformanceCounterCategory class to quickly retrieve a list of all of the counters or instances in a specific category. Each time you request a value from a category by creating an instance of a **PerformanceCounter** object, setting it to an existing counter, and retrieving a value, the system reads through the entire category and retrieves the requested counter. This means that if you request five different counters from a category that contains 20 counters, the system reads all 20 counters in the category five times, searching for each counter separately. Using **GetCounters**, you get the same data, but the category is read only once.

The **GetCounters** method returns an array of type **PerformanceCounter** that contains the counters in the category. You can use the **Item** property to work with the contents of the collection.

In addition to retrieving counters, you can also use the static GetCategories method to return a list of categories on the current computer or any server to which you have access. **GetCategories** returns an array of **PerformanceCounterCategories** objects.

To retrieve the counters in a category

1. Create a **PerformanceCounter** object and configure it so that it points to the desired category. For more information, see Creating PerformanceCounter Component Instances or Configuring PerformanceCounter Component Instances.
2. Create an array of type **PerformanceCounter** to contain the resulting category list.
3. Call the **GetCounters** method on the **PerformanceCounterCategory** class and specify the desired category as a parameter.
4. Set the results to the array.

   The following example shows how to retrieve all of the counters in the **Cache** category. This code assumes that you are working in a Windows Form that contains a button and a **ListBox** control. It also assumes that you have a reference to the **Systems.Diagnostics** namespace and an **imports** statement (for Visual Basic) or a **using** statement (for C#) to the same:

```
' Visual Basic
Private Sub btnGetCounters_Click(ByVal sender As System.Object, _
   ByVal e As System.EventArgs) Handles btnGetCounters.Click
   Dim mypc() As PerformanceCounter
   Dim i As Integer
   Dim myCat As New PerformanceCounterCategory("Cache")
   ' Remove the current contents of the list.
   Me.ListBox1.Items.Clear()
   ' Retrieve the counters.
   mypc = myCat.GetCounters
   ' Add the retrieved counters to the list.
   For i = 0 To mypc.Length - 1
      Me.ListBox1.Items.Add(mypc(i).CounterName)
   Next
End Sub

// C#
private void btnGetCounters_Click(System.Object sender, System.EventArgs e)
{
   System.Diagnostics.PerformanceCounter[] mypc;
   System.Diagnostics.PerformanceCounterCategory mycat =
      new System.Diagnostics.PerformanceCounterCategory("cache");
   // Remove the current contents of the list.
   this.listBox1.Items.Clear();
   // Retrieve the counters.
   mypc = mycat.GetCounters();
   // Add the retrieved counters to the list.
   for (int i = 0; i < mypc.Length; i++)
   {
      this.listBox1.Items.Add(mypc[i].CounterName);
   }
}
```

To retrieve all categories on a machine

1. Create an array of type **PerformanceCounter** to contain the resulting category list.
2. Call the **GetCategories** method on the **PerformanceCounterCategory** class and specify the desired category as a parameter.
3. Set the results to the array.

   The following code shows how to retrieve all categories on the local machine. This code assumes that you are working in a Windows form that contains a button and a **ListBox** control. It also assumes that you have a reference to the **Systems.Diagnostics** namespace and an imports statement (for Visual Basic) or a using statement (for C#) to the same:

```
' Visual Basic
Private Sub btnGetCategories_Click(ByVal sender As System.Object, _
ByVal e As System.EventArgs)
   Dim myCat2 As PerformanceCounterCategory()
   Dim i As Integer
   ' Clear the contents of the list box.
   Me.listBox2.Items.Clear()
   ' Retrieve the categories.
   myCat2 = PerformanceCounterCategory.GetCategories
   ' Add the retrieved categories to the list.
   For i = 0 To myCat2.Length - 1
      Me.listBox2.Items.Add(myCat2(i).CategoryName)
   Next
End Sub

// C#
private void btnGetCategories_Click(object sender, System.EventArgs e)
{
   System.Diagnostics.PerformanceCounterCategory[] myCat2;
   // Clear the list's current contents.
   this.listBox2.Items.Clear();
   // Retrieve the categories.
   myCat2 =
      System.Diagnostics.PerformanceCounterCategory.GetCategories();
   // Add the retrieved categories to the list.
   for (int i = 0; i < myCat2.Length; i++)
   {
      this.listBox2.Items.Add(myCat2[i].CategoryName);
   }
}
```

See Also

Performance Counter Value Retrieval | Creating PerformanceCounter Component Instances | Configuring PerformanceCounter Component Instances

Working with Performance Counters in Server Explorer

Using Server Explorer, you can explore existing performance counters on any computer to which you have access. You can view any counter available on these computers, including those you create in Visual Studio .NET.

> **Note** You can also see information about performance counters in the Windows 2000 Performance window.

> **Note** The Servers node of the Server Explorer is not available in the Standard Edition of Visual Basic and Visual C# .NET. For more information, see Visual Basic Standard Edition Features or Visual C# Standard Edition Features.

The Performance Counters node in Server Explorer lists the categories that exist on each server. You can expand each counter to see the instances it represents. Note that instances apply to the category rather than to individual counters, so each counter within a category will have the same instances.

To locate a performance counter in Server Explorer

1. Expand the **Server** node, and then expand the node for the server you want to view.

   > **Tip** If the server you want to view is not listed, you need to add it. For more information, see Accessing and Initializing Server Explorer.

2. Locate and expand the **Performance Counters** node beneath it.

To locate a performance counter in Windows

1. Access the **Control Panel** for your computer.
2. Double-click **Administrative Tools**, and then double-click **Performance**. The **Performance** console window appears.
3. To add a counter to the display, right-click the graph area in the window and select **Add Counters**.

See Also

Accessing and Initializing Server Explorer

Performance Counter Walkthroughs

The walkthroughs in this section introduce you to the use of the **PerformanceCounter** component. The walkthroughs show you how to use both system performance counters and custom performance counters.

In This Section

Walkthrough: Retrieving Categories and Counters
Presents step-by-step instructions on how to retrieve and work with Windows performance counter categories and counters, using the **PerformanceCounter** component.

Walkthrough: Changing and Retrieving Performance Counter Values
Presents step-by-step instructions on how to retrieve raw and calculated values from a Windows performance counter using the **PerformanceCounter** component.

Related Sections

Component Programming Walkthroughs
Lists all the walkthroughs related to component authoring and programming in Visual Basic and C#.

System Monitoring Walkthroughs
Lists walkthroughs you can use to learn how to work with **EventLog**, **PerformanceCounter**, and **ServiceController** components.

Walkthroughs in Visual Basic and Visual C#
Lists all the step-by-step topics that use Visual Basic and Visual C#.

Visual Basic Code Example Topics
Shows quickstart-like code examples that help you perform common tasks in Visual Basic .NET.

Walkthrough: Retrieving Categories and Counters

The procedures in this walkthrough take you through the process of creating and configuring **PerformanceCounter** component instances and using them to retrieve lists of performance counter categories and counters on your system. A performance counter is the means by which Windows collects performance data on various system resources. Windows contains a set of predefined counters, organized into categories, with which you can interact. Each category and counter is related to a specific area of system functionality.

In this walkthrough you will:

- Instantiate a **PerformanceCounter** component and configure it to interact with a specific category of system-generated counters.
- Create a Windows Application that displays information about categories and counters in a list box.
- Use the **GetCategories** method to return a list of categories on the local computer.
- Use the **GetCounters** method to return a list of counters from the specified category.

To create your Windows Application

1. From the **New Project** dialog box, create a Visual Basic or C# Windows Application.
2. From the **Windows Forms** tab of the Toolbox, add two buttons and two list boxes to your form. Arrange them in any order you like, and then set the following properties for them:

Control	Property	Value
Button1	Name	**btnGetCounters**
	Text	**Get Counters**
Button2	Name	**btnGetCategories**
	Text	**Get Categories**
ListBox1	Name	**lstCounters**
	ScrollAlwaysVisible	**True**
ListBox2	Name	**lstCategories**
	ScrollAlwaysVisible	**True**

3. Save your work.

To retrieve the category list

1. In Design view, double-click the **Get Categories** button to access the Code Editor. Your cursor will be positioned in the **btnGetCategories_Click** event.
2. In the **btnGetCategories_Click** procedure, add the following code to retrieve the list of categories from the local computer.

```
' Visual Basic
Private Sub btnGetCategories_Click(ByVal sender As System.Object, _
   ByVal e As System.EventArgs) Handles btnGetCategories.Click

   Dim myCat2 As PerformanceCounterCategory()
   Dim i As Integer
   ' Remove the current contents of the list.
   Me.lstCategories.Items.Clear()
   ' Retrieve the categories.
   myCat2 = PerformanceCounterCategory.GetCategories
   ' Add the retrieved categories to the list.
   For i = 0 To myCat2.Length - 1
      Me.lstCategories.Items.Add(myCat2(i).CategoryName)
   Next
End Sub

// C#
private void btnGetCategories_Click(object sender, System.EventArgs e)
{
   System.Diagnostics.PerformanceCounterCategory[] myCat2;
   // Remove the current contents of the list.
```

```
   this.lstCategories.Items.Clear();
   // Retrieve the categories.
   myCat2 =
      System.Diagnostics.PerformanceCounterCategory.GetCategories();
   // Add the retrieved categories to the list.
   for (int i = 0; i < myCat2.Length; i++)
   {
      this.lstCategories.Items.Add(myCat2[i].CategoryName);
   }
}
```

To retrieve the counter list

1. In Design view, double-click the **Get Counters** button to access the Code Editor. The insertion point will be positioned in the **btnGetCounters_Click** event.
2. Add the following code to retrieve the list of counters from the selected category.

```
' Visual Basic
Private Sub btnGetCounters_Click(ByVal sender As System.Object, _
   ByVal e As System.EventArgs) Handles btnGetCounters.Click

   Dim instanceNames() As String
   Dim counters As New System.Collections.ArrayList()

   If (Me.lstCategories.SelectedIndex <> -1) Then
      Dim mycat As New PerformanceCounterCategory( _
         Me.lstCategories.SelectedItem.ToString())
      ' Remove the current contents of the list.
      Me.lstCounters.Items.Clear()
      ' Retrieve the counters.
      Try
         instanceNames = mycat.GetInstanceNames()
         If (instanceNames.Length = 0) Then
            counters.AddRange(mycat.GetCounters())
         Else
            Dim i As Integer
            For i = 0 To instanceNames.Length - 1
               counters.AddRange( _
                  mycat.GetCounters(instanceNames(i)))
            Next
         End If
         ' Add the retrieved counters to the list.
         Dim counter As PerformanceCounter
         For Each counter In counters
            Me.lstCounters.Items.Add(counter.CounterName)
         Next
      Catch ex As Exception
         MessageBox.Show( _
            "Unable to list the counters for this category:" _
            & ControlChars.CrLf & ex.Message)
      End Try
   End If
End Sub

// C#
private void btnGetCounters_Click(object sender, System.EventArgs e)
{
   string[] instanceNames;
   System.Collections.ArrayList counters = new
```

```
        System.Collections.ArrayList();
    if (this.lstCategories.SelectedIndex != -1)
    {
        System.Diagnostics.PerformanceCounterCategory mycat = new
            System.Diagnostics.PerformanceCounterCategory(
            this.lstCategories.SelectedItem.ToString());
        // Remove the current contents of the list.
        this.lstCounters.Items.Clear();
        // Retrieve the counters.
        try
        {
            instanceNames = mycat.GetInstanceNames();
            if (instanceNames.Length == 0)
            {
                counters.AddRange(mycat.GetCounters());
            }
            else
            {
                for (int i = 0; i < instanceNames.Length; i++)
                {
                    counters.AddRange(mycat.GetCounters(instanceNames[i]));
                }
            }

            // Add the retrieved counters to the list.
            foreach (System.Diagnostics.PerformanceCounter counter
                in counters)
            {
                this.lstCounters.Items.Add(counter.CounterName);
            }
        }
        catch (System.Exception ex)
        {
            MessageBox.Show(
                "Unable to list the counters for this category:\n"
                + ex.Message);
        }
    }
}
```

To test your application

1. Save and compile your application.
2. Click the **Get Categories** button. You will see a list of categories in the list box.
3. Select one of the categories and click the **Get Counters** button. You will see a list of the counters for the selected category.

See Also

Monitoring Performance Thresholds | Introduction to Monitoring Performance Thresholds | Performance Counter Walkthroughs

Walkthrough: Changing and Retrieving Performance Counter Values

> **Note** The Servers node of Server Explorer used in this walkthrough is not available in the Standard Edition of Visual Basic and Visual C# .NET. For more information, see Visual Basic Standard Edition Features or Visual C# Standard Edition Features.

The procedures in this walkthrough show you how to work with the values of a performance counter using the methods on the **PerformanceCounter** class. A performance counter is the means by which Windows collects performance data on various system resources. Windows contains a set of pre-defined counters, organized into categories, with which you can interact. The values you retrieve from a counter can either be raw values or calculated values that alter over time. There are several ways to increase and decrease the current value of a counter.

In the course of this walkthrough, you will:

- Create and configure a **PerformanceCounter** component to interact with a system-provided counter.
- Create a Windows application that contains UI elements that retrieve and display values from a counter.
- Write code using the **RawValue** method to both set and retrieve the raw value of the counter.
- Write code using the **Increment**, **IncrementBy**, and **Decrement** methods to alter the value of the counter and retrieve its new value.

To create your Windows Application

1. On the **File** menu, point to **New**, then click **Project**.
2. In the **New Project** dialog box, choose Visual Basic or Visual C# in the left pane, then choose the **Windows Application** template. Name the project **PerformanceCounterExample**.
3. From the **Windows Forms** tab of the Toolbox, add the following controls to your application:
   - Two labels
   - A text box
   - Five buttons
4. Set the following properties on your controls:

Control	Property	Value
Label1	Name	lblCounterValue
	Text	(blank)
Label2	Name	lblSystemCounterValue
	Text	(blank)
Textbox1	Name	txtValue
	Text	(blank)
Button1	Name	btnSetRawValue
	Text	Set Raw Value of the Custom Counter
Button2	Name	btnGetNextValue
	Text	Get Next Value of System Counter
Button3	Name	btnIncrement
	Text	Increase the Custom Counter by 1
Button4	Name	btnDecrement
	Text	Decrease the Custom Counter by 1
Button5	Name	btnIncrementBy
	Text	Increase Custom Counter by a value

5. Arrange the controls as you like.
6. Save your work.

To create and configure your PerformanceCounter component

1. Open Server Explorer and access the **Servers** node. For more information, see Accessing and Initializing Server Explorer.
2. Locate the listing for your computer under the **Servers** node and expand it. You will see entries for performance counters,

messages queues, event logs, and services.

3. Expand the **Performance Counters** node and locate the **Processor** node.
4. Locate the **% Processor Time** node and expand it.
5. Drag the **_Total** counter onto the form. A **PerformanceCounter** component configured for the **_Total** counter is added to your project. The component is named **PerformanceCounter1** in Visual Basic, and **performanceCounter1** in Visual C#.

To retrieve the raw value of the system counter

1. In the designer, double-click the **Get Next Value** button to create the **Click** event handler. This method will report the calculated value of the counter, not the raw value. Add the following code to retrieve and display the next value of the counter your **PerformanceCounter** instance is watching:

```
' Visual Basic
Private Sub btnGetNextValue_Click(ByVal sender As System.Object, _
ByVal e As System.EventArgs) Handles btnGetNextValue.Click
   lblSystemCounterValue.Text = _
      "The current value of the system counter is: " _
      & PerformanceCounter1.NextValue().ToString()
End Sub

// C#
private void btnGetNextValue_Click(object sender, System.EventArgs e)
{
   lblSystemCounterValue.Text =
      "The current value of the system counter is: "
      + performanceCounter1.NextValue().ToString();
}
```

For the rest of your procedures, you will work with a custom category and counter.

To create a custom counter and category

1. On the **View** menu, click **Designer** to access the designer for Form1.
2. In the designer, open Server Explorer and access the **Servers** node. For more information, see Accessing and Initializing Server Explorer.
3. Locate the listing for your computer under the **Servers** node and expand it.
4. Right-click the **Performance Counters** node and select **Create New Category** from the shortcut menu.

   The **Performance Counter Builder** dialog box appears.

5. Enter **MyNewCategory** for the **Category name**.
6. Click **New** to add a new counter, and name it **MyNewCounter**. Click **OK** to create the new category and counter.

   > **Note** You will need permission to write to the registry for this step. Contact your system administrator if you are unable to create the new counter.

7. In Server Explorer, browse to the new counter and drag it onto the form. A new **PerformanceCounter** component is added to the project, configured for the new counter, **MyNewCounter**.

   > **Note** These steps create the counter on your computer. If you were to deploy this application, you would need to create the counter on the target computer. You can do this by adding an Installer to the project. Select the **PerformanceCounter** object, and click **Add Installer** in the Properties window. For more information, see Introduction to Installation Components.

To set the raw value of the custom counter

1. In the designer, select the **PerformanceCounter2** in Visual Basic, or **performanceCounter2** in Visual C#.
2. Set the **ReadOnly** property to **false**.
3. Double-click the **btnSetRawValue** button to create the **Click** event handler in the Code Editor.
4. Add the following code to set the raw value of the counter you created and to display the counter's value in the first of the label controls.

```
' Visual Basic
Private Sub btnSetRawValue_Click(ByVal sender As System.Object, _
ByVal e As System.EventArgs) Handles btnSetRawValue.Click
   PerformanceCounter2.RawValue = CLng(txtValue.Text)
   lblCounterValue.Text = PerformanceCounter2.NextValue().ToString()
End Sub

// C#
private void btnSetRawValue_Click(object sender, System.EventArgs e)
{
   performanceCounter2.RawValue = long.Parse(txtValue.Text);
   lblCounterValue.Text = performanceCounter2.NextValue().ToString();
}
```

To increase the custom counter value by one and display it

1. In the designer, double-click the **Increase the Custom Counter by 1** button to create the **Click** event handler in the Code Editor.
2. Add the following code to add one to your custom counter and to display the counter's value in the first of the label controls.

```
' Visual Basic
Private Sub btnIncrement_Click(ByVal sender As System.Object, _
ByVal e As System.EventArgs) Handles btnIncrement.Click
   PerformanceCounter2.Increment()
   lblCounterValue.Text = PerformanceCounter2.NextValue().ToString()
End Sub

// C#
private void btnIncrement_Click(object sender, System.EventArgs e)
{
   performanceCounter2.Increment();
   lblCounterValue.Text = performanceCounter2.NextValue().ToString();
}
```

To decrease the custom counter and display its value

1. In the designer, double-click the **Decrease the Custom Counter by 1** button to create the **Click** event handler in the Code Editor.
2. Add the following code to decrease the counter value by one and to display the counter's value in the first of the label controls.

```
' Visual Basic
Private Sub btnDecrement_Click(ByVal sender As System.Object, _
ByVal e As System.EventArgs) Handles btnDecrement.Click
   PerformanceCounter2.Decrement()
   lblCounterValue.Text = PerformanceCounter2.NextValue().ToString()
End Sub

// C#
private void btnDecrement_Click(object sender, System.EventArgs e)
{
   performanceCounter2.Decrement();
   lblCounterValue.Text = performanceCounter2.NextValue().ToString();
}
```

To increase the custom counter by a user-defined value

1. In the designer, double-click the **Increase Custom Counter by a value** button to create the **Click** event handler in the

Code Editor.

2. Add the following code to increase the counter value by the value entered in the text box and to display the counter's value in the first of the label controls.

```
' Visual Basic
Private Sub btnIncrementBy_Click(ByVal sender As System.Object, _
ByVal e As System.EventArgs) Handles btnIncrementBy.Click
   PerformanceCounter2.IncrementBy(CLng(txtValue.Text))
   lblCounterValue.Text = PerformanceCounter2.NextValue().ToString()
End Sub

// C#
private void btnIncrementBy_Click(object sender, System.EventArgs e)
{
   performanceCounter2.IncrementBy(long.Parse(txtValue.Text));
   lblCounterValue.Text = performanceCounter2.NextValue().ToString();
}
```

To test your application

1. Save your files.
2. Run the application.
3. Click the **Get Next Value of System Counter** button to retrieve the current value of the Processor category's counter. Since you are retrieving the value using the NextValue() method, the first call will return 0.

   You should see the current value displayed in the label.

4. Type **25** in the text box and click **Set Raw Value of Custom Counter**.

   The label field should be updated to indicate that the raw value is now 25.

5. Click the **Increase Custom Counter by 1** button.

   The value in the label should increase by 1.

6. Click the **Decrease Custom Counter by 1** button.

   The value in the label should decrease by 1.

7. Enter **25** in the text box and click **Increase Custom Counter by a value**.

   The value in the label should increase by 25.

   You can also view your performance counter in the Windows Performance tool.

To view your performance counter in the Windows Performance tool

1. On the taskbar, click the **Start** button, point to **Settings**, and then click **Control Panel**.
2. In **Control Panel**, double-click **Administrative Tools**, and then double-click **Performance**.
3. Right-click the counter list below the performance graph, and select **Add Counters** from the shortcut menu.
4. Select **MyNewCategory** from the **Performance object** list and **MyNewCounter** from the counter list. Click **Add** to finish.
5. Click **View Report** from the toolbar to display the value of your counter.

See Also

Monitoring Performance Thresholds | Introduction to Monitoring Performance Thresholds | Performance Counter Walkthroughs

Logging Application, Server, and Security Events

Windows event logs allow your applications and components to record information about important events. You can use these records to audit access to your system, troubleshoot problems, and re-create usage patterns.

Using the **EventLog** component and language features in Visual Studio .NET or the .NET Framework, you can easily connect to Windows event logs on both local and remote computers and write entries to these logs. You can also read entries from existing logs and create your own custom event logs on computers that have Windows 2000 or Windows NT installed.

In This Section

Introduction to the EventLog Component
Provides basic information on event log features.
Administering Event Logs
Suggests additional topics on deleting event logs and performing administrative tasks on event logs.
Creating EventLog Component Instances
Gives directions for creating and configuring **EventLog** component instances in your projects.
Writing Entries to Event Logs
Gives directions for configuring your component to write entries to event logs.
Reading Event Log Entries
Gives directions for reading event log entries.
Handling the EntryWritten Event
Describes how to create event handlers for your **EventLog** components that automatically call a procedure when an entry is written to a log.
Working with Event Logs in Server Explorer
Gives directions for using Server Explorer to investigate the event logs on a computer.
Security Ramifications of Event Logs
Describes the effects of event log interactions on application security.
Walkthrough: Exploring Event Logs, Event Sources, and Entries
Presents step by step instructions on how to use the basic features of the **EventLog** component and language features.

Related Sections

EventLog Programming Architecture
Describes the language elements used in **EventLog** component programming.
Programming with Components
Presents a high-level overview of component programming and authoring.
EventLog Members
Describes the major features of the **EventLog** class.
Creating Installation Components
Describes how to use the Add Installer link in the Properties window to add an installation component to applications with **EventLog** component instances.
Visual Basic and Visual C# Projects
Describes the projects types used in this chapter and how to choose between them.
Deploying Applications and Components
Lists the major deployment topics and the information they contain.
Manageability
Discusses best practices for managing distributed applications.

Introduction to the EventLog Component

Event logging in Microsoft Windows provides a standard, centralized way for you to have your applications record important software and hardware events. When an error occurs, the system administrator or support technicians must determine what caused the error, attempt to recover any lost data, and prevent the error from recurring. It is helpful if applications, the operating system, and other system services record important events such as low-memory conditions or failed attempts to access a disk. The system administrator can use the event log to help determine what conditions caused the error and the context in which it occurred.

Windows supplies a standard user interface for viewing these event logs and a programming interface for examining log entries. In Microsoft Visual Basic version 6.0, you could perform limited write operations to some event logs, but you could not easily read or interact with all the logs available to you.

Using the **EventLog** component, you can easily connect to event logs on both local and remote computers and write entries to these logs. You can also read entries from existing logs and create your own custom event logs

In addition to writing to event logs and reading their entries, you can use Server Explorer to view a list of the event logs for any server to which you have access. You can expand the **Event Logs** node in Server Explorer to see a list of the logs that currently exist and the events that have been written to them. You can also drag a specific event log from Server Explorer to your designer to create an instance of the **EventLog** component that automatically has its properties set to point to the log you selected.

> **Note** The **Servers** node of Server Explorer is not available in the Standard Edition of Visual Basic and Visual C# .NET. For more information, see Visual Basic Standard Edition Features or Visual C# Standard Edition Features.
>
> **Note** You can also use the **GetEventLogs** method to retrieve a list of all event logs on a machine.
>
> **Note** There are security restrictions that affect your ability to use event logs. For more information, see Security Ramifications of Event Logs.

Key Concepts in Windows Event Logs

An *event,* as defined in Windows, is any significant occurrence — whether in the operating system or in an application — that requires users to be notified. Critical events are sent to the user in the form of an immediate message on the screen. Other event notifications are written to one of several *event logs* that record the information for future reference. Every event log entry is classified into one of the following categories: errors, warnings, information, success audits, or failure audits.

There are three event logs available by default on computers running Windows 2000 or Windows NT 4.0:

- a *System log*, which tracks events that occur on system components (for example, a problem with a driver);
- a *Security log*, which tracks security changes and possible breaches;
- an *Application log*, which tracks events that occur in a registered application.

In addition to these logs, other programs such as Active Directory may create their own default logs. You can also create your own custom logs using the language features in the **System.Diagnostics** namespace.

Choosing Events to Log

The following are examples of cases in which event logging can be helpful:

- Resource problems. If an application gets into a low-memory situation (caused by a code bug or inadequate memory) that degrades performance, logging a warning event when memory allocation fails might provide a clue about what went wrong.
- Information events. A server-based application (such as a database server) records a user logging on, opening a database, or starting a file transfer. The server can also log error events it encounters (cannot access file, host process disconnected, and so on), corruptions in the database, or whether a file transfer was successful.

Event logging consumes resources such as disk space and processor time. The amount of disk space that an event log requires and the overhead for an application that logs events depend on how much information you choose to log. Therefore, it is important to log only essential information. It is also best to place event log calls in an error path in the code rather than in the main code path, so as not to reduce performance.

See Also

Types of Event Log Entries | Event Log References | EventLog Programming Architecture |

Walkthrough: Exploring Event Logs, Event Sources, and Entries

Types of Event Log Entries

When you write an entry to a log, you specify the type of entry you're making. The following table lists the types of events and their contents.

Entry Type	Contents
Information	An information event indicates an infrequent but significant successful operation. For example, when Microsoft SQL Server successfully loads, it may be appropriate to log an information event stating that "SQL Server has started." Note that while this is appropriate behavior for major server services, it is generally inappropriate for a desktop application to log an event each time it starts.
Warning	A warning event indicates a problem that is not immediately significant but that may cause future complications. Resource consumption is a good candidate for a warning event. For example, an application can log a warning event if disk space is low. If an application can recover from an event without loss of functionality or data, it can generally classify the event as a warning event.
Error	An error event indicates a significant problem the user should know about, such as a loss of functionality or data. For example, if a service cannot be loaded as the system boots, it can log an error event.
Success audit	A success audit event is a security event that occurs when an audited access attempt is successful. For example, a successful logon attempt is a success audit event.
Failure audit	A failure audit event is a security event that occurs when an audited access attempt fails. For example, a failed attempt to open a file is a failure audit event.

By default, the event type is set to **Information** if you do not specify otherwise. You can set the type by using a parameter on an overloaded form of the WriteEntry method. When you are reading log entries, you can query the **EntryType** property on the **EventLogEntry** class to see what type of entry you are working with.

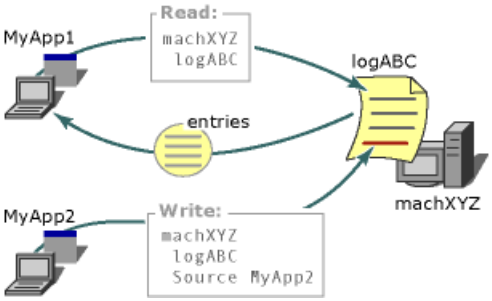
See Also

Introduction to the EventLog Component | Event Log References | EventLog Programming Architecture | Walkthrough: Exploring Event Logs, Event Sources, and Entries

Event Log References

There are three default event logs on any computer: an Application event log, a Security event log, and a System event log. In addition to these logs, there may be other logs that are installed by applications on the server and custom logs that are created by users. You identify which of these event logs you want to work with. The method of referencing a specific log differs depending on whether you want to read the log's entries or write entries to the log. The following illustration shows the overall process of referencing logs for read or write operations.

Event Log Referencing



Referencing a Log to Read Entries

When you read entries from an event log, you uniquely identify the log you want by machine name and log name. *Machine name* denotes the server on which the event log exists, and *log name* is simply the name of the log. Machine name is optional; if you do not specify a machine, the local computer is assumed.

For example, suppose you have an instance of the **EventLog** component that you want to use to read entries from a custom log called "OrderEntrySystem, " which is stored on a server named "myserver". You would connect to this log by specifying that:

- The machine name is "myserver"
- The log name is "OrderEntrySystem"

You can retrieve the machine name of a log by querying the **MachineName** property. You can retrieve the log name by querying the **Log** property.

> **Security Note** Treat the data from an event log as you would any other input coming from outside your system. Your application may need to validate the data in the event log before using it as input. Another process, possibly a malicious one, may have accessed the event log and added entries.

Referencing a Log to Write Entries

If you are writing entries to an event log, you must specify the **Source** property. The **Source** property registers your component with the event log as a valid source of entries. You can specify any string as the value for the **Source** property; however, it may be helpful to use the name of your project for this value.

> **Note** If the source is already registered with the log, you do not need to set the log name property to write to the log. You can also not set the machine name; in this case the local machine will be assumed.

For example, suppose you have an instance of the **EventLog** component that you want to use to write entries to a custom log called "OrderEntrySystem", which is stored on the local computer. Assuming that the source has not already been registered, you would connect to this log by specifying that the machine name is ".", the log name is "OrderEntrySystem", and by setting the source to a string of your choice.

> **Note** In this case the period in the machine name indicates the local computer.

A single source can be associated with (and therefore write entries to) only one event log at a time, but a single event log can have many different sources writing to it at once. This means that each instance of the **EventLog** component, when configured to interact with a specific source, can write to only one event log. If you want to write to multiple logs, you must use different instances or change the **Source** property of an existing instance.

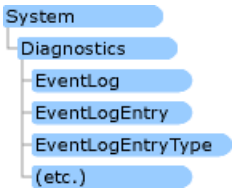
See Also

Introduction to the EventLog Component | Walkthrough: Exploring Event Logs, Event Sources, and Entries

EventLog Programming Architecture

The **EventLog** component uses these portions of the .NET Framework namespaces:

EventLog Namespace



When you add an instance of the **EventLog** component to your application in Visual Studio, the system automatically creates the references and import statements you need to access the namespace and classes. However, if you are creating your **EventLog** components in code or in the Microsoft .NET Framework, you might need to create a reference to **System.Diagnostics** and add a statement to your project importing the same library.

For instructions on adding and removing project references, see Adding and Removing References.

Key Programming Elements in the EventLog Class

The **EventLog** component raises an event called EventLog.EntryWritten Event whenever an entry is written to the log you are watching. You can define a handler for this event. In addition, you can use the following methods to interact with an instance of the **EventLog** component:

- You can use the EventLog.Exists (String) method to determine whether a particular event log exists.
- You can use the EventLog.CreateEventSource method to define a string that creates a relationship between a source of events and a log, and use the EventLog.DeleteEventSource method to remove this association. You can also use this method to create new custom event logs. Just type a new log name as the log in which you want to create your source, and the system will create the log automatically.

  > **Note** You can set the EventLog.Source property to any string; it does not have to correspond to the name of your application. A single application can contain multiple components that use different source strings to identify themselves to event logs.

- You can use the EventLog.WriteEntry method to write event information to any event log to which you have access. The **WriteEntry** method automatically registers your component or application as a valid source if it is not already registered with the designated log.
- You can use the EventLog.SourceExists method to determine whether a particular source is already registered. This is useful when you are attempting to create a new source on a given computer, because the **CreateEventSource** method will cause a run-time error if the source is already registered on that computer.
- You can use the EventLog.GetEventLogs method to retrieve a list of all the event logs on a particular computer.
- You can use the EventLog.Delete method to delete an entire event log. Use this method with caution, as deleting a log will also delete any sources registered in it.

See Also

Introduction to the EventLog Component

Administering Event Logs

You can perform administrative tasks on event logs on your local and remote computers, such as creating custom logs, deleting event logs, and clearing log entries.

> **Note** You must have appropriate access rights on the remote machines to create and delete logs. For more information, see Security Ramifications of Event Logs.

In This Section

Deleting Event Logs
Gives directions for deleting an existing log and all of its entries.
Determining If Specific Event Logs Exist
Gives directions for querying to see if a specific log exists.
Creating and Removing Custom Event Logs
Gives directions for creating custom event logs on the local computer.
Clearing Event Logs
Gives directions for deleting entries from an event log.

Related Sections

Introduction to the EventLog Component
Provides an overview of the **EventLog** component and how you can use it to interact with existing logs.

Deleting Event Logs

You can delete any event log on your local computer or a remote server, provided you have the appropriate registry rights. When you delete a log, the system first deletes the file that contains the log's contents and then accesses the registry and removes the registration for all of the event sources that were registered for that log. Even if you re-create the log at a later point, this process will not create the sources by default, so some applications that previously were able to write entries to that log may not be able to write to the new log.

You must have registry rights on the machine on which you want to delete a log. For more information, see your Windows documentation.

> **Note** Re-creating an event log can be a difficult process. It is good practice to not delete any of the system-created event logs, such as the Application log. You can delete your custom logs and re-create them as needed.

To delete an event log

- Call the **Delete** method and specify the name of the log you want to delete.

```
' Visual Basic
EventLog.Delete("MyCustomLog")

// C#
System.Diagnostics.EventLog.Delete ("MyCustomLog");
```

> **Note** Because the **Delete** method is static, you do not need to create an instance of the **EventLog** component before you call the method — instead, you can call the method on the **EventLog** class itself.
>
> **Tip** To delete an event log on a remote computer, specify the name of the computer as a second parameter.

The following code shows an example of verifying a source and deleting a log if the source exists. This code assumes that an **Imports** or **Using** statement exists for the **System.Diagnostics** namespace:

```
' Visual Basic
If EventLog.Exists("MyCustomLog") Then
   EventLog.Delete("MyCustomLog")
End If

// C#
if (System.Diagnostics.EventLog.Exists("MyCustomLog"))
{
   System.Diagnostics.EventLog.Delete("MyCustomLog");
}
```

See Also

Administering Event Logs | Security Ramifications of Event Logs | Walkthrough: Exploring Event Logs, Event Sources, and Entries

Determining If Specific Event Logs Exist

You can use the EventLog.Exists method to determine whether a specific log exists. If the log exists, the method returns **true**. You might use this method if you are creating a custom log and want to make sure that a log by that name does not already exist.

You must have registry rights in order to find out if a log exists. If you do not have the appropriate rights, the query will always return **false**. For information on setting registry rights, see your Windows documentation.

In addition to the **Exists** method, you can use a similar method, the EventLog.SourceExists method, to determine if a particular source string exists. See Determining If an Event Source Exists for more information.

> **Note** Because the **Exists** and **SourceExists** methods are static, you do not need to create an instance of the **EventLog** component before you call either method — instead, you can call the method on the **EventLog** class itself.

You must have appropriate access rights on the remote machines to create and delete logs. For more information, see Security Ramifications of Event Logs.

To determine if an event log exists

- Call the **Exists** method, specifying the log on the local computer that you want to verify as a parameter.

  The following example shows how to determine whether the Application log exists on the local machine. This code assumes that an **Imports** or **Using** statement exists for the **System.Diagnostics** namespace:

  ```
  ' Visual Basic
  Dim logExists As Boolean
  logExists = EventLog.Exists("Application")

  // C#
  bool logExists = System.Diagnostics.EventLog.Exists("Application");
  ```

  If the log exists, the message box will show "True." If it does not exist, the message box will show "False."

  > **Tip** You can specify a computer name as a second parameter to determine if an event log exists on a remote computer.

  > **Note** **Application** is case sensitive.

See Also

Administering Event Logs | Creating and Removing Custom Event Logs | Security Ramifications of Event Logs | Walkthrough: Exploring Event Logs, Event Sources, and Entries

Creating and Removing Custom Event Logs

You can use the **EventLog** class to create a custom event log on a local or remote computer. You might create a custom log if you wanted to organize your entries in a more granular way than is allowed when your components write entries to the default Application log. For example, suppose you have a component called OrderEntry that writes entry information to an event log. You are interested in backing up and saving these entries for a longer period of time than some other entries in the Application log. Rather than registering your component to write to the Application log, you can create a custom log called OrdersLog and register your component to write entries to that log instead. That way, all of your order information is stored in one place and will not be affected if the entries in the Application log are cleared.

You can indirectly use the **CreateEventSource** method to create a custom log. This method creates a new source and lets you specify the log to write to. If you specify that you want to write to a log that does not already exist, the system automatically creates a custom log for you and registers your component as a source for that log.

> **Note** You remove a custom log the same way you remove any log — by calling the **Delete** method. For more information, see Deleting Event Logs.

It is important to note the difference between creating an event log and creating an instance of the **EventLog** component. When you use the **CreateEventSource** method, you are creating a new custom event log in Windows, rather than a component in your project or application. When you create an instance of the **EventLog** component, you create a component inside your project that references an external event log. You can view the event logs created with the **CreateEventSource** method in the Event Viewer, but you cannot view component instances there.

You can view custom logs in Server Explorer, or in the Windows 2000 Computer Management window. For more information, see Working with Event Logs in Server Explorer or Launching Event Viewer.

> **Note** The Servers node of Server Explorer is not available in the Standard Edition of Visual Basic and Visual C# .NET. For more information, see Visual Basic Standard Edition Features or Visual C# Standard Edition Features.

You must have appropriate access rights on the remote machines to create and delete logs. For more information, see Security Ramifications of Event Logs.

> **Security Note** When you create an event log, be aware that the resource may already exist. Another process, perhaps a malicious one, may have already created the resource and have access to it. When you put data in the event log, the data is available to the other process. For information on existing event logs, see Determining If Specific Event Logs Exist.

To create a custom event log

- Call the **CreateEventSource** method and specify the source string and the name of the log file you want to create.

  > **Note** If you specify null ("") as the log name, it defaults to **Application**. This will not create a new log, but will register the specified source for the Application log. If you create a new log, only the first eight letters are evaluated when determining if the name is unique.

  The following example shows how to create a custom log called MyNewLog on the local machine. This code assumes that an **Imports** or **Using** statement exists for the **System.Diagnostics** namespace:

  ```
  ' Visual Basic
  EventLog.CreateEventSource("MyApp1", "MyNewLog")

  // C#
  System.Diagnostics.EventLog.CreateEventSource("MyApp1", "MyNewLog");
  ```

  To create a custom event log on a remote machine, specify the machine name as a third parameter. The following code shows an example.

  ```
  ' Visual Basic
  EventLog.CreateEventSource("MyApp1", "MyNewLog", "myserver")

  // C#
  System.Diagnostics.EventLog.CreateEventSource(
```

```
    "MyApp1", "MyNewLog", "myserver");
```

See Also

Administering Event Logs | Deleting Event Logs | Working with Event Logs in Server Explorer | Launching Event Viewer | Security Ramifications of Event Logs

Clearing Event Logs

Event logs are set to a maximum size that determines how many entries each log can contain. When an event log is full, it will either stop recording entries or begin overwriting the oldest entries with new entries, depending on the log's settings in the Windows Event Viewer. In either case, you can clear the log of its existing entries to free the log and allow it to start recording events again. You must have Administrator rights to the computer on which the log resides in order to clear entries.

By default, the Application, System, and Security logs are set to a default maximum size of 4992 K. Custom logs are set to a default maximum of 512 K.

> **Note** You can also use the Windows Event Viewer to free up space on a log that has become full. You can set the log to overwrite existing events, you can write log entries to an external file, or you can increase the maximum size of the log. You cannot, however, remove only some of the entries in a log; when you clear a log, you remove all of its contents. For more information, see Launching Event Viewer or your Event Viewer documentation.

To clear an event log's contents from Server Explorer

1. Open **Server Explorer**, and then expand the node for the server on which the event log you want to clear resides. For more information, see Accessing and Initializing Server Explorer.

   > **Tip** If you do not see the server you want listed in the **Servers** node, right-click **Servers**, click **Add Server**, enter the name of your server, and then click **OK**.

   > **Note** The Servers node of Server Explorer is not available in the Standard Edition of Visual Basic and Visual C# .NET. For more information, see Visual Basic Standard Edition Features or Visual C# Standard Edition Features.

2. Expand the **Event Logs** node.

   You will see a list of the three main event logs — Application, Security, and System.

3. Right-click the log node you want to clear, and then click **Clear Event Log**.
4. Click **Yes** when prompted to confirm the deletion.

To clear an event log's contents programmatically

1. Create an instance of the **EventLog** component and set the necessary properties to configure it. For more information, see Creating EventLog Component Instances and Configuring EventLog Component Instances.
2. Use the EventLog.Clear method to clear the contents of the appropriate event log.

   Your code should look something like this:

```
' Visual Basic
EventLog1.Clear()

// C#
EventLog1.Clear();
```

See Also

Working with Event Logs in Server Explorer | Launching Event Viewer | Accessing and Initializing Server Explorer | Administering Event Logs | Walkthrough: Exploring Event Logs, Event Sources, and Entries

Creating EventLog Component Instances

You create **EventLog** components when you want to add event log functionality to your application. **EventLog** components allow you to connect to existing event logs, write entries to them, and react to entries the logs receive.

EventLog components have no visual user interface. If you add an instance of the **EventLog** component to a visual item (such as a form), the component appears in a small area below the bottom border of the form. This is known as the component tray and acts as a place to display all of the non-visual items associated with the form.

> **Note** Non-visual classes that inherit from **System.ComponentModel.Component** support a visual design surface similar to the component tray, on which your **EventLog** component and other components are arranged. The arrangement of items on this designer is not important, as the interface you see will never be displayed to the application's end users.

There are several ways you can create an instance of the **EventLog** component:

- You can drag an instance of the **EventLog** component from the **Components** tab of the **Toolbox** to a form or other designer.
- You can locate the log you want in Server Explorer and add it to your designer, creating an instance of the **EventLog** component that is pre-configured to point to that log.
- You can create an instance of the **EventLog** component in code.

To create an EventLog component instance from Server Explorer

1. Open **Server Explorer**. For more information, see Accessing and Initializing Server Explorer.
2. Expand the node for the server you want to view, and then locate and expand the **Event Logs** node beneath it.

   > **Note** The Servers node of Server Explorer is not available in the Standard Edition of Visual Basic and Visual C# .NET. For more information, see Visual Basic Standard Edition Features or Visual C# Standard Edition Features.

3. Locate the log you want to add to your project, and then right-click it.
4. Click **Add to Designer**.
5. Set the necessary properties for the component. For more information, see Configuring EventLog Component Instances.

To create an EventLog component instance from the Toolbox

1. Access the **Components** tab of the **Toolbox**.
2. Select the **EventLog** icon and drag it to the designer surface for your form or component.
3. Set the necessary properties for the component. For more information, see Configuring EventLog Component Instances.

To create an EventLog component instance programmatically

1. Create an instance of the **EventLog** class in code. Your code might look like this:

```
' Visual Basic
Dim EventLog1 As New System.Diagnostics.EventLog()

// C#
System.Diagnostics.EventLog EventLog1 = new System.Diagnostics.EventLog();
```

2. Set the necessary properties for the component. For more information, see Configuring EventLog Component Instances.

See Also

Introduction to the EventLog Component | Configuring EventLog Component Instances | Adding Your Application as a Source of Event Log Entries | Determining If an Event Source Exists | Removing an Event Source | Walkthrough: Exploring Event Logs, Event Sources, and Entries

Configuring EventLog Component Instances

There are three major properties involved in configuring an instance of the **EventLog** component:

- The Log property indicates the log with which you want to interact.
- The MachineName property indicates the computer on which the log you are working with resides.
- The Source property indicates the source string which will be used to identify your component when it writes entries to a log.

The way you configure your **EventLog** component instance depends on how you are planning to use it:

- If you are planning to write entries to a log, you must do one of two things. Either you must register a source using the EventLog.CreateEventSource method and set the **EventLog** component's **Source** property to that same source, or you must set the **MachineName**, **Log**, and **Source** properties for the component.

  > **Note** You can set the **Log** property on your component directly, or you can set it in code that registers your component as an event source. After registering the component as a source, a connection is already set in the server's registry linking your component to a specific log.

- If you are planning to read log entries or monitor a log, you must set the **MachineName** and **Log** properties to indicate the log you are watching. You do not need to set the **Source** property in this case.
- If you want your component to be able to receive notification of **EntryWritten** events, you must set EventLog.EnableRaisingEvents to **true**. For more information, see Handling the EntryWritten Event.

  The following example shows how to configure your component to allow you to write entries to the Application log. In this case, the source has not already been registered with the log, so you must specify the source string, log, and machine name. You do not need to explicitly call the **CreateEventSource** method because the EventLog.WriteEntry call will register the source with the log if it has not already been registered.

```
' Visual Basic
Dim EventLog1 As New EventLog("Application", "myserver", "newsource")
EventLog1.WriteEntry("Test")

// C#
System.Diagnostics.EventLog EventLog1 = new
   System.Diagnostics.EventLog("Application", "myserver", "newsource");
EventLog1.WriteEntry("Test");
```

  If the source has already been registered, you can simply set the **Source** property and write your entry. For example, the following code shows how to write an entry to a log using a source that has already been registered.

```
' Visual Basic
If Not EventLog.SourceExists("ExistingSourceString") Then
    EventLog.CreateEventSource("ExistingSourceString", "Application")
End If
Dim EventLog1 As New EventLog()
EventLog1.Source = "ExistingSourceString"
EventLog1.WriteEntry("TestEntry2")

// C#
if (!System.Diagnostics.EventLog.SourceExists("ExistingSourceString"))
   System.Diagnostics.EventLog.CreateEventSource(
      "ExistingSourceString", "Application");

System.Diagnostics.EventLog EventLog1 =
   new System.Diagnostics.EventLog();
EventLog1.Source = "ExistingSourceString";
EventLog1.WriteEntry("TestEntry2");
```

Note If you do not set the **MachineName** property, as shown in these examples, the local machine is assumed.

For more information on these properties, see Event Log References.

The entries you retrieve from an event log are categorized by multiple properties. For information on these properties, see EventLogEntry Members.

To configure an instance of the EventLog component

1. Create an instance of the **EventLog** component. For more information, see Creating EventLog Component Instances.
2. Set the appropriate combination of the **Log**, **MachineName**, and **Source** properties:

If you will be	**Do this**
Reading or monitoring event logs	Set **Log** and **MachineName**
Writing entries to a log using a new **Source** string	Set **Source**, **Log**, and **MachineName**
Writing entries to a log for an existing **Source**	Set the **Source** property to the existing source; the other properties will be set automatically

3. To enable your component instance to receive notifications of the **EntryWritten** event, set the **EnableRaisingEvents** property to **true**.

See Also

Event Log References | Creating EventLog Component Instances | Adding Your Application as a Source of Event Log Entries | Writing Entries to Event Logs

Adding Your Application as a Source of Event Log Entries

Before your component can write an entry to an event log, you must register it with the event log as a valid source of events. You can do this by using the EventLog.CreateEventSource method and specifying a string that uniquely identifies your component to the event log. When you write a log entry, the system uses the source you identified to find the appropriate log in which to place your entry. Your **EventLog** component instance can write to only a single log at a time.

> **Note** By default, if you try to write an entry without first having registered your component as a valid source, the system automatically registers the source with the event log, using the value of the **Source** property as the source string.

If you specify the name of a log that does not exist when you use the **CreateEventSource** method, the system creates a new, custom event log for you with that name the first time you try to write an entry to the log. For more information, see Creating and Removing Custom Event Logs.

To establish your component as an event source

- To create entries on the local machine, call the **CreateEventSource** method and specify the source string and log name as parameters.

  > **Note** If you specify null (" ") for the log name, it defaults to **Application**.

  The following example shows how to register your component MyApp1 as a source for the Application log. This code assumes that an **Imports** or **Using** statement exists for the **System.Diagnostics** namespace:

```
' Visual Basic
EventLog.CreateEventSource("MyApp1", "Application")

// C#
System.Diagnostics.EventLog.CreateEventSource("MyApp1", "Application");
```

  > **Tip** To create an event source on a remote computer, specify the machine name as a third parameter. The following code shows an example:

```
' Visual Basic
EventLog.CreateEventSource("MyApp1", "Application", "myserver")

// C#
System.Diagnostics.EventLog.CreateEventSource(
   "MyApp1", "Application", "myserver");
```

See Also

Determining If an Event Source Exists | Removing an Event Source | Introduction to the EventLog Component | Creating and Removing Custom Event Logs | Walkthrough: Exploring Event Logs, Event Sources, and Entries

Determining If an Event Source Exists

Once you identify a source for a particular event log, this information is stored in the server's registry files until you remove that source. If you try to re-register a source that has already been registered as a valid source for a given log, the system raises a run-time error. You can use the EventLog.SourceExists method to determine whether or not a particular source has already been registered.

To determine if a source has already been registered

- Call the **SourceExists** method, specifying the source name to query.

  The following example shows how to determine if a source using the string MyApp1 is already registered, and to register it with the Application log if it is not.

```
' Visual Basic
If Not EventLog.SourceExists("MyApp1") Then
    EventLog.CreateEventSource("MyApp1", "Application")
End If

// C#
if (!System.Diagnostics.EventLog.SourceExists("MyApp1"))
   System.Diagnostics.EventLog.CreateEventSource(
      "MyApp1", "Application");
```

  To determine if a source is registered on a remote computer, specify the computer name as a second parameter. The following code shows an example:

```
' Visual Basic
If Not EventLog.SourceExists("MyApp1", "myserver") Then
    EventLog.CreateEventSource("MyApp1", "Application", "myserver")
End If

// C#
if (!System.Diagnostics.EventLog.SourceExists("MyApp1", "myserver"))
   System.Diagnostics.EventLog.CreateEventSource(
      "MyApp1", "Application", "myserver");
```

See Also

Removing an Event Source | Adding Your Application as a Source of Event Log Entries | Walkthrough: Exploring Event Logs, Event Sources, and Entries

Removing an Event Source

You can remove your source if you no longer need to use it to write entries to that log. Doing so affects all components that used that source to write to the log. For example, if you have two components that write to a log using the source string "mysource," removing "mysource" as a valid source of events affects both the components.

You might remove an event source if you were changing your source from one log to another. For example, suppose you want to stop writing entries to an OrdersProcessed log and start writing entries to an OrdersSent log. First, you would remove the association of the source with the first log, and then you would register the source with the new log to which you wanted to write events.

To remove an event source

- Call the EventLog.DeleteEventSource method, specifying the source name to remove.

  The following example shows how to unregister an event source named MyApp1 from the local machine. This code assumes that an **Imports** or **Using** statement exists for the **System.Diagnostics** namespace:

```
' Visual Basic
EventLog.DeleteEventSource("MyApp1")

// C#
System.Diagnostics.EventLog.DeleteEventSource("MyApp1");
```

  Tip To remove an event source from a remote computer, specify the computer name as a second parameter. The following code shows an example:

```
' Visual Basic
EventLog.DeleteEventSource("MyApp1", "myserver")

// C#
System.Diagnostics.EventLog.DeleteEventSource("MyApp1", "myserver");
```

  Note Removing a source does not remove the entries that were written to that log using this source. However, it does affect the entries by adding information to them indicating that the source cannot be found.

See Also

Determining If an Event Source Exists | Adding Your Application as a Source of Event Log Entries | Walkthrough: Exploring Event Logs, Event Sources, and Entries

Writing Entries to Event Logs

When you write an entry to an event log, you specify the message you want to write to the log as a string. A message should contain all information needed to interpret what caused the problem and what to do to correct the problem.

There are two ways you can write an entry to the log; both are equally valid. The most direct way is to register an event source with the log to which you want to write, then instantiate a component and set its **Source** property to that log, and finally call **WriteEntry**. If you do this, you do not have to set the **Log** property for the component instance; the log is automatically determined when you connect to a source that has already been registered. For more information on registering a source, see Adding Your Application as a Source of Event Log Entries.

The other way to approach this process is to instantiate an **EventLog** component, set its **Source**, **Machine**, and **Log** properties, and call the **WriteEntry** method. In this case, the **WriteEntry** method would determine whether the source already existed and register it on the fly if it did not.

The following conditions must be met in order to successfully write a log entry:

- The source must be registered with the desired log.

  > **Note** You must set the **Source** property on your **EventLog** component instance before you can write entries to a log. When your component writes an entry, the system automatically checks to see if the source you specified is registered with the event log to which the component is writing, and calls **CreateEventSource** if needed.

- The message you specify cannot be more than 16K in length.
- Your application must have write access to the log to which it is attempting to write. For more information, see Security Ramifications of Event Logs.

You can specify several parameters when you write an entry, including the type of entry you're making, an ID that identifies the event, a category, and any binary data you want to append to the entry. For more information on the properties associated with an entry, see EventLog Members.

To write an event log entry

1. Instantiate an **EventLog** component. For more information, see Creating EventLog Component Instances.
2. Use the EventLog.CreateEventSource method to register an event source with the log to which you want to write an entry, using a unique string for the source string. Set the **Source** property for the component to the source you registered. For more information, see Configuring EventLog Component Instances. Call the **WriteEntry** method to specify the entry to be written to the log.

```
' Visual Basic
If Not EventLog.SourceExists("MyApp1") Then
    EventLog.CreateEventSource("MyApp1", "Application")
End If
EventLog1.Source = "MyApp1"
EventLog1.WriteEntry("This is a simple event log entry")

// C#
if (!System.Diagnostics.EventLog.SourceExists("MyApp1"))
   System.Diagnostics.EventLog.CreateEventSource(
      "MyApp1", "Application");

EventLog1.Source = "MyApp";
EventLog1.WriteEntry ("This is a simple event log entry");
```

See Also

Introduction to the EventLog Component | Configuring EventLog Component Instances | Creating EventLog Component Instances | EventLog Members | Adding Your Application as a Source of Event Log Entries | Walkthrough: Exploring Event Logs, Event Sources, and Entries

Reading Event Log Entries

You use the **Entries** collection to look at the entries in a particular event log. You can use standard collection properties such as **Count** and **Item** to work with the elements the collection contains. You might read event log entries to find out more information about a problem that occurred in your system, to identify usage patterns, or to identify problems (such as a failing hard drive) before they cause damage.

> **Note** The **Entries** collection is read-only; to write a message to a log you must use the **WriteEntry** method. For information, see Writing Entries to Event Logs.

If you ask for the count of entries in a new custom log that has not yet been written to, the system will return the count of the entries in the Application log on that server. Make sure that logs you are counting have been created and written to in order to avoid this problem.

> **Note** There are security restrictions that affect your ability to use event logs. For more information, see Security Ramifications of Event Logs.
>
> **Security Note** Treat the data from an event log as you would any other input coming from outside your system. Your application may need to validate the data in the event log before using it as input. Another process, possibly a malicious one, may have accessed the event log and added entries.

To read event log entries

1. Create an instance of the **EventLog** component. For more information, see Creating EventLog Component Instances.
2. Set the **Log** and **MachineName** properties for the component. For more information, see Configuring EventLog Component Instances.
3. Use the **Entries** collection to review the entries in the log. The following example shows how to retrieve all of the entries from a log.

```
' Visual Basic
Dim entry As EventLogEntry
For Each entry In EventLog1.Entries
   Console.WriteLine(entry.Message)
Next

// C#
foreach (System.Diagnostics.EventLogEntry entry in EventLog1.Entries)
{
   Console.WriteLine(entry.Message);
}
```

See Also

Introduction to the EventLog Component | Handling the EntryWritten Event | Walkthrough: Exploring Event Logs, Event Sources, and Entries

Handling the EntryWritten Event

You can create event handlers for your **EventLog** components that automatically call a procedure when an entry is written to a log. There are several ways you can create an event handler for an instance of the **EventLog** component, but the simplest way is to let Visual Studio .NET handle most of the syntax for you automatically. When you double-click your **EventLog** component in the designer, Visual Studio moves you to the Code Editor and creates the event handler and the blank procedure it calls. You can then fill in the processing that you want to occur in the **EntryWritten** procedure.

For more information on event handlers, see Event Handlers in Visual Basic and Visual C#.

To create a default handler for the EntryWritten event

1. From the designer, double click on the **EventLog** component for which you want to create a handler.

   Note The Code Editor appears and two items are added to your code: the handler that creates and registers a delegate and calls the procedure, and a blank procedure for the EventLog.EntryWritten Event event.

2. In the blank procedure for the **EntryWritten** event, define code to receive and process entries when this event is called. Your code might look like this:

```
' Visual Basic
Private Sub EventLog1_EntryWritten(ByVal sender As System.Object, _
   ByVal e As System.Diagnostics.EntryWrittenEventArgs) _
   Handles EventLog1.EntryWritten

   If e.Entry.Source = "MyApplication" Then
      Console.WriteLine("Entry written by my app. Message: " & _
         e.Entry.Message)
   Else
      Console.WriteLine("Entry written by another application. ")
   End If
End Sub


// C#
private void eventLog1_EntryWritten(object sender,
   System.Diagnostics.EntryWrittenEventArgs e)
{
   if (e.Entry.Source == "MyApplication")
      Console.WriteLine("Entry written by my application. Message: "
         + e.Entry.Message);
   else
      Console.WriteLine("Entry was written by another application.");
}
```

3. Set the **EnableRaisingEvents** property to **true**.

To create a handler programmatically

1. Use the **AddHandler** method to create an event handler of type **EventLogEventHandler** for your component that will call the **EventLog1.EntryWritten** procedure when an entry is written to the log. Your code should look like this:

```
' Visual Basic
AddHandler EventLog1.EntryWritten, _
   New System.Diagnostics.EntryWrittenEventHandler( _
   AddressOf Me.EventLog1_EntryWritten)

// C#
this.eventLog1.EntryWritten += new
   System.Diagnostics.EntryWrittenEventHandler(
   this.eventLog1_EntryWritten);
```

Note For more information on this syntax, see Event Handlers in Visual Basic and Visual C#.

2. Create the **EntryWritten** procedure and define the code you want to process the entries.
3. Set the **EnableRaisingEvents** property to **true**.

See Also

Event Handlers in Visual Basic and Visual C#| Configuring EventLog Component Instances |
Configuring EventLog Component Instances

Working with Event Logs in Server Explorer

Using Server Explorer, you can explore existing event logs on any computer to which you have access. You can view any log available on these computers, including those logs you create from Visual Studio. NET.

> **Note** The Servers node of Server Explorer is not available in the Standard Edition of Visual Basic and Visual C# .NET. For more information, see Visual Basic Standard Edition Features or Visual C# Standard Edition Features. You can also see information about event logs in the Microsoft Windows 2000 Computer Management window. For more information, see Launching Event Viewer.

To locate an event log in Server Explorer

1. Expand the **Servers** node, and then expand the node for the server you want to view.

   > **Tip** If the server you want to view is not listed, you need to add it. For more information, see Accessing and Initializing Server Explorer.

2. Locate and expand the **Event Logs** node beneath it.

To locate a log in Computer Management on a Windows 2000 server

1. On the desktop for your computer, right-click **My Computer**, and then click **Manage**.
2. In the dialog box that appears, expand the **System Tools** node.
3. Expand the **Event Viewer** node. A full list of the event logs on the computer appears. You can click any log to see the entries it contains.

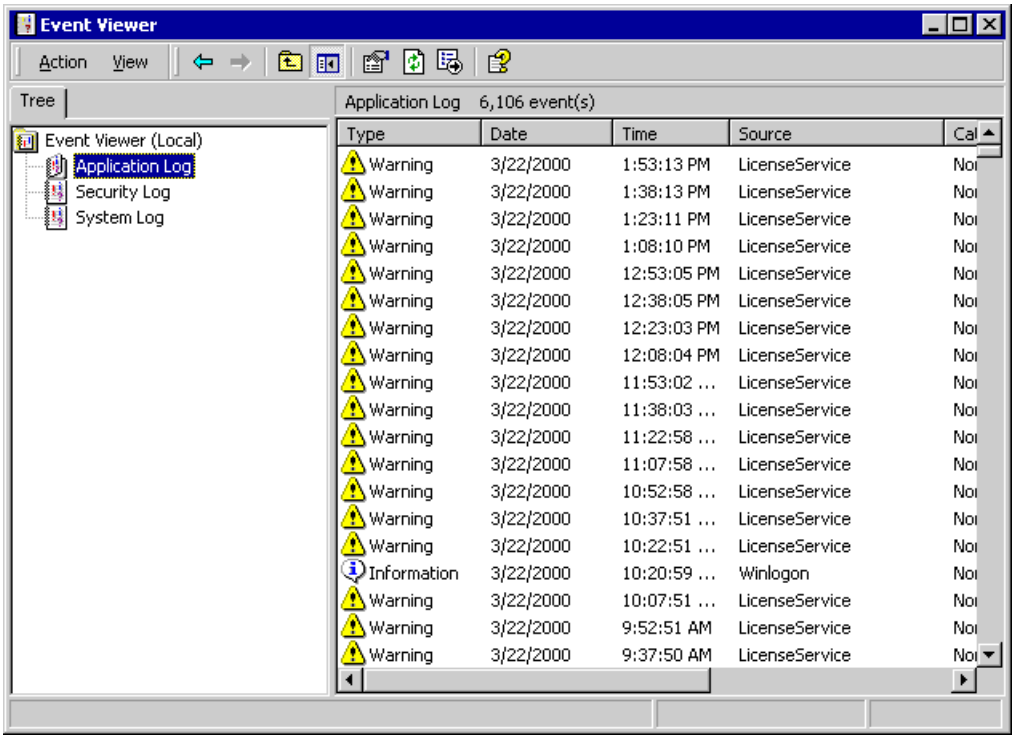
See Also

Introduction to the EventLog Component | Launching Event Viewer | Clearing Event Logs | Accessing and Initializing Server Explorer

Launching Event Viewer

Event Viewer is a Microsoft Windows 2000 application that allows you to monitor and administer event logs and their entries. You can launch the Event Viewer application from Server Explorer if you want to access the information it contains directly rather than through your Visual Studio .NET code. The following illustration shows the Event Viewer window.

Event Viewer Window



To launch Event Viewer

1. Access **Server Explorer** and locate the **Event Logs** node. For more information, see Accessing and Initializing Server Explorer.
2. Expand the **Servers** node.
3. Right-click the **Event Logs** node and select **Launch Event Viewer**.

See Also

Working with Event Logs in Server Explorer | Clearing Event Logs | Accessing and Initializing Server Explorer

Security Ramifications of Event Logs

Access to the event logs is determined by the account under which the application is running. The LocalSystem account is a special account that service applications can use. The Administrator account consists of the administrators for the system. The Server Operator account (ServerOp) consists of the administrators of the domain server. The World account includes all users on all systems.

The following table shows the accounts that are granted Read, Write, and Clear access to each log.

Log	Account	Access
Application	LocalSystem	Read Write Clear
	Administrator	Read Write Clear
	ServerOp	Read Write Clear
	World	Read Write
Security	LocalSystem	Read Write Clear
	Administrator	Read Write
	World	None
System	LocalSystem	Read Write Clear
	Administrator	Read Write Clear
	World	Read Clear
	ServerOp	Read

In addition, users can read and clear the Security log if they have been granted one of the following:

- The "manage auditing and Security log" user right.
- The SE_AUDIT_NAME privilege. For more information, see Authorization Data Types and Constants.

For more information, see your Windows documentation.

If you are using event logs in an ASP.NET application, access to the event logs is through another account, the ASPNET account. The default settings of the ASPNET user account restrict access to the event logs. The ASPNET user account does not have permission to create new categories, though it can add entries to an existing log. You can use impersonation with the ASPNET account to allow creation of new categories. The impersonation identity must have sufficient privileges to create categories. If your application needs event logs that can be specified before deployment, they can be created by the deployment project. For more information, see ASP.NET Web Application Security.

When you create an event log, be aware that the resource may already exist. Another process, perhaps a malicious one, may have already created the resource and have access to it. When you put data in the event log, the data is available to the other process. For information on existing event logs, see Determining If Specific Event Logs Exist.

See Also

Introduction to the EventLog Component

Walkthrough: Exploring Event Logs, Event Sources, and Entries

This walkthrough will take you through the major feature areas of event logging in your Visual Studio application. During this walkthrough, you will learn how to:

- Create an **EventLog** component.
- Write code to create and delete custom event logs.
- Write entries of various types to the custom log.
- Read entries from the custom log.
- Verify the existence of logs and event sources.
- Clear log entries.
- Use Server Explorer to verify the results of your event log actions.

Creating the User Interface

In this walkthrough, you will create a Windows Form and use a series of controls on it to launch a series of event-logging actions.

To create the form and controls for your application

1. From the **New Project** dialog box, create a Visual Basic or C# Windows Application, and name it **EventLogApp1**.
2. Add eight buttons to the Windows Form, and set the following properties for them:

Control	Text Property	Name Property
Button1	**Create Custom Log**	**CreateLog**
Button2	**Delete Log**	**DeleteLog**
Button3	**Write Entry**	**WriteEntry**
Button4	**Clear Log**	**ClearLog**
Button5	**Verify Log Exists**	**VerifyLog**
Button6	**Verify Source Exists**	**VerifySource**
Button7	**Remove Event Source**	**RemoveSource**
Button8	**Read Entry**	**ReadEntry**

3. In turn, do the following for each button:
   a. In the designer, double-click the button to create a default event handler for that button. The Code Editor appears and a sub for the button's **Click** event appears.
   b. Return to Design view, and double-click the next button.
   c. Continue until you have created a default event handler sub for each button.
4. From the **Components** tab of the Toolbox, drag an **EventLog** component to the form.

   An **EventLog** component instance appears in the component tray area at the bottom of the form.

Creating and Deleting a Custom Log

In this procedure, you will use the **SourceExists** method to verify that the source you are using does not already exist, and then you will call the **CreateEventSource** method with the name of a log that does not exist. Because this log does not exist, the system will create a custom log for you when this code is run.

To create the custom log

1. In the Code Editor, locate the **CreateLog_Click** procedure.
2. Type in the following code. The event log and source are created as a pair, and the source cannot already exist before the event log is created. After creating the event log, the **EventLog** component is configured to access the new event log.

```
' Visual Basic
' Source cannot already exist before creating the log.
If EventLog.SourceExists("Source1") Then
   EventLog.DeleteEventSource("Source1")
End If

' Logs and Sources are created as a pair.
EventLog.CreateEventSource("Source1", "NewLog1")
```

```
' Associate the EventLog component with the new log.
EventLog1.Log = "NewLog1"
EventLog1.Source = "Source1"

// C#
// Source cannot already exist before creating the log.
if (System.Diagnostics.EventLog.SourceExists("Source1"))
{
   System.Diagnostics.EventLog.DeleteEventSource("Source1");
}

// Logs and Sources are created as a pair.
System.Diagnostics.EventLog.CreateEventSource("Source1", "NewLog1");
// Associate the EventLog component with the new log.
eventLog1.Log = "NewLog1";
eventLog1.Source = "Source1";
```

Security Note When you create an event log, you need to decide what to do if that resource already exists. Another process, perhaps a malicious one, may have already created the event log and have access to it. When you put data in the event log, the data is available to the other process.

To delete a custom log

1. In the Code Editor, locate the **DeleteLog_Click** procedure.
2. Type in the following code:

```
' Visual Basic
If EventLog.Exists("NewLog1") Then
   EventLog.Delete("NewLog1")
End If

// C#
if (System.Diagnostics.EventLog.Exists("NewLog1"))
{
   System.Diagnostics.EventLog.Delete("NewLog1");
}
```

Writing Entries to the Log

In this procedure, you will use the **EventLog** component instance you created to write entries to the log. To do so, you will first configure the component to use the source string you just created, then you will specify two entries to write: an informational event and an error event.

To write entries to the log

1. In the Code Editor, locate the **WriteEntry_Click** procedure.
2. Type in the following code. This code uses the overloaded EventLog.WriteEntry method to write to the event log. The second form shown allows you to specify the type of message. If you view the entries using the Server Explorer in the "Testing Your Code" section below, the different types of entries are indicated by different icons.

```
' Visual Basic
EventLog1.WriteEntry("This is an informational message")
EventLog1.WriteEntry("This is an error message", _
   Diagnostics.EventLogEntryType.Error)

// C#
eventLog1.WriteEntry("This is an informational message");
eventLog1.WriteEntry("This is an error message",
   System.Diagnostics.EventLogEntryType.Error);
```

Clearing Log Entries

In this procedure, you will use the **Clear** method to remove existing entries from the custom log.

To clear log entries

1. In the Code Editor, locate the **ClearLog_Click** procedure.
2. Call the **Clear** method on the **EventLog** component instance:

```
' Visual Basic
EventLog1.Clear()

// C#
eventLog1.Clear();
```

Verifying Logs and Sources

In this procedure, you will create two procedures: one that verifies the existence of the custom log, and one that verifies the existence of your source string. These procedures will be used to test the results of various actions you perform when you run the project.

To verify that the custom log exists

1. In the Code Editor, locate the **VerifyLog_Click** procedure.
2. Create a message box that evaluates whether the specified event log exists and displays true or false accordingly. Use this code:

```
' Visual Basic
Dim logExists As Boolean = EventLog.Exists("NewLog1")
MessageBox.Show("Does the log exist? " & logExists.ToString())

// C#
bool logExists = System.Diagnostics.EventLog.Exists("NewLog1");
MessageBox.Show("Does the log exist? " + logExists.ToString());
```

To verify that the source exists

1. In the Code Editor, locate the **VerifySource_Click** procedure.
2. Create a message box that evaluates whether the specified source exists and displays true or false accordingly. Use this code:

```
' Visual Basic
Dim sourceExists As Boolean = EventLog.SourceExists("Source1")
MessageBox.Show("Does the source exist? " + sourceExists.ToString())

// C#
bool sourceExists =
   System.Diagnostics.EventLog.SourceExists("Source1");
MessageBox.Show("Does the source exist? " + sourceExists.ToString());
```

Removing Sources

In this procedure, you will write code to delete a source string. To do so, you will first verify that the source in question (Source1) exists, then call the **DeleteEventSource** method to remove it.

To remove the event source you created

1. In the Code Editor, locate the **RemoveSource_Click** procedure.
2. Add the following code:

```
' Visual Basic
If EventLog.SourceExists("Source1") Then
```

```
   EventLog.DeleteEventSource("Source1")
End If

// C#
if (System.Diagnostics.EventLog.SourceExists("Source1"))
{
   System.Diagnostics.EventLog.DeleteEventSource("Source1");
}
```

Reading Entries

In this procedure, you will write code to iterate through the event log's entries collection and display the existing messages in the log.

To read entries from the custom log you created

1. In the Code Editor, locate the **ReadEntry_Click** procedure.
2. Add the following code:

```
' Visual Basic
Dim entry As EventLogEntry
If EventLog1.Entries.Count > 0 Then
   For Each entry In EventLog1.Entries
      System.Windows.Forms.MessageBox.Show(entry.Message)
   Next
Else
   MessageBox.Show("There are no entries in the log.")
End If

// C#
if (eventLog1.Entries.Count > 0)
{
   foreach (System.Diagnostics.EventLogEntry entry
      in eventLog1.Entries)
   {
      MessageBox.Show(entry.Message);
   }
}
else
{
   MessageBox.Show("There are no entries in the log.");
}
```

Testing your Code

In this section, you will use Server Explorer to verify the results of your code.

Launching Server Explorer

1. From the **View** menu, access Server Explorer.
2. Expand the node for your current server, and then expand the **Event Logs** node beneath it.

> **Note** The Servers node of Server Explorer is not available in the Standard Edition of Visual Basic and Visual C# .NET. For more information, see Visual Basic Standard Edition Features or Visual C# Standard Edition Features. In this case, you can use the Windows Event Viewer instead to view the results of your application.

Build and run your application

1. Save your files and press **F5** to build and launch your project. The form appears with the eight buttons you created.
2. Click the **Create Custom Log** button.

> **Note** You must have appropriate privileges to the server on which your application runs in order to create Windows event logs. If you receive a security error at this point, see your system administrator.

3. Return to the product, while still in run mode, and right-click the **Event Logs** node in Server Explorer.
4. Click **Refresh**.
5. Verify that the **NewLog1** log now appears in the **Event Logs** node.

To test creating, deleting, and verifying custom logs

1. Return to your running form, and click the **Verify Log Exists** button.

   A prompt should appear with the text set to "True."

2. Click the **Delete Log** button, and then click the **Verify Log Exists** button again.

   This time, the prompt should say "False."

3. Click the **Create Custom Log** button again to recreate the log.

To test writing entries to and reading entries from the custom log

1. In the form, click the **Write Entry** button.
2. Access Server Explorer, and expand the **NewLog1** log.

   > **Note** The Servers node of Server Explorer is not available in the Standard Edition of Visual Basic and Visual C# .NET. For more information, see Visual Basic Standard Edition Features or Visual C# Standard Edition Features.

3. Expand the **Source1** node beneath it.

   You should now see that two entries have been added to the log. One will have an icon indicating that it is an informational entry, and one will have an icon indicating that it is an error.

4. Return to the form, and click the **Read Entry** button.

   You will receive two prompts: one containing the informational entry and one containing the error entry.

   > **Note** If you clicked the **Write Entry** button multiple times, you will have received more prompts.

To test clearing the log

1. In the form, click the **Clear Log** button.
2. In Server Explorer, right-click the **NewLog1** node, and then click **Refresh**.

   You should now see that the log no longer contains entries.

To test removing the source string

1. In the form, click the **Remove Event Source** button.
2. Click the **Verify Source** button. You should receive a prompt that says "False" to indicate that the source **Source1** no longer exists.
3. Click the **Write Entry** button.

   > **Note** This action will allow you to write entries to the log, because the **WriteEntry** method will set the source if it does not currently exist.

4. In Server Explorer, right-click the **NewLog1** node, and then click **Refresh**. You should see two entries in the log.
5. Click the **Verify Source** button again. You should receive a prompt that says "True" to indicate that the source **Source1** exists.
6. Optionally, you may want to click the **DeleteLog** button when you are through testing. This will remove the log **NewLog1** and the source **Source1** so that there are no permanent changes to your system's configuration.

See Also

System Monitoring Walkthroughs | Logging Application, Server, and Security Events

Monitoring Windows Services

You can use the **ServiceController** component to interact with the Windows services on your local computer or on any computer to which you have access. The **ServiceController** component allows you to perform administrative tasks such as starting, stopping, and pausing services; to retrieve lists of available services; and to call custom commands on the service.

In This Section

Introduction to Communicating with Existing Services
Gives directions for using the **ServiceController** component to interact with an existing service.
Creating ServiceController Component Instances
Gives directions for creating and configuring **ServiceController** components.
Performing Administrative Tasks on Services
Gives directions for starting, stopping, and pausing your service through the methods available on the **ServiceController** component.
Retrieving Lists of Services
Gives directions for returning lists of services on a specific computer.

Related Sections

Windows Service Applications
Provides links to topics about working with Windows services.
Introduction to Windows Service Applications
Provides an overview of the important concepts about Windows services.

Introduction to Communicating with Existing Services

You can use the **ServiceController** component to connect to and control the behavior of existing services. When you create an instance of the **ServiceController** component, you set it to interact with a specific Windows service. You can then use the component instance to start, stop, and otherwise manipulate the service. For example, you might create a **ServiceController** component and use it with a **Timer** component to start and stop a service based on a timed interval.

You can do several useful things with the **ServiceController** component. For example:

- You can start and stop the service to which you are connected, pause it, or perform any of the other administrative actions to which the service is capable of responding. For more information, see Performing Administrative Tasks on Services.
- You can query various properties of the service with which the component interacts and retrieve the property values. For example, you might determine whether the service can be stopped, paused, and continued by retrieving the value of the **CanStop** and **CanPauseandContinue** properties.
- You can return a list of the services that exist on a particular computer.
- You can specify a custom command to call on your service. For more information on custom commands, see MSDN Online.

The **ServiceController** component actually passes its requests for actions to the Services Control Manager, rather than to the service itself. The Services Control Manager then stops, starts, or pauses the service or performs other actions as specified. After the **ServiceController** has passed the request on to the Services Control Manager, it returns; it does not wait for the Services Control Manager to pass the request on to the service. Because of this, you should handle exceptions within the service class itself rather than in the **ServiceController** class.

> **Note** In general, you should not add a **ServiceController** component to a Windows Service application if your intent is for that component to manipulate the service contained in the same project. Code within the project that creates a service cannot be used to control the service; it must be controlled from another context.

See Also

Creating ServiceController Component Instances | Performing Administrative Tasks on Services | Retrieving Lists of Services

Creating ServiceController Component Instances

You can use the **ServiceController** component to connect to and control the behavior of existing services. When you create an instance of the **ServiceController** class, you set its properties so that it interacts with a specific Windows service. You can then use the class to start, stop, and otherwise manipulate the service.

You will most likely use a **ServiceController** in an administrative capacity. For example, you might create a Windows or ASP .NET Web application that sends custom commands to a service via a **ServiceController** component instance. This would be useful because the Services Control Manager does not support custom commands.

There are several ways you can create an instance of the **ServiceController** component:

- You can drag an instance of the **ServiceController** component from the **Components** tab of the **Toolbox** to a form or other designer.
- You can add a **ServiceController** component to your designer from Server Explorer
- You can create an instance of the **ServiceController** class in code.

After you create an instance of **ServiceController**, you must set two properties on it to identify the service with which it interacts: the machine name and the name of the service you want to control.

> **Note** By default, **MachineName** is set to the local computer, so you do not need to change it unless you want to set the component instance to point to another computer.

To create and configure a ServiceController component from Server Explorer

1. In Server Explorer, add the server you want if it is not already listed. For more information, see Accessing and Initializing Server Explorer.

   > **Note** The Servers node of Server Explorer is not available in the Standard Edition of Visual Basic and Visual C# .NET. For more information, see Visual Basic Standard Edition Features or Visual C# Standard Edition Features.

2. Expand the **Services** node, and then locate the service on which you want to perform administrative tasks.
3. Right-click the name of the service, and click **Add to Designer**.

   A **ServiceController** component appears in your project, configured to interact with the selected service.

To create and configure a ServiceController component from the Toolbox

1. Access the **Components** tab of the **Toolbox**.
2. Select the **ServiceController** icon and drag it to the designer surface for your form or component.
3. Set the following properties.

Property	Setting
MachineName	The name of the computer on which the service exists, or "." for the local computer.
ServiceName	The name of the service with which you want to interact.

> **Tip** You can use **Server Explorer** to see what the **ServiceName** should be set to for any given service. Select the service in which you're interested in **Server Explorer**, and the **ServiceName** for it appears in the **Properties** window.

To programmatically create and configure a ServiceController component

1. Create an instance of the **ServiceController** class in your code.
2. Set the **MachineName** and **ServiceName** properties to indicate the service you want to control.

   The following example shows how to create a **ServiceController** component that interacts with the IIS Admin service on the local computer. The component then queries the service associated with the controller to determine if it can accept Stop commands, and if so, issues the command. This example is part of a Windows project in which a form with various buttons is used to start, stop, and otherwise manipulate the service. Each time a button is clicked, the condition is evaluated and status information is displayed in a label control.

   > **Note** Before these samples will run, you must add a reference to **System** and **System.ServiceProcess** DLLs and include an **Imports** or **Using** statement for the both namespaces.

```
' Visual Basic
Sub Main()
   Dim myController As _
      New System.ServiceProcess.ServiceController("IISAdmin")

   If myController.CanStop Then
      Debug.WriteLine(myController.ServiceName & " can be stopped.")
   Else
      Debug.WriteLine(myController.ServiceName & " cannot stop.")
   End If
End Sub

// C#
public static void Main(string[] args)
{
   System.ServiceProcess.ServiceController myController =
      new System.ServiceProcess.ServiceController("IISAdmin");
   if (myController.CanStop)
   {
      System.Diagnostics.Debug.WriteLine(
         myController.DisplayName + "  can be stopped.");
   }
   else
   {
      System.Diagnostics.Debug.WriteLine(
         myController.DisplayName + "  cannot stop.");
   }
}
```

See Also

Introduction to Communicating with Existing Services | Performing Administrative Tasks on Services | Retrieving Lists of Services

Performing Administrative Tasks on Services

You can perform a variety of standard administrative tasks on a service, including starting and stopping it, pausing it, and restarting it. You can perform these tasks in two ways: via Server Explorer, or by calling methods on a **ServiceController** component instance that has been connected to an existing service.

Generally, the service author writes code that customizes the action associated with a specific task. For example, a service can contain code to respond to an **OnPause** command. In that case, the custom processing for the Pause task is carried out before the service is paused. For information on defining functioning for the service, see Creating Windows Services.

The particular tasks that are available for a service depend on its properties; for example, you can set the **CanStop** property for a service to **false**. This renders the Stop command unavailable on that particular service; if you try to stop the service from Server Explorer, the necessary menu item appears dimmed. If you try to stop the service from code, the system raises an error: "Failed to stop <servicename.>"

To perform an administrative task from Server Explorer

1. In **Server Explorer**, add the server you want if it is not already listed. For more information, see Accessing and Initializing Server Explorer.

   Note The Servers node of Server Explorer is not available in the Standard Edition of Visual Basic and Visual C# .NET. For more information, see Visual Basic Standard Edition Features or Visual C# Standard Edition Features.

2. Expand the **Services** node, and then locate the service on which you want to perform administrative tasks.
3. Right-click the name of the service, and click the task you want to perform.

To perform an administrative task programmatically

1. Create an instance of the **ServiceController** class and configure it to interact with the service you want to administer. For more information, see Creating ServiceController Component Instances.
2. Call the appropriate method to start, stop, pause, or continue the service.

   Tip You might want to query the corresponding property for any of these commands to determine if the service will respond to your request; that is, you might want to create a conditional statement that queries the **CanStop** property on the service and determine whether it is set to **true** before you call the **Start** method.

   The following example shows how to evaluate whether a service can accept a given command before issuing the command.

```
' Visual Basic
If myController.CanStop Then
   myController.Stop()
End If

// C#
if (myController.CanStop)
{
   myController.Stop();
}
```

See Also

Introduction to Communicating with Existing Services | Creating ServiceController Component Instances | Retrieving Lists of Services | Accessing and Initializing Server Explorer | Creating Windows Services

Retrieving Lists of Services

You can use the **GetServices** method on the **ServiceController** class to retrieve a list of the services on a particular computer. The **GetServices** method returns an array of all of a computer's available services, except for those associated with device drivers. You can retrieve the device driver services using the **GetDevices** method.

For both of these methods, you can either choose a specific computer you're interested in retrieving information about, or you can pass no parameters to retrieve the services on the local computer only.

To retrieve a list of services

1. Create an array of type **ServiceController** to contain the results of your query.
2. Call the appropriate method on the **ServiceController** class and set the results to the array, using one of the following forms:

   - To retrieve non-device driver services from the local computer, call the **GetServices** method without any parameters:

     ```
     ' Visual Basic
     Dim services() As System.ServiceProcess.ServiceController
     services = System.ServiceProcess.ServiceController.GetServices()

     // C#
     System.ServiceProcess.ServiceController[] services;
     services = System.ServiceProcess.ServiceController.GetServices();
     ```

   - To retrieve non-device driver services from a specific computer, call the **GetServices** method and specify the computer from which you want to retrieve information as a string parameter:

     ```
     ' Visual Basic
     Dim services() As System.ServiceProcess.ServiceController
     services = System.ServiceProcess.ServiceController.GetServices( _
        "machinename")

     // C#
     System.ServiceProcess.ServiceController[] services;
     services = System.ServiceProcess.ServiceController.GetServices(
        "machinename");
     ```

   - To retrieve device driver services from the local computer, call the **GetDevices** method without any parameters:

     ```
     ' Visual Basic
     Dim services() As System.ServiceProcess.ServiceController
     services = System.ServiceProcess.ServiceController.GetDevices()

     // C#
     System.ServiceProcess.ServiceController[] services;
     services = System.ServiceProcess.ServiceController.GetDevices();
     ```

   - To retrieve device driver services from a specific computer, call the **GetDevices** method and specify the computer from which you want to retrieve information as a string parameter:

     ```
     ' Visual Basic
     Dim services() As System.ServiceProcess.ServiceController
     services = System.ServiceProcess.ServiceController.GetDevices( _
        "machinename")

     // C#
     System.ServiceProcess.ServiceController[] services;
     services = System.ServiceProcess.ServiceController.GetDevices(
        "machinename");
     ```

The following code shows how to retrieve a set of services on the local computer and display information about each of them in a listbox. Note that the **GetServices** function returns an array of **ServiceController** objects. Each **ServiceController** returned represents one service on the computer being queried:

```
' Visual Basic
Sub ListServices()
   Dim ListBox1 As ListBox

   Dim services() As System.ServiceProcess.ServiceController
   Dim i As Integer
   services = System.ServiceProcess.ServiceController.GetServices()
   ListBox1.Items.Clear()
   For i = 0 To services.Length - 1
      ListBox1.Items.Add(services(i).ServiceName)
   Next
End Sub


// C#
private void ListServices()
{
   ListBox listBox1 = null;

   System.ServiceProcess.ServiceController[] services;
   services = System.ServiceProcess.ServiceController.GetServices();
   listBox1.Items.Clear();
   for (int i = 0; i <  services.Length; i++)
   {
      listBox1.Items.Add(services[i].ServiceName);
   }
}
```

See Also

Introduction to Communicating with Existing Services | Creating ServiceController Component Instances | Performing Administrative Tasks on Services

Monitoring and Managing Windows Processes

The **Process** component gives you the ability to view and manage most Windows process tasks. These tasks include starting and stopping processes and retrieving process property information, such as memory and thread usage.

In This Section

Introduction to Monitoring and Managing Windows Processes
Provides an overview of the **Process** component and how you can use it to interact with system resources.
Creating and Binding to Processes
Provides a list of topics relating to creating **Process** components.
Managing Processes
Explains how to use **Process** components to manage existing processes on your computer, such as starting, stopping, and retrieving process information.
Retrieving Information About Processes
Describes how to retrieve process property information, such as memory and thread usage.
Walkthrough: Managing a Windows Process
Provides step-by-step instructions for working with a **Process** component.

Related Sections

Process Class
Provides syntax and related reference information about the **Process** component.
Creating System Monitoring Components
Provides links to information about the **PerformanceMonitor**, **EventLog**, and **ServiceController** components.
Programming with Components
Gives links to topics about working with components and authoring your own controls and components.
Component Authoring Walkthroughs
Provides a list of detailed walkthroughs related to component design and use.

Introduction to Monitoring and Managing Windows Processes

You can use the **Process** component to accomplish most Windows process management tasks quickly and easily. You can use this component to work with processes on either local or remote computers. On a local computer, you can start and stop a process and also query it for specific types of information, including the names of the modules it has loaded, the time it was started, the number of handles it is consuming, and the set of its threads. On a remote computer, you cannot start or stop a process, but you can query existing processes for information.

To work with processes in your classes, you add an instance of the **Process** component to the class. This enables your class to access processes on the servers in your enterprise, and also to start and stop new processes. Other components in your application can call the **Process** component if it is necessary for them to start a new process or manipulate an existing process.

The PerformanceCounter and EventLog components may be better suited for monitoring the system performance. For more information, see Monitoring Performance Thresholds and Logging Application, Server, and Security Events.

Restrictions

Security issues require you to have sufficient access rights on the target computer to run code before attempting to stop processes or retrieving process property information. Any attempt to perform these tasks without the proper access authorization will result in an exception error.

See Also

Securability | Creating and Binding to Processes | Binding to Existing Processes | Creating Process Components | Starting Processes | Stopping Processes | Monitoring Performance Thresholds | Creating and Removing Custom Event Logs

Creating and Binding to Processes

You can use the **Process** component to bind to any running process to which you have rights. After the component has been bound to a target process, you can perform several tasks including stopping the process and viewing process statistics. For example, if a **Process** component has been bound to a process, it will store the starting and stopping time of the target process. You can take this information from the component's properties and assigned them to variables for later use in maintenance applications.

In This Section

Creating Process Components
Gives directions for creating **Process** components.
Binding to Existing Processes
Explains how to bind components to existing processes.

Related Sections

Starting Processes
Describes how to use the **Process** component to start processes on your computer.
Stopping Processes
Describes how to use the **Process** component to stop processes currently running on your computer.
Retrieving Information About Processes
Explains how to use the **Process** component to retrieve property information about processes on your computer.

Creating Process Components

You create instances of the **Process** component to act as a connection between your application and the processes on a computer. Once you bind a **Process** component instance to an individual process, you can use it to stop or retrieve information about the target process. You can use these methods to create applications that monitor and maintain individual processes or perform actions on a group of processes.

To create an instance of the Process component from the Toolbox

1. Drag the **Process** component from the **Components** tab of the Toolbox to the project.

   The component will reside in the component tray of the designer, located at the bottom of the design surface, and will not be visible to the end user.

2. Right-click the component and select **Properties** from the shortcut menu.
3. Expand the **StartInfo** section of the component's property page and set any of the starting properties you require. For more information, see ProcessStartInfo Members.

To create an instance of the Process component programmatically

1. Create an instance of the **Process** component class in code.

```
' Visual Basic
Dim myProcess as New Process()

// C#
using System.Diagnostics;
Process myProcess = new Process();
```

2. Call **Start** to start a new process. For more information, see Starting Processes.

See Also

ProcessStartInfo Members | Starting Processes | Stopping Processes |
Introduction to Monitoring and Managing Windows Processes

Binding to Existing Processes

You can use the **Process** component as a connection between your application and the processes on your computer. You bind to an existing process by using the GetProcessesByName method to return an array of processes (in the form of Process component instances) associated with each running instance of the specified file name. The first element of the array will be the first instance of the running process that the method finds. If there is only one instance of the process, the index of 0 can be used to reference the single instance of the process. You can use each of these returned instances to manipulate the processes with which they are associated.

To bind to an existing process

1. Create an instance of the **Process** component. For more information, see Creating Process Components.
2. Declare an empty array of the type **Process** to hold the array of **Process** components.

```
' Visual Basic
Dim myProcesses() as Process

// C#
Process[] myProcesses;
```

3. Fill the process array by calling the GetProcessesByName or GetProcessById methods and use the returning value to populate the process array.

   The following example shows how to call the **GetProcessesByName** method to add Notepad.exe to the process array.

```
' Visual Basic
myProcesses = Process.GetProcessesByName("Notepad")

// C#
myProcesses = Process.GetProcessesByName("Notepad");
```

4. Use the indexed property value to manipulate a single process in the array.

```
' Visual Basic
' Closes the first instance of Notepad in the process array.
myProcesses(0).CloseMainWindow()

// C#
// Closes the first instance of Notepad in the process array.
myProcesses[0].CloseMainWindow();
```

See Also

Starting Processes | Stopping Processes | Retrieving Information About Processes | Introduction to Monitoring and Managing Windows Processes

Managing Processes

You can use the **Process** component to start and stop processes and to retrieve information about your running processes on a computer. For example, you might use an instance of the **Process** component to determine whether a process had stopped responding. If the value returned true, you could have the component force the process to stop or prompt the user with the information and offer a choice of options.

You can use the **Process** component to create applications that allow you to view and manipulate processes and their property values. The topics in this section cover the various tasks associated to managing process using the **Process** component.

In This Section

Starting Processes
Gives instructions on how to use the **Process** component to start processes.
Stopping Processes
Explains how to use the **Process** component to stop running processes.
Determining If a Process Is Responding
Gives instructions on how to use the **Process** component to determine if a process has stopped responding to user input.
Determining If a Process Has Exited
Describes how to use the **Process** component to determine if a process has exited.
Viewing Running Processes
Gives instructions on how to use the **Process** component to view a list of all processes currently running on a computer.
Waiting for Processes to Complete Actions
Describes how to find out if a process has completed an action.

Related Sections

Introduction to Monitoring and Managing Windows Processes
Gives an introduction to working with Windows processes.
Adding Controls Without a User Interface to Windows Forms
Discusses components and controls, such as the **Process** component, that do not have a user interface.
Class vs. Component vs. Control
Gives an overview of the new component structure of the .NET Framework.
Walkthrough: Managing a Windows Process
Gives step-by-step instructions for managing Windows processes.

Starting Processes

You can use the **Process** component to start processes on your system by calling the Start method. Before you call Start, you must specify the file name of the process to start by setting the FileName property to either the fully qualified path to the target process, or in the case of qualified Windows applications such as Notepad, simply the process name.

You can set the **FileName** property at design time, using the Properties window, or at run time by using a value of the **StartInfo** enumeration. If you set the file name at run time, you can do one of the following:

- Set the appropriate value of the **StartInfo** enumeration and then call **Start**, or
- Call the static form of the **Start** method and specify **FileName** as a parameter. Use this approach if you do not need to set any other starting parameters; you cannot set any other opening arguments in this method.

To start a process at run time using the StartInfo properties

1. Set the starting information by exposed by the **StartInfo** property.
2. Call the **Start** method of the **Process** component.

The following example shows how to open Notepad in a maximized window.

```
' Visual Basic
Dim myProcess as New Process()
myProcess.StartInfo.FileName = "Notepad.exe"
myProcess.StartInfo.WindowStyle = ProcessWindowStyle.Maximized
myProcess.Start()

// C#
Process myProcess = new Process();
myProcess.StartInfo.FileName = "Notepad";
myProcess.StartInfo.WindowStyle = ProcessWindowStyle.Maximized;
myProcess.Start();
```

To start a process at run time by passing the FileName parameter

- Call the **Start** method, entering the **FileName** parameter as a string expression.

```
' Visual Basic
myProcess.Start("Notepad.exe")

// C#
Process myProcess = Process.Start("Notepad");
```

See Also

Creating Process Components | Managing Processes | Introduction to Monitoring and Managing Windows Processes

Stopping Processes

There are two methods you can use to stop a process with a **Process** component. The method you use depends on the type of process being stopped:

- If the process has a graphical user interface, call the CloseMainWindow method. This method sends a close request to the main window of the process and behaves in the same manner as selecting the Close command from the user interface. Using this method gives the target program a chance to prompt the user to save any unsaved data during the cleanup operation.
- If the process does not have a user interface, call the Kill method.

> **Caution** Calling the **Kill** method will stop the process immediately without prompting to save changed data. Any unsaved data will be lost.

If you want the component to be notified when the operating system has shut down a process, you must set the **EnableRaisingEvents** property to **true**. The **EnableRaisingEvents** property is used in asynchronous processing to notify your application that a process has exited. For more information, see Process.EnableRaisingEvents Property.

To stop a process

1. Call the GetProcessesByName method to retrieve the process you want to stop.

   For more information, see Binding to Existing Processes.

2. Call one of the following methods:

   - If the process has a user interface, call the **CloseMainWindow** method.
   - If the process is windowless, call the **Kill** method.

   The following example shows how to call the **CloseMainWindow** method to close all instances of Notepad currently running on a local computer:

```
' Visual Basic
Dim myProcesses() as Process
Dim myProcess as Process
' Returns array containing all instances of "Notepad".
myProcesses() = Process.GetProcessesByName("Notepad")
For Each myProcess in myProcesses
   myProcess.CloseMainWindow()
Next

// C#
Process[] myProcesses;
// Returns array containing all instances of Notepad.
myProcesses = Process.GetProcessesByName("Notepad");
foreach(Process myProcess in myProcesses){
    myProcess.CloseMainWindow();
}
```

See Also

Managing Processes | Binding to Existing Processes | Introduction to Monitoring and Managing Windows Processes

Determining If a Process Is Responding

You can use the Responding property to determine if the user interface of a process is responding. When you attempt to read the **Responding** property, a request is sent to the user interface of the target process. If there is an immediate response, the return property value is **true**; a **false** property value is returned if there is no response from the interface. This property is useful if you need to force a frozen application to close.

To determine if a process is responding

1. If the process was not started with a component, associate a **Process** component to the target process. For more information, see Binding to Existing Processes.
2. Read the **Responding** property by calling the **Responding** method.
3. Determine a course of action to take depending on the property value.

The following example shows how to determine if Notepad is responding. If the **Response** property is **true,** call the **CloseMainWindow** method to close the application. If the **Response** property is **false**, the **Kill** method is called to force the process to close.

```
' Visual Basic
Dim myProcesses() as Process
myProcesses = Process.GetProcessesByName("Notepad.exe")
' Tests the Responding property for a True return value.
If myProcesses(0).Responding Then
   myProcesses(0).CloseMainWindow()
Else
   ' Forces the process to close if the Responding value is False.
   myProcesses(0).Kill()
End If

// C#
Process[] notepads;
notepads = Process.GetProcessesByName("Notepad.exe");
// Test to see if the process is responding.
if (notepads[0].Responding)
{
   notepads[0].CloseMainWindow();
}
else
{
   notepads[0].Kill();
}
```

See Also

Retrieving Information About Processes | Starting Processes | Managing Processes | Stopping Processes | Binding to Existing Processes

Determining If a Process Has Exited

You can use the HasExited property to determine if the process associated to a **Process** component has stopped running. The property value returns **true** if the process is closed and **false** if the process is still running.

> **Note** This value is only returned for processes that are started by a **Process** component.

The **Process** component does not need to close the associated process to obtain the **HasExited** property. Administrative information such as the **HasExited** and **ExitTime** properties are stored regardless of the way the associated process was closed. The information is stored even if the user closes the process by selecting the Close command in its interface. This information is useful if you want to make sure all processes started with your **Process** components close when you exit an application.

To determine if a process has exited

- Read the **HasExited** property of the **Process** component that was used to open the process.

The following example shows how to use the **HasExited** property to determine if the process associated with a **Process** component called Process1 has been closed. If it is open, call **CloseMainWindow** to close the application.

```
' Visual Basic
If Not Me.notepad.HasExited Then
   ' If the process is still running, close it.
   Me.notepad.CloseMainWindow
End If


// C#
if (!notepad.HasExited) {
   notepad.CloseMainWindow();
}
```

See Also

Starting Processes | Managing Processes | Stopping Processes | Retrieving Information About Processes

Viewing Running Processes

When working with processes on a system, it is sometimes necessary to view all processes that are running at a given time. For example, if you want to create an application that gives you the functionality of stopping processes, you must first see which processes are running. You could fill a list box with the process names and select which process to perform any other actions on.

To view running processes

1. Declare an empty array of the type **Process**.
2. Fill the empty array with the return value from the GetProcesses method.
3. Iterate through the process array using the indexed value to obtain the process name of each process in the array and write it to a console.

The following example shows how to call the **GetProcesses** method of a **Process** component to return the process array and writes the **ProcessName** value to a console.

```
' Visual Basic
Dim myProcesses() as Process
Dim myProcess As Process
myProcesses = Process.GetProcesses()
' Iterate through the process array.
For Each myProcess in myProcesses
   Console.WriteLine(myProcess.ProcessName)
Next

// C#
Process[] myProcesses = Process.GetProcesses();
foreach(Process myProcess in myProcesses)
{
   Console.WriteLine(myProcess.ProcessName);
}
```

See Also

Binding to Existing Processes | Managing Processes | Stopping Processes | Starting Processes

Waiting for Processes to Complete Actions

A process is said to be idle when its main window is waiting for input from the system. In order to test the process for its idle state, you must first bind a **Process** component to it. You can call the WaitForInputIdle method before having the target process perform an action.

The **WaitForInputIdle** method instructs a **Process** component to wait for the associated process to enter an idle state. The method is useful, for example, when your application waits for a process to finish creating its main window before communicating with that window. The **WaitForInputIdle** method only works with processes that have a user interface.

To wait for a process to complete an action

1. Associate an instance of a **Process** component to a process you want to start. For more information, see Starting Processes.
2. Start the process by calling the **Start** method.
3. Call the appropriate **WaitForInputIdle** method:

   - **WaitForInputIdle()** - instructs the **Process** component to wait indefinitely for the associated process to enter an idle state.
   - **WaitForInputeIdle(Int32) -** instructs the **Process** component to wait the specified number of milliseconds for an associated process to enter an idle state.

The following example shows how to call the **WaitForInputIdle** method to wait for Notepad to complete loading before assigning its modules property to an empty array.

```
' Visual Basic
Dim myProcess as New Process()
myProcess.Start("Notepad.exe")
myProcess.WaitForInputIdle()

// C#
Process myProcess;
myProcess = Process.Start("Notepad");
myProcess.WaitForInputIdle();
```

See Also

Binding to Existing Processes | Starting Processes | Stopping Processes |
Introduction to Monitoring and Managing Windows Processes

Retrieving Information About Processes

You can use the **Process** component to retrieve property information about a process. The component can retrieve several useful categories of information from the properties of the target component and store the values in its own cached properties.

In This Section

Investigating Memory Usage for Processes
Provides directions on using the **Process** component to return process memory usage.
Investigating Thread Usage for Processes
Provides directions on using the **Process** component to get the thread count used by a process.
Investigating Libraries Loaded by Processes
Provides directions on using the **Process** component to list libraries loaded by a process.
Refreshing Process Component Properties
Provides directions on how to refresh the **Process** component's properties.

Related Sections

Binding to Existing Processes
Provides directions how to bind a **Process** component to a process.
Creating Process Components
Provides directions how to create a **Process** component.
Managing Processes
Provides links to topics on starting and stopping processes, as well as retrieving property information.
Determining If a Process Has Exited
Provides directions on using a **Process** component to determine if a process has exited.
Determining If a Process Is Responding
Provides directions on using a **Process** component to determine if a process is responding.
Introduction to Monitoring and Managing Windows Processes
Describes essential concepts about the **Process** component.

Investigating Memory Usage for Processes

If you need to view memory statistics for a process, the **Process** component provides six memory-usage properties that can be accessed at run time. Each property provides a different statistic for memory allocation.

To investigate memory usage for a process

1. Bind a process component instance to the process. For more information, see Binding to Existing Processes.
2. If the property cache needs to be refreshed, call the **Refresh** method. For more information, see Refreshing Process Component Properties.
3. Read the memory-usage property you want by referencing the appropriate property.

Property	Returns
PrivateMemorySize	The number of bytes that the associated process has allocated that cannot be shared with other processes.
PeakVirtualMemorySize	The maximum amount of virtual memory that the process has requested.
PeakPagedMemorySize	The maximum amount of memory that the associated process has allocated that could be written to the virtual paging file.
PagedSystemMemorySize	The amount of memory that the system has allocated on behalf of the associated process that can be written to the virtual memory paging file.
PagedMemorySize	The amount of memory that the associated process has allocated that can be written to the virtual memory paging file.
NonpagedSystemMemorySize	The amount of memory that the system has allocated on behalf of the associated process that cannot be written to the virtual memory paging file.

The following example shows how to use the **Process** component to read the **PrivateMemorySize** property for Notepad and assign the returned property value to NotepadMemory. The value is then displayed in a console. Because Component1(0) is a new instance of the **Process** component, the property cache does not need to be refreshed.

```
' Visual Basic
Dim NotepadMemory as Integer
Dim component1() as Process
component1 = Process.GetProcessesByName("Notepad.exe")
NotepadMemory = component1(0).PrivateMemorySize
Console.WriteLine("Memory used: " & NotepadMemory & ".")

// C#
int memory;
Process[] notepads;
notepads = Process.GetProcessesByName("Notepad.exe");
notepadMemory = notepads[0].PrivateMemorySize;
Console.WriteLine("Memory used: {0}.", notepadMemory);
```

See Also

Retrieving Information About Processes | Binding to Existing Processes | Introduction to Monitoring and Managing Windows Processes

Investigating Thread Usage for Processes

You can view the process threads by reading the Threads property value of the **Process** component. The return value is a collection of the type **ProcessThread** that represents the operating system threads currently running in the process. You can then iterate through the array to view individual thread properties using the Array(Index) property. The primary thread is not necessarily the thread at index 0 of the array.

To investigate thread usage for a process

1. If the process was not started by a **Process** component, associate a **Process** component to the desired process. For more information see Binding to Existing Processes.
2. Assign the **Threads** property value of the process to an empty collection variable of the type **ProcessThread**.
3. Iterate through the array index to view properties for a single **Thread**.

The following example shows how to read the **Threads** property of Notepad and assign the value to an empty array. The **Thread.BasePriority** value of the first thread in the **ProcessThread** array is then read and displayed in a textbox named TextBox1.

```
' Visual Basic
Dim myCollection as ProcessThreadCollection
Dim myProcesses() as Process
' Create an instance of the Process Component and associate
' it to the target process.
myProcesses() = Process.GetProcessesByName("Notepad.exe")
' Read the Process.Threads property and assign it to the empty array.
myCollection = myProcesses(0).Threads
' Read desired ProcessThread property.
Me.Textbox1.Text = myCollection(0).BasePriority.ToString()

// C#
ProcessThreadCollection threads;
Process[] notepads;
// Retrieve the Notepad processes.
notepads = Process.GetProcessesByName("Notepad");
// Read the Process.Threads property.
threads = notepads[0].Threads;
// Read desired ProcessThread property.
This.Textbox1.Text = threads[0].BasePriority.ToString()
```

See Also

Introduction to Monitoring and Managing Windows Processes | Binding to Existing Processes | Retrieving Information About Processes | Managing Processes | Creating Process Components

Investigating Libraries Loaded by Processes

The **Modules** property of the **Process** component provides access to the loaded libraries for a process. The **Modules** property returns a collection of the type **ProcessModules**, which includes all loaded libraries for the target process. You can then iterate through the collection using the **Indexed** property to look at individual libraries.

To investigate library usage for a process

1. If the target process was not started by a **Process** component, bind a new instance of a **Process** component to the process. For more information, see Binding to Existing Processes.
2. Declare an empty array of type **ProcessModules** to hold the module collection.
3. Assign the **Modules** property to the **ProcessModules** variable. This populates the **ProcessModules** array with the modules from the target module.
4. Iterate through the **ProcessModules** array by using the **Array(Index)** property to view and manage individual libraries.

The following example shows how to return all loaded libraries for Microsoft Word and then pass the library name to a list box on Form1:

```
' Visual Basic
Dim WordProcesses() as Process
Dim WordModules as ProcessModuleCollection
Dim aModule as ProcessModule
WordProcesses = Process.GetProcessesByName("WinWord")
' Populate the module collection.
WordModules = WordProcesses(0).Modules
' Iterate through the module collection.
For each aModule in WordModules
   Form1.Listbox1.Items.Add(aModule.ModuleName)
Next

// C#
Process[] wordapps;
ProcessModuleCollection modules;
wordapps = Process.GetProcessesByName("WinWord");
// Populate the module collection.
modules = wordapps[0].Modules;
// Iterate through the module collection.
foreach (ProcessModule aModule in modules)
{
   Form1.Listbox1.Items.Add(aModule.ModuleName);
}
```

See Also

Managing Processes | Retrieving Information About Processes | Creating Process Components | Binding to Existing Processes

Refreshing Process Component Properties

A **Process** component's properties are stored in a cache. The properties are used to store the property values of the process that the component is bound to.

The properties are not filled until you make the first property request from the process. At that time the component fills its cached properties with the property values of the process that it is bound to. The component does not renew this information when subsequent property value requests are made.

To view current property values, you must call the **Refresh** method of the **Process** component before requesting property values of the process. When you call the **Refresh** method, the values in the component's properties are replaced with the current data from the properties of the target process.

To refresh Process component properties

1. Create an instance of the process programmatically or by adding a **Process** component to the project in design view. For more information, see Binding to Existing Processes.
2. If any changes have occurred to the process properties prior to retrieving property information, call the **Refresh** method.
3. Set properties to read the appropriate process information and assign the returned value to a variable.

The following example shows how to open an instance of Notepad and then calls the **Refresh** method to update the component's properties. The code then reads the updated **WorkingSet** property of the component to return the total amount of memory allocated for the associated process. The updated property value is then written to a console screen six times in two-second intervals. The console stays open for five seconds after the procedure is finished.

```
' Visual Basic
Dim Notepad As New Process()
Dim myProcess As Process
Dim i As Integer
Notepad.Start("Notepad.exe")
For i = 0 to 5
   ' Forces the Process component to get a new set of property values.
   Notepad.Refresh
   ' Writes the property value to the console screen.
   Console.WriteLine(Notepad.WorkingSet)
   ' Waits two seconds before running the next loop.
   System.Threading.Thread.Sleep(2000)
Next
' Closes Notepad and waits 5 seconds before closing the console screen.
Notepad.CloseMainWindow()
System.Threading.Thread.Sleep(5000)

// C#
Process notepad;
notepad = Process.Start("Notepad");
for (int i = 0;i < 5; i++)
{
   // Forces the Process component to get a new set
   // of property values.
   notepad.Refresh();
   // Writes the property value to the console screen.
   Console.WriteLine(notepad.WorkingSet.ToString());
   // Waits two seconds before running the next loop.
   System.Threading.Thread.Sleep(2000);
}
// Closes Notepad and waits 5 seconds before closing
// the console screen.
notepad.CloseMainWindow();
System.Threading.Thread.Sleep(5000);
```

See Also

Binding to Existing Processes | Introduction to Monitoring and Managing Windows Processes | Viewing Running Processes | Monitoring and Managing Windows Processes

Walkthrough: Managing a Windows Process

The procedures in this topic walk you through the steps involved in creating a process, responding when a process stops, and stopping processes. In the first section, you will create a Windows application with **Button** controls to start and stop a **Notepad** process. You will start several instances of **Notepad** individually and then stop them as a group. In the second section, you will create a console application that enumerates the processes running on your computer.

For more information on using the **Process** component to interact with system processes, see Introduction to Monitoring and Managing Windows Processes.

To create your application

1. From the **New Project** dialog box, create a Visual Basic or Visual C# **Windows Application**.
2. With the Form1 designer open, click the **Windows Forms** tab in the Toolbox, and then add two buttons to the form.
3. In the Properties window, change the following properties:

Control	Property	Value
Button1	Name	**ButtonStart**
	Text	**Start Process**
Button2	Name	**ButtonStop**
	Text	**Stop Process**

4. Click the **Component** tab in the Toolbox, and then drag an instance of the **Process** component to the designer surface.
5. Name the component **myProcess**.

To start the Notepad.exe process

1. In the Properties window for the **Process** component instance, expand the **StartInfo** property, and set the **FileName** property to **notepad.exe**.
2. Double-click the **Start** button to access the Code Editor, and then add the following code to the **ButtonStart_Click()** event:

```
' Visual Basic
myProcess.Start

// C#
myProcess.Start();
```

3. Save all files, and then build and run the application.
4. Click the **Start Process** button a few times. You will see separate instances of Notepad for each click.

   Security Note If you are running in a partial-trust context, the process might throw a SecurityException Class exception due to insufficient privileges. For more information, see Code Access Security Basics.

5. Close the individual **Notepad** applications.
6. Close the **Form1** application.

To close all current instances of the Notepad.exe process

1. From Design view, double-click the **Stop** button to access the **ButtonStop_Click()** procedure.
2. Add the following code to close the current instances of Notepad.

```
' Visual Basic
Private Sub ButtonStop_Click(ByVal sender As System.Object, _
   ByVal e As System.EventArgs) Handles ButtonStop.Click
   Dim myProcesses() As Process
   Dim instance As Process
   myProcesses = Process.GetProcessesByName("Notepad")
   For Each instance In myProcesses
      instance.CloseMainWindow()
   Next
End Sub
```

```
// C#
private void ButtonStop_Click(object sender, System.EventArgs e)
{
   System.Diagnostics.Process[] myProcesses;
   myProcesses =
      System.Diagnostics.Process.GetProcessesByName("Notepad");
   foreach (System.Diagnostics.Process instance in myProcesses)
   {
      instance.CloseMainWindow();
   }
}
```

3. Save all files, and then build and run your application.
4. Start several instances of Notepad by clicking the **Start Process** button.
5. Click the **Stop Process** button to immediately close all running instances of Notepad.
6. Close the **Form1** application.

To configure the component to wait for the Notepad process to finish running

In this procedure, you will freeze your code until the process finishes.

1. Access the Code Editor for Form1.
2. Modify the code by adding a line of code to cause a pause before closing each window.

```
' Visual Basic
Private Sub ButtonStop_Click(ByVal sender As System.Object, _
   ByVal e As System.EventArgs) Handles ButtonStop.Click
   Dim myProcesses() As Process
   Dim instance As Process
   myProcesses = Process.GetProcessesByName("Notepad")
   For Each instance In myProcesses
      instance.WaitForExit(3000)
      instance.CloseMainWindow()
   Next
End Sub

// C#
private void ButtonStop_Click(object sender, System.EventArgs e)
{
   System.Diagnostics.Process[] myProcesses;
   myProcesses =
      System.Diagnostics.Process.GetProcessesByName("Notepad");
   foreach (System.Diagnostics.Process instance in myProcesses)
   {
      instance.WaitForExit(3000);
      instance.CloseMainWindow();
   }
}
```

3. Save all files, and then build and run the application.
4. Start several instances of Notepad by clicking the **Start Processes** button.
5. Click the **Stop Processes** button to close the processes as before.

   You will notice that the application now waits 3 seconds for each process to stop before closing the next instance of the process.

6. Close the **Form1** application.

In this portion of the walkthrough, you will create a new console application that will retrieve and list the processes on the local computer.

To enumerate the processes on your computer

1. Create a new **Console Application** project.
2. In Solution Explorer, right-click the project and select **Properties**.
3. On the **General** tab of the **Property Pages** dialog box, set the **StartupObject** property for your project to **Sub Main**. Note that the **StartupObject** in C# will be the class name which contains the **Static Main Method**.
4. Open the Code Editor for the module, and then add the following code to your module to enumerate the processes on your computer:

```
' Visual Basic
Shared Sub Main()
   Dim processes() As Process
   Dim instance As Process
   Dim process As New Process()
   processes = process.GetProcesses
   For Each instance In processes
      Console.WriteLine(instance.ProcessName)
   Next
   System.Threading.Thread.Sleep(5000)
End Sub


// C#
static void Main()
{
   System.Diagnostics.Process[] processes;
   processes = System.Diagnostics.Process.GetProcesses();
   foreach (System.Diagnostics.Process instance in processes)
   {
      Console.WriteLine(instance.ProcessName);
   }
   System.Threading.Thread.Sleep(5000);
}
```

5. Save all files, and then build and run your application.

   The application will open a console window containing a list of all processes currently running on your computer. Calling **Thread.Sleep** pauses the console for 5 seconds before closing it.

   > **Note** You may receive exceptions in these two procedures if you call **GetProcesses** or **GetProcessesByName** and one of the returned processes finishes before you make the next call. In that case, you will receive an exception saying that the process does not exist.

See Also

Managing Processes | Starting Processes | Stopping Processes

Creating Active Directory Components

The .NET Framework provides two Active Directory component classes: **DirectoryEntry** and **DirectorySearcher**. They are classes of the **System.DirectoryServices** namespace that use the Active Directory Services Interfaces (ADSI) technology. ADSI is the set of interfaces Microsoft provides as a flexible tool for working with a variety of network providers. ADSI gives the administrator the ability to locate and manage resources on a network with relative ease, regardless of the network's size. The topics in this section cover the various tasks and concepts involved in working with Active Directory objects.

In This Section

Introduction to Active Directory Objects
Provides an introduction to Active Directory components and requirements for using them.
Active Directory Technology Backgrounder
Summarizes key concepts for Visual Studio developers about the Windows Active Directory hierarchy and provides links to the SDK and other resources.
Creating DirectoryEntry Component Instances
Describes how to create instances of the **DirectoryEntry** component.
Modifying Object Properties in an Active Directory Hierarchy
Explains how to bind to an Active Directory object and modify its properties.
Creating Nodes in an Active Directory Hierarchy
Describes the steps involved in creating a node in an Active Directory hierarchy.
Removing Active Directory Nodes
Describes how to remove nodes and entire subtrees from Active Directory.
Searching Active Directory Hierarchies
Describes how to search The Active Directory hierarchy using the **DirectorySearcher** component.
Walkthrough: Adding Active Directory Objects
Demonstrates locating a resource on a network and iterating through its property values.

Related Sections

DirectoryEntry Class
Provides syntax and other reference information about the **DirectoryEntry** component.
DirectorySearcher Class
Provides syntax and other reference information about the **DirectorySearcher** component.
Directory Services
Provides information from the Platform SDK about Active Directory, ADSI, ADSI for Microsoft Exchange Server, the Lightweight Directory Access Protocol (LDAP) API, as well as a glossary.
Active Directory Services Interface (ADSI): Frequently Asked Questions
Provides answers to common questions about ADSI, including service providers supported.

Introduction to Active Directory Objects

Active Directory Service Interfaces (ADSI) is a programmatic interface for Microsoft Windows Active Directory. It enables your applications to interact with diverse directories on a network, using a single interface. Visual Studio .NET and the .NET Framework make it easy to add ADSI functionality with the **DirectoryEntry** and **DirectorySearcher** components. For an overview of Active Directory, see Active Directory Technology Backgrounder.

Using ADSI, you can create applications that perform common administrative tasks, such as backing up databases, accessing printers, and administering user accounts. ADSI makes it possible for you to:

- Log on once to work with diverse directories. The **DirectoryEntry** component class provides username and password properties that can be entered at runtime and communicated to the Active Directory object you are binding to.
- Use a single application programming interface (API) to perform tasks on multiple directory systems by offering the user a variety of protocols to use. The **DirectoryServices** namespace provides the classes to perform most administrative functions.
- Perform "rich querying" on directory systems. ADSI technology allows for searching for an object by specifying two query dialects: SQL and LDAP. For more information see Searching Active Directory Hierarchies.
- Access and use a single, hierarchical structure for administering and maintaining diverse and complicated network configurations by accessing an Active Directory tree.
- Integrate directory information with databases such as SQL Server. The **DirectoryEntry** path may be used as an ADO.NET connection string provided that it is using the LDAP provider.

Searching Directories

You can use the **DirectorySearcher** class to search and perform queries against an Active Directory hierarchy using the Lightweight Directory Access Protocol (LDAP). You can perform queries based on properties (Rich Query) of an object that is located in a large directory to find a specific object by one or more of its property values. For more information, see Searching Active Directory Hierarchies.

Administering Directories

The **DirectoryEntry** component can be bound to an object in the directory to perform administrative tasks, such as modifying properties or monitoring informational changes. For example, suppose that your company allows employees to place online orders for books needed in their jobs. Spending limits vary depending on the employee's position. When an employee places a book order, the company's tracking application can use a **DirectoryEntry** component to verify information about the employee's position and spending limits. If the limit is exceeded, the **DirectoryEntry** component can retrieve the e-mail address for the employee and his manager and can then send additional information to both of them.

Requirements

Both the **DirectoryEntry** component and **DirectorySearcher** component require that you have the ADSI SDK or ADSI runtime installed on your computer in order to create applications with their functionality. ADSI 2.5 is installed by default with Windows 2000 or Windows XP. If you are using a previous version of Windows, you can install the SDK yourself from the Microsoft Web site.

The following requirements apply to **DirectoryEntry** components:

- To make changes to property values of Active Directory objects, you must have administrative rights to the object you are bound to. For information about Active Directory user rights see your Active Directory documentation in the Windows 2000 help directory.

The following requirements apply to **DirectorySearcher** components:

- **DirectorySearcher** component instances require the LDAP provider to perform rich queries against an Active Directory hierarchy.
- You must have a directory service provider, such as Active Directory or Lightweight Directory Access Protocol (LDAP), installed on your computer for the ADSI APIs to be useful when conducting searches. For more information about the directory services you can use and how to install them, see the Microsoft Web site.

See Also

Walkthrough: Adding Active Directory Objects | Active Directory Technology Backgrounder |

Active Directory Technology Backgrounder

Active Directory is the Windows directory service that provides a unified view of complex networks. It reduces the number of directories and namespaces with which a developer must contend. To use the ADSI technology, client computers must have ADSI 2.5 or later installed on them. ADSI 2.5 is automatically installed with Windows 2000 and therefore does not require installation.

In a typical organization, there are often diverse types of directories in use. These may include Windows NT directories, application-specific directories, and directories of network resources. This variety can cause problems for end users, developers, and administrators. A user may encounter problems while trying to log on to multiple resources. An administrator may face difficulties in maintaining network resources on diverse systems. As a developer, you may need to write an application to seamlessly interact with several different directories.

> **Visual Basic Note** Although you could access Active Directory Services Interfaces (ADSI) in Microsoft Visual Basic version 6.0, programming for it involved a complex series of API calls. Now you can simply use the **DirectoryEntry** component.

Active Directory Basic Concepts

Active Directory provides a straightforward structuring of a complicated network of computers. An Active Directory system is arranged in a hierarchical tree. Each node represents a resource or service available on the network and contains a set of properties that can be retrieved and manipulated.

Directory Service

A directory is an information source used to store information about interesting objects. A telephone directory stores information about telephone subscribers. In a file system, the directory stores information about files. In a distributed computing system or a public computer network such as the Internet, there are many interesting objects: printers, fax servers, applications, databases, and other users. Users want to find and use these objects. Administrators want to manage how these objects are used.

A directory service differs from a directory in that a service is both the information source and the functionality that makes the information available to users. For example, a directory service stores information about objects such as users. That information can be used for identification when communicating with network resources, but also as the definition of where the object fits into the overall hierarchical scheme. *Scope* is the limits defined for searching the tree or subtrees. For more information about namespaces, scopes, containers, and domains, see About ADSI. For more information about the administration and data models, see Active Directory Architecture.

Active Directory Schema

Active Directory schemas define attributes for directory objects in much the same way that database schemas define structures for databases. This schema information is stored with an Active Directory hierarchy to help expedite member searches on large directories. For more information, see Active Directory Schema.

Security

Active Directory provides a variety of security tools such as Secure Sockets Layer and Kerberos encryption to create a flexible security environment for developers. For more information, see Active Directory Security.

Working with Active Directory in Visual Studio or the .NET Framework

The **DirectoryEntry** component makes it easy to access an object from a directory and work with its data and behaviors. When you give the **DirectoryEntry** component a valid directory path in an Active Directory hierarchy, it returns an ADSI COM object that you can manipulate. Examples of ADSI objects include users, computers, services, organizations of user accounts and computers, file systems, and file service operations.

You can use the **DirectoryEntry** component to work with directories in an Active Directory hierarchy in several ways:

- You can search an Active Directory hierarchy to find a particular service or object.
- You can encapsulate a node from a hierarchy and manipulate or query its properties.
- You can add new nodes to the hierarchy.

You can use the **DirectoryEntry** component to automate common administrative tasks, such as adding users and groups, managing printers, and setting permissions on network resources. You can use **DirectoryEntry** components to interact with any resource on any directory system in your enterprise.

ADSI and Active Directory Schema

Active Directory uses a schema to define what information about a network node is accessible. The **DirectoryEntry** and **DirectorySearcher** components use this information to conduct searches and view individual properties of nodes in the directory. Schemas are stored within an Active Directory hierarchy, making searching and accessing a large directory more efficient. Schemas provide a means to create global definitions for object types stored in the directory and are given useful names such as "user," and "computer". The type of information that you would access on a network computer would be different from that of a user in a domain.

The **DirectoryEntry** component also uses Active Directory schemas when creating new objects. You are required to specify a preexisting schema name that the object will be associated to. Microsoft provides example schemas with the ADSI SDK in the MMC folder of the Samples directory.

Providers and Objects

Each type of directory system you can access with the **DirectoryEntry** component has a specific directory protocol (called a *service provider*) that allows you to access and work with that directory's contents. ADSI was created to eliminate the difficulty of switching back and forth between these protocols by enabling developers to access all of the protocols through a single interface.

When you create an instance of a **DirectoryEntry** component, you specify a path for it that indicates the type of provider in use on the system you are accessing and the object to which you want to bind. The following table lists the service providers you can access and the identifiers for each.

Service provider	Path identifier
Windows 2000 or Windows XP	WinNT://path
Lightweight Directory Access Protocol (LDAP)	LDAP://path
Novell NetWare Directory Service	NDS://path
Novell Netware 3.*x*	NWCOMPAT://path

For more information about what constitutes a valid directory path in an Active Directory hierarchy for each type of provider, see Namespaces.

Each service provider makes available to you a different set of objects with associated data and behaviors that you can access and manipulate. These objects correspond to items and resources in the directory tree for that namespace. Many of the providers have objects in common. For example, all of the providers give you access to a group object that represents a group account, and a user object that represents a user account.

For the Windows NT service providers, you can access domains, computers, print queues, and sessions. For the LDAP provider, you can access organizations, locality, and Root DS Entry (rootDSE) objects. The Root DS Entry is a required set of operational attributes that the user can read to find out fundamental characteristics of the directory and the server. The rootDSE is required only for LDAP providers.

See Also

Introduction to Active Directory Objects | Creating DirectoryEntry Component Instances |
Walkthrough: Adding Active Directory Objects

Creating DirectoryEntry Component Instances

You create a **DirectoryEntry** or **DirectorySearcher** component instance when you want to interact with the contents of an Active Directory hierarchy in your application. For more information, see Introduction to Active Directory Objects. To use either the **DirectoryEntry** component or the **DirectorySearcher** component, you must add a reference to the **System.DirectoryServices** namespace to your application. For more information, see Adding and Removing References. For more information on creating DirectorySearcher components, see Searching Active Directory Hierarchies.

There are several ways you can create an instance of the **DirectoryEntry** component:

- You can drag an instance of the **DirectoryEntry** component from the Components tab of the Toolbox to a designer.
- You can create an instance of the **DirectoryEntry** component in code.

> **Note** Unlike many of the other components on the Components tab of the Toolbox, **DirectoryEntry** components are not represented in Server Explorer, so you cannot create instances of the component from there.

To create an instance of the DirectoryEntry component in a designer

1. Drag a **DirectoryEntry** object from the **Components** tab of the **Toolbox** to the design surface for your form or component.
2. In the **Properties** window, set the **Path** property for your component to the location in the Active Directory hierarchy for the resource to which you want to bind the component, using this syntax:

```
Provider://providerpath
```

Where	Is
Provider	An indicator of the service provider for the directory to be accessed. The provider identifiers are case-sensitive and must be typed as shown. Must be one of the following values: • LDAP:// • WinNT:// • NDS:// • NWCOMPAT://
ProviderPath	Depending on the type of service provider chosen, the parameters in the provider path vary. For more information, see ADSI System Providers.

For example, the following are valid paths for different types of providers:

```
LDAP://MyDom.fabrikam.com/CN=SomeCN,DC=DEV,DC=SOMEDC,DC=COM,O=Internet
LDAP://CN=SomeCN,DC=DEV,DC=SOMEDC,DC=COM,O=Internet
WinNT://MyDomain/Group
WinNT://MyDomain/MyComputer/aPrinter/
NDS://ndsServer/O=Internet/DC=COM/DC=SOMEDC/DC=DEV/CN=SomeCN
NWCOMPAT://binderyServer/ADivision
```

To create an instance of the DirectoryEntry component programmatically

- Create an instance of the **DirectoryEntry** class in code, passing the path for the object to which you want to bind as a parameter.

  The following example shows how to pass the provider path to your **DirectoryEntry** object:

```
' Visual Basic
Dim myDir As
   New System.DirectoryServices.DirectoryEntry("provider://path")

// C#
System.DirectoryServices.DirectoryEntry myDir =
```

```
    new System.DirectoryServices.DirectoryEntry("provider://path");
```

The following example shows how to create a **DirectoryEntry** component instance in code, initialize it to an LDAP path, and retrieve its name and GUID for display in a form element:

```
' Visual Basic
Dim strpath As String = "LDAP://DC=onecity,DC=corp,DC=fabrikam,DC=com"
Dim myDE As New System.DirectoryServices.DirectoryEntry(strpath)
Dim nameGUID As String
nameGUID = myDE.Name & myDE.Guid.ToString()

// C#
String strpath = "LDAP://DC=onecity,DC=corp,DC=fabrikam,DC=com";
System.DirectoryServices.DirectoryEntry myDE = new
   System.DirectoryServices.DirectoryEntry(strpath);
string nameGuid = myDE.Name + myDE.Guid.ToString();
```

See Also

Introduction to Active Directory Objects | Removing Active Directory Nodes | Searching Active Directory Hierarchies | Walkthrough: Adding Active Directory Objects

Modifying Object Properties in an Active Directory Hierarchy

You can use the **DirectoryEntry** class to retrieve a node from an Active Directory hierarchy and modify its property content. An example would be changing an employee's personal information after the employee moves to a different area of the organization. After you modify a node, you must commit your changes in order for them to be saved to the tree. For more information, see Introduction to Active Directory Objects.

Note You must have administrator rights to a directory in order to modify any of its information.

To modify an object in the Active Directory hierarchy

1. Create an instance of the **DirectoryEntry** component, binding it to the known object in the active directory hierarchy with the object's path. For more information, see Creating DirectoryEntry Component Instances.

```
' Visual Basic
Dim User As New System.DirectoryServices.DirectoryEntry( _
   "LDAP://CN=nate sun, CN= users, DC=Sales, DC=CompanyA, DC=COM")

// C#
System.DirectoryServices.DirectoryEntry user =
   new System.DirectoryServices.DirectoryEntry(
   "LDAP://CN=nate sun/CN= users, DC=Sales, DC=CompanyA, DC=COM");
```

2. Make the necessary changes to the object's properties by referring to it by name.

   The following example shows how to change the office telephone number for an employee:

```
' Visual Basic
User.Properties("HomeNumber")(0) = "(425) 555-1212"

// C#
user.Properties["HomeNumber"][0]= "(425) 555-1212";
```

3. Call the CommitChanges method save your modifications to the item in the directory.

```
' Visual Basic
User.CommitChanges()

// C#
user.CommitChanges();
```

See Also

Introduction to Active Directory Objects | Searching Active Directory Hierarchies | Walkthrough: Adding Active Directory Objects | Removing Active Directory Nodes

Creating Nodes in an Active Directory Hierarchy

The Active Directory hierarchy can be comprised of several thousand nodes. Each node represents an object, such as a network printer or a user in a domain. Corporate networks constantly change as new employees are hired and objects such as network printers and computers are added. ADSI technology provides ways to programmatically add these objects to the directory tree.

To create a directory entry in the hierarchy, you must use the Children property of the **DirectoryEntry** component. The **Children** collection provides an **Add** method that allows you to add a node to the **Children** collection directly below the parent node that you are currently bound to. To do this, you must specify a name for the new node and the name of a schema template that you wish to associate with the node. For example, you may want to use a schema titled "computer" to add new computers.

To create a node in an Active Directory hierarchy

1. Bind an instance of the **DirectoryEntry** object to the parent directory and call the **Add** method to add a child to the directory. For more information, see Creating DirectoryEntry Component Instances.

```
' Visual Basic
Dim entry As New DirectoryServices.DirectoryEntry("WinNT://DomainName")
entry.Children.Add("NewEntryName", "SchemaNametoUse")

// C#
System.DirectoryServices.DirectoryEntry entry =
   new System.DirectoryServices.DirectoryEntry("WinNT://DomainName");
entry.Children.Add("NewEntryName","NameofSchemaTemplate");
```

See Also

Introduction to Active Directory Objects | Searching Active Directory Hierarchies | Walkthrough: Adding Active Directory Objects | Removing Active Directory Nodes | Modifying Object Properties in an Active Directory Hierarchy

Removing Active Directory Nodes

The **DirectoryEntry** component gives the administrator the ability to remove objects within a directory or entire directory subtrees from the hierarchy.

- To remove a child entry from children collection of the current parent directory, call the Remove method. The **Remove** method takes only one argument, the entry name to be removed, and must be used only on an empty container or an object which is not used as a container for a directory tree, such as printers or computer nodes.
- To delete an entry and all of its children, call the DeleteTree method. The **DeleteTree** method deletes the current entry and all of its subtree entries from the hierarchy.

To delete a child entry

- Bind an instance of the **DirectoryEntry** class to the parent directory and call the **Remove** method. For more information, see Creating DirectoryEntry Component Instances.

```
' Visual Basic
Dim entry As New DirectoryServices.DirectoryEntry("WinNT://DomainName")
Dim entryToRemove As DirectoryServices.DirectoryEntry
' Add code here to set entryToRemove to the entry you want to remove.
entry.Children.Remove(entryToRemove)

// C#
System.DirectoryServices.DirectoryEntry entry =
   new System.DirectoryServices.DirectoryEntry("WinNT://DomainName");
System.DirectoryServices.DirectoryEntry entryToRemove = null;
// Add code here to set entryToRemove to the entry you want to remove.
entry.Children.Remove(entryToRemove);
```

To delete an entire subtree

- Bind an instance of the **DirectoryEntry** class to the parent directory and call the **DeleteTree** method. For more information, see Creating DirectoryEntry Component Instances.

```
' Visual Basic
Dim entry As New DirectoryServices.DirectoryEntry("WinNT://DomainName")
entry.DeleteTree()

// C#
System.DirectoryServices.DirectoryEntry entry =
   new System.DirectoryServices.DirectoryEntry("WinNT://DomainName");
entry.DeleteTree();
```

See Also

Introduction to Active Directory Objects | Walkthrough: Adding Active Directory Objects | Modifying Object Properties in an Active Directory Hierarchy | Creating DirectoryEntry Component Instances

Searching Active Directory Hierarchies

You can use the **DirectorySearcher** class to search and perform queries against an Active Directory hierarchy using the Lightweight Directory Access Protocol (LDAP). LDAP is the only provider that allows directory searching. An administrator can make alter and delete objects found in the hierarchy. For more information, see Introduction to Active Directory Objects. When you instantiate the **DirectorySearcher** object, you pass it the root you want to search in the bound directory system, an optional filter specifying what you want to retrieve, and an optional list of properties to be retrieved. You can also set properties to determine:

- Whether to cache the search results on the local computer; set this using the **CacheResults** property. When set to **true**, directory information is stored on the local computer.
- How long to spend on the search; set this using the **TimeLimit** property.
- Whether to retrieve property names only or names and values together; set this using the **PropertyNamesOnly** property.
- Whether or not to perform a paged search in which results of a specific size are returned one at a time; set this using the **PageSize** property.
- The maximum number of entries to return by setting the **SizeLimit** property.

> **Note** If the maximum number of returned entries and **TimeLimit** properties exceed limitations set on the server, the server settings will override the component settings.

To search an Active Directory hierarchy

1. Create an instance of the **DirectorySearcher** object and initialize it to the **DirectoryEntry** value that you want to search. Create an instance of the **SearcherResultsEntry** object.
2. Set the **Filter** property of the **DirectorySearcher** object to the value that you want to search for.
3. Initialize the search by calling the **FindAll** method of the **DirectorySearcher**. The **DirectorySearcher** object will return a collection of all objects in the target directory whose values match those specified in its **Filter** property.

   The following example searches a directory for the user name "John" and then iterates through the returned collection, writing each object's path to a console. Note that this example will not search subdirectories.

```
' Visual Basic
Dim entry As New DirectoryServices.DirectoryEntry("LDAP://Path")
Dim mySearcher As New System.DirectoryServices.DirectorySearcher(entry)
Dim result As System.DirectoryServices.SearchResult
mySearcher.Filter = ("(anr= John)")
For Each result In mySearcher.FindAll()
   Console.WriteLine(result.GetDirectoryEntry().Path)
Next

// C#
System.DirectoryServices.DirectoryEntry entry =
   new System.DirectoryServices.DirectoryEntry("LDAP://Path");
System.DirectoryServices.DirectorySearcher mySearcher = new
   System.DirectoryServices.DirectorySearcher(entry);
mySearcher.Filter = ("(anr= John)");
foreach(System.DirectoryServices.SearchResult result in
   mySearcher.FindAll())
{
   Console.WriteLine( result.GetDirectoryEntry().Path );
}
```

See Also

Introduction to Active Directory Objects | Walkthrough: Adding Active Directory Objects | Removing Active Directory Nodes | Modifying Object Properties in an Active Directory Hierarchy

Walkthrough: Adding Active Directory Objects

The procedures in this topic demonstrate the process of using the **DirectoryEntry** component to list the users, groups, and services on your local computer. The **DirectoryEntry** component uses Active Directory technology to do this work. Each entry created by the component contains a list of its properties.

To create the user interface

1. On the **File** menu, point to **New**, and then click **Project**.
2. In the **New Project** dialog box, choose Visual Basic or Visual C# in the left pane, then choose the **Windows Application** template. Name the project **ActiveDirectory**.
3. From the **Windows Forms** tab of the Toolbox, drag a **TreeView** control onto Form1.
4. Set the **Name** property of the **TreeView** control to **viewPC**.

   The **TreeView** control will be modified later in the walkthrough to contain three top-level nodes, one each for users, groups, and services. Each second-level node will represent one user, group, or service registered on your computer. Each user, group, and service will have two child nodes, one for its Active Directory path and one for its properties.

To configure a DirectoryEntry component

1. In Solution Explorer, right-click your project and select **Add Reference** from the shortcut menu.
2. Add a reference to **System.DirectoryServices.dll**.
3. From the **Components** tab of the Toolbox, drag a **DirectoryEntry** component instance to the designer.
4. Set the **Name** property of the **DirectoryEntry** component to **entryPC**.
5. Set the **Path** property of the **DirectoryEntry** component to **WinNT:*//Domain/YourComputerName***. Use your domain and computer name to direct the **DirectoryEntry** component to examine your local computer using the WinNT service provider for Active Directory.

   Note If you do not know your domain or computer name, contact your system administrator.

   Tip If you have the incorrect domain or computer name, the application will throw an exception at run time. You can enter **WinNT:*//YourComputerName*** for the path, and the service provider will attempt to locate the computer.

To add the top-level nodes to the TreeView control

1. In the designer, double-click **Form1** to create the **Form_Load** event handler in the Code Editor.
2. Modify the **Form_Load** method by adding code to create the three top-level nodes for users, groups, and services. When the application runs, the **Children** property of the component named **entryPC** contains all the entries found in **Path** (your computer). Each child is also a **DirectoryEntry** object. Because you are using the WinNT provider on a computer path, the **SchemaClassName** property of the child **DirectoryEntry** object will be one of "User", "Group", or "Service". A select or switch statement can be used to determine the entry type and add a node to the correct group.

   You will write the **AddPathAndProperties** method in the next step.

```
' Visual Basic
Private Sub Form1_Load(ByVal sender As System.Object, _
ByVal e As System.EventArgs) Handles MyBase.Load
   Dim users As New TreeNode("Users")
   Dim groups As New TreeNode("Groups")
   Dim services As New TreeNode("Services")
   viewPC.Nodes.AddRange(New TreeNode() {users, groups, services})

   Dim child As System.DirectoryServices.DirectoryEntry
   For Each child In entryPC.Children
      Dim newNode As New TreeNode(child.Name)
      Select Case child.SchemaClassName
         Case "User"
            users.Nodes.Add(newNode)
         Case "Group"
```

```
            groups.Nodes.Add(newNode)
         Case "Service"
            services.Nodes.Add(newNode)
      End Select
      AddPathAndProperties(newNode, child)
   Next
End Sub

// C#
private void Form1_Load(object sender, System.EventArgs e)
{
   TreeNode users = new TreeNode("Users");
   TreeNode groups = new TreeNode("Groups");
   TreeNode services = new TreeNode("Services");
   viewPC.Nodes.AddRange(new TreeNode[] { users, groups, services });

   foreach(System.DirectoryServices.DirectoryEntry child
      in entryPC.Children)
   {
      TreeNode newNode = new TreeNode(child.Name);
      switch (child.SchemaClassName)
      {
         case "User" :
            users.Nodes.Add(newNode);
            break;
         case "Group" :
            groups.Nodes.Add(newNode);
            break;
         case "Service" :
            services.Nodes.Add(newNode);
            break;
      }
      AddPathAndProperties(newNode, child);
   }
}
```

3. Add the following code to the **Form1** class to create the `AddPathAndProperties` method. The `node` parameter represents the **TreeNode** object associated with that entry. The `entry` parameter represents one entry in the directory: a user, a group, or a service.

   The **DirectoryEntry** component has a **Properties** property that contains a string-indexed collection of properties. The contents of this collection depend on the schema of the **DirectoryEntry** component. The **Properties** collection has a **PropertyNames** property, which is a collection of all the names of the properties. By stepping through this collection, you can retrieve each property. Each property also has a collection of associated values. In this method, you only retrieve the first value for each property.

```
' Visual Basic
Private Sub AddPathAndProperties(ByVal node As TreeNode, _
ByVal entry As System.DirectoryServices.DirectoryEntry)
   node.Nodes.Add(New TreeNode("Path: " & entry.Path))
   Dim propertyNode As New TreeNode("Properties")
   node.Nodes.Add(propertyNode)

   Dim propertyName As String
   Dim oneNode As String
   For Each propertyName In entry.Properties.PropertyNames
      Try
         oneNode = propertyName & ": " & _
            CType(entry.Properties(propertyName)(0), String)
```

```
        Catch
           oneNode = propertyName & ": " & _
              "No text representation."
        End Try
        propertyNode.Nodes.Add(New TreeNode(oneNode))
     Next
  End Sub

  // C#
  private void AddPathAndProperties(TreeNode node,
     System.DirectoryServices.DirectoryEntry entry)
  {
     node.Nodes.Add(new TreeNode("Path: " + entry.Path));
     TreeNode propertyNode = new TreeNode("Properties");
     node.Nodes.Add(propertyNode);
     foreach (string propertyName in entry.Properties.PropertyNames)
     {
        string oneNode = propertyName + ": " +
           entry.Properties[propertyName][0].ToString();
        propertyNode.Nodes.Add(new TreeNode(oneNode));
     }
  }
```

To run the program

1. Press F5 to run the program.
2. Open the nodes to examine the paths and properties. The users all share the same set of properties, because they all share the same schema. The same is true of the group and service nodes.

Next Steps

This walkthrough used the WinNT service provider for Active Directory. Other services are available, including the Lightweight Directory Access Protocol (LDAP), the Novell NetWare Directory Service (NDS), and the Novell Netware 3.x service (NWCOMPAT). Each provider provides a different set of objects to allow you to examine and manipulate directories.

The **DirectorySearcher** component can be used to search for entries from a root path. The **DirectorySearcher** works with the LDAP provider.

See Also

Creating Active Directory Components | Active Directory Technology Backgrounder | DirectoryEntry Class | DirectorySearcher Class

Creating Installation Components

Installation components allow you to set up custom actions that will occur when your application is deployed. You do this if your application requires additional system resources — such as databases, configuration files, Windows event logs, message queues, or services — to be set up and configured in order to work correctly.

In This Section

Introduction to Installation Components
Describes the purpose of installation components, the types of installation components that are available to you, and the classes they are represented by in the **System** namespace.
Framework Component Deployment and the Design Process
Describes how installation components work and the general design process to use in your applications that require them.
Installation Process
Provides information on what happens during the installation process.
State Storage in the Installation Process
Explains the usage of the **Install**, **Commit**, **Rollback**, and **Uninstall** methods and how state is maintained throughout the installation process.
Creating Installation Components - High-Level Process
Lists the high-level steps in the process of adding installation components to your application and configuring them to behave correctly.
Walkthrough: Installing an Event Log Component
Presents an example of creating and configuring an instance of the **EventLog** component, adding the necessary installation component to deploy it, and deploying and testing your project.

Related Sections

Walkthrough: Using a Custom Action to Create a Database During Installation
Presents an example of creating a database during installation using an installation component and a custom action.
Adding Installers to your Service Application
Provides directions for creating the installation components that are necessary to install, start, and run your service.
Installing and Uninstalling Services
Gives instructions for installing your service and registering it with the system.
Adding Installation Components to Your Projects
Explains how to add the necessary installation components to your application.
Configuring Installation Components
Explains how to change the default values for an installation component to tailor its behavior.
Overriding Default Methods on Installation Components
Explains how to change the default functioning of the **Install**, **Commit**, **Rollback**, and **Uninstall** methods.

Introduction to Installation Components

In Microsoft Visual Studio 6.0, application setup typically involved copying all of the run-time and support files associated with an application from the development computer to the target computer. In Visual Studio .NET, an application consists not only of the traditional program files but also of associated resources, such as message queues, event logs, performance counters, databases, and configuration files, that must be created on the target computer, as well. You can configure your application to create these resources when your application is installed and to remove them if your application is uninstalled, using what are called *installation components*.

For example, suppose you are deploying an application that writes to a custom performance counter. The counter must be installed when the application is deployed. Using an installation component, you can set up the application so that it automatically creates and configures the necessary performance counter on a remote computer when the application is installed.

There are two types of installation components: *installer classes* that can be added to your project, and *pre-defined installation components* that are associated with components.

Installer classes can be added to any application to perform actions during installation (for example, creating a database on a target computer). In a deployment project, the compiled installer classes from your project are added as custom actions that are run at the end of installation. Deployment projects also have the ability to capture user input during installation and pass it to the installer class using the **CustomActionData** property — for example, you might want to let the user choose a location for a database. For more information on consuming installation components in deployment projects, see Walkthrough: Using a Custom Action to Create a Database During Installation.

Predefined installation components are associated on a one-to-one basis with the components that need them. For example, if you have an instance of the **MessageQueue** component in your project and it requires your deployed application to have a queue installed and configured, you create an installation component to handle the task. If you then add an instance of the **EventLog** component to your project and want to configure the event log in use, you must add a second installation component to perform that processing.

There are five predefined installation components that you can use in your projects:

- The **EventLog** installer (based on the **System.Diagnostics.EventLogInstaller** class) allows you to install and configure a custom event log that your application needs in order to run. For more information, see Introduction to the EventLog Component.
- The **MessageQueue** installer (based on the **System.Messaging.MessageQueueInstaller** class) allows you to install and configure a queue that your application needs in order to run. For more information, see Creating Messaging Components.
- The **PerformanceCounter** installer (based on the **System.Diagnostics.PerformanceCounterInstaller** class) allows you to install and configure a custom performance counter that your application needs in order to run. For more information, see Monitoring Performance Thresholds.
- The **Service** and **ServiceProcess** installers (based on the **System.ServiceProcess.ServiceInstaller** and **System.ServiceProcess.ServiceProcessInstaller** classes) are used together to allow you to install service applications. For more information, see Monitoring Windows Services.

All of the installation components in a project are stored within classes marked with the **RunInstallerAttribute=True**. By default, a class named **ProjectInstaller** that contains this attribute is added to your project when you add a predefined installation component; installer classes added via the **Add New Item** dialog box also contain this attribute and are named **Installer1** by default. For more information on what happens during the installation process, see Installation Process.

See Also

Framework Component Deployment and the Design Process | Installation Process | Introduction to the EventLog Component | Introduction to Messaging | Monitoring Performance Thresholds | Monitoring Windows Services | Walkthrough: Installing an Event Log Component

Framework Component Deployment and the Design Process

When you create framework components, such as **MessageQueue** and **EventLog** components, the standard development practice assumes that you will first create the necessary system resources on a test computer, using Server Explorer, and then create your components and link them to that test resource.

For example, suppose you are building an order entry system that uses a queue called OrdersReceived to process user data. As your first step, you would create the OrdersReceived test queue and set all of the necessary properties to configure it in a way appropriate for your application. You might set properties to determine the maximum queue size and turn on journal recording for that queue.

After creating the OrdersReceived queue on a test computer, you would create your project and add an instance of the **MessageQueue** component to it, linking it to the queue. You would then write your code, compile, and test the project without having to deploy it to the final server on which it will reside.

When you are ready to deploy your project, you move it from the test server to its final location on a production server, where the necessary resources that the application used on your test machine need to be recreated. Rather than manually configuring these resources on the new server, you can use installation components to do this work for you. When you add an installation component to your project, it accesses the existing test resources and copies the properties you set when you configured the resources, so that the same resource can later be recreated on the production server.

For example, in the case of the messaging application described above, the installation component for your queue would access the **MessageQueue** component instance and learn from its properties that it accessed a queue called OrdersReceived, on your test server. The installation component would then access the OrdersReceived queue and copy its configuration properties, including the **Path** and **Label**, the journal settings you made, and the maximum queue size. These values are stored in the code for the installation component. When you compile and deploy your project, the necessary resource is created on that computer.

You do not have to follow this design process, but it provides an efficient way to work with your test and production resources. If you do not create and configure a test resource from which property values can be copied to the installation component, you can access the installation component in the **ProjectInstaller** class (or any class with the **RunInstallerAttribute** value set to **true**) and manually set the necessary values to create and install the resource in the state you desire.

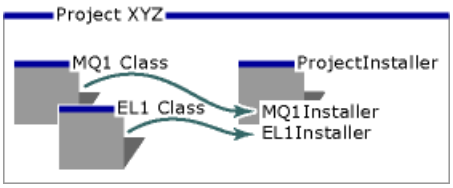
See Also

Introduction to Installation Components | Installation Process

Installation Process

When you create installation components for your Framework Services components, a default class called **ProjectInstaller** is created in your project. This file contains all of the installation components you create for that project. The following figure shows this process.

Installation Components and Project

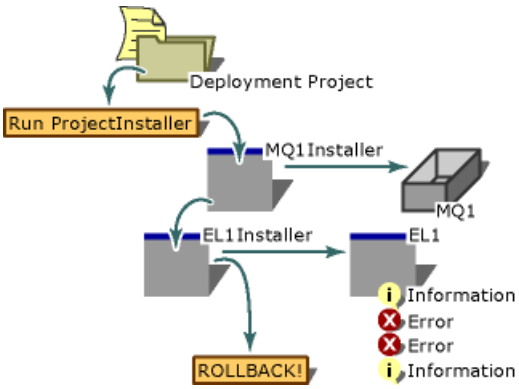


After you create an installation component and its associated **ProjectInstaller** class for your project, you compile the project into a DLL or EXE. You then can do one of two things:

- You can use the **installutil.exe** tool to install that DLL on your local computer.
- You can create a deployment package for your application and add your project's DLL to it as a custom action. When deployment occurs, the deployment project recognizes that it needs to run the **ProjectInstaller** class, and it launches the installation process for your component resources.

Resource installation is transactional. That is, the installation processes through each installer in your **ProjectInstaller** class (or any class with the **RunInstallerAttribute** value set to **true**) and makes sure that no errors occur before committing the installation as final. If any errors occur along the way, all of the resources that have previously been installed are uninstalled in a rollback transaction. Rollback is performed in the reverse order of install. The following figure shows this process.

Installation Process



In this picture, the **ProjectInstaller** class is installing two component resources — a message queue and an event log. The message queue installation succeeds without error, so processing is passed on to the event log installation. An error is encountered in this log, so a rollback begins. During the rollback, the message queue is uninstalled and the system is returned to its original state. For more information on how the **Rollback** method works, see State Storage in the Installation Process.

For a list of the overall steps in creating and testing an installation component, see Creating Installation Components - High-Level Process.

See Also

Introduction to Installation Components | State Storage in the Installation Process | Creating Installation Components - High-Level Process

State Storage in the Installation Process

During installation of components, the **ProjectInstaller** class (or any class with the **RunInstallerAttribute** value set to **true**) writes a file with the extension .**InstallState** to the deployment computer. This file is used to store information about the original state of the system before your resources were installed, and the changes that were made during installation. This file is used in each of the installation component's methods:

- The **Install** method accesses a blank copy of this file in the **stateSaver** parameter, and uses it to write information about all of the successful installations it performs.
- The **Commit** method receives information from the file in the **savedState** parameter. You can work with the contents of this file using the **IDictionary** object if you want to.
- The **Rollback** method receives information from the file in the **savedState** parameter, and uses it to undo previous transactions if an error occurs in the **Install** method. After rollback completes, the state file is deleted.
- The **Uninstall** method receives information from the file in the **savedState** parameter and uses the file to return the system to its original state.

The **InstallState** file is saved on the deployment computer to the same directory in which your project's DLL is installed. You should not modify this file directly, but you can use the **IDictionary** object to work with its contents.

See Also

Introduction to Installation Components

Creating Installation Components - High-Level Process

These are the overall steps in creating and testing an installation component for one of your resource-dependent components. See the procedures referenced in this section for more details on each step.

To create an installation component

1. For each component in your application that needs a system resource installed during deployment, add an installation component to your application. For details, see Adding Installation Components to Your Projects.
2. In the **ProjectInstaller** class for your project, customize any properties or methods for which you want to override the defaults. For details, see Configuring Installation Components or Overriding Default Methods on Installation Components.

   > **Note** If you are working in the Microsoft .NET Framework SDK, you must instantiate the component installers in the constructor for the **ProjectInstaller** class and add them to the list of components to install.

3. Save and compile your application project.
4. Create the deployment project for your project, and add your project's DLL to the Custom Action Editor. For details, see Custom Actions Management in Deployment.
5. Save and compile your deployment project.
6. Deploy the project to the destination computer, and verify that your resource was created correctly. For details, see Deploying Applications and Components.

   > **Note** You can also use the Microsoft Visual Studio .NET tool **installutil.exe** to deploy the installation of the DLL you created in step 3, rather than creating and testing the deployment package.

See Also

Adding Installation Components to Your Projects | Configuring Installation Components | Overriding Default Methods on Installation Components

Adding Installation Components to Your Projects

Installation components are used to perform actions during the installation of an application (for example, creating and configuring resources such as message queues or databases). There are two types of installation components: installer classes that can be added to any project, and predefined installation components that are associated with specific components. The procedure for adding installation components differs for each type.

To add an installer class to your project

1. In Solution Explorer, select the project where you want to add the class. On the **Project** menu, choose **Add New Item**.
2. In the **Add New Item** dialog box, select **Installer Class**.
3. Override any methods for which you need to perform custom processing. For details, see Overriding Default Methods on Installation Components.

Microsoft Visual Studio .NET ships predefined installation components that can install resources associated with your **MessageQueue** components, **EventLog** components, Windows service applications, and **PerformanceCounter** components. When you work with an instance of one of these components or application types, you can select a hyperlink in the Properties window to automatically add the appropriate installer to your project.

After you add the first installation component to your project, a class named **ProjectInstaller** appears in your project. Each subsequent installation component you place in the project is also added to that class. The **ProjectInstaller** class is automatically registered with the deployment project for your solution.

Visual Studio .NET automatically copies property values that are needed to create your components to the installation component. In some cases, you may have to modify the copied values. For more information, see Configuring Installation Components.

To add a predefined installation component to your project

1. In the designer, click the component for which you want to add an installation component, and then click the **Add Installer** link in the Properties window.

   > **Note** Predefined installation components are only available for **MessageQueue**, **EventLog**, and **PerformanceCounter** components, and for service applications. If you are working with a service application, you select the Component Designer surface rather than any particular component.

   A **ProjectInstaller** class appears in your project and an installation component is added to it.

2. If necessary, access the **ProjectInstaller** class in the Code Editor and modify any of the values that were copied to the installation component. For details, see Configuring Installation Components.
3. Override any methods for which you need to perform custom processing. For details, see Overriding Default Methods on Installation Components.
4. Continue to perform steps 1 through 3 for each installation component your project needs.

See Also

Configuring Installation Components | Overriding Default Methods on Installation Components

Configuring Installation Components

When an installation component is created for a component in your project, Visual Studio .NET copies property values that are necessary for the installer to recreate the chosen resource. For example, for an **EventLog** installer, the system copies the values of the **Log** and **Source** properties and uses these values during installation to create your log in the correct location.

The property values are copied either from the component or from the resource with which the component interacts on your test server. For example, suppose you have an instance of the **MessageQueue** component that interacts with a test queue called OrdersReceived on a server called server1. When you create an installer for this component, it copies the **Path** property's value from the component, and then uses this to locate the resource on the test machine. All of the properties set for that resource on the test machine are also copied to installation code. If there is no physical test resource, the system copies all of its values from the component, and you need to fill in the missing necessary properties manually in the **ProjectInstaller** class.

The following code shows how the default properties for this **MessageQueue** installer might look in the **ProjectInstaller** class:

```
' Visual Basic
Private WithEvents MessageQueueInstaller1 As System.Messaging.MessageQueueInstaller
Private Sub InitializeComponent()
   Me.MessageQueueInstaller1 = New System.Messaging.MessageQueueInstaller
   MessageQueueInstaller1.MaximumJournalSize = 4294967295&
   MessageQueueInstaller1.BasePriority = 0%
   MessageQueueInstaller1.Path = "server1\OrdersReceived"
   Me.Installers.Add(MessageQueueInstaller1)
End Sub

// C#
private System.Messaging.MessageQueueInstaller messageQueueInstaller1;
private void InitializeComponent()
{
   this.messageQueueInstaller1 = new System.Messaging.MessageQueueInstaller();
   messageQueueInstaller1.MaximumJournalSize = 42949672951;
   messageQueueInstaller1.BasePriority = 0;
   messageQueueInstaller1.Path = "server1\\OrdersReceived";
   this.Installers.Add(messageQueueInstaller1);
}
```

You can manually edit the property values if there is a property that you want to set differently on the deployment computer. In addition to changing default property values, you can also override some of the default methods in the installation component if you want to change the processing that occurs during installation. For more information, see Overriding Default Methods on Installation Components.

To change the default values for an installation component

1. After adding an installation component to your solution, open the **ProjectInstaller** class.
2. Locate the installation component you want to modify, and access it in the Code Editor.
3. Locate the **InitializeComponent** procedure. The default property values for your component are located within this procedure.
4. Make any changes you need to make to the property values that will be used to create your component.

See Also

Adding Installation Components to Your Projects | Overriding Default Methods on Installation Components

Overriding Default Methods on Installation Components

By default, every installation component has the following methods:

- **Install** — All of the installation functions the installer must perform are carried out within this method. The **ProjectInstaller** class (or any class with the **RunInstallerAttribute** value set to **true**) calls **Install** on each of the installation components it contains, and then calls either **Commit** (if the installations were all successful) or **Rollback** (if any errors occurred).
- **Commit** — **Commit** is called after the **Install** method has been successfully run on all of the installation components the installer class contains. At this point, all of the installations are committed. Usually, to avoid raising exceptions, no processing is done in the **Commit** method.
- **Rollback** — The **Rollback** method is used to undo all previous installation work if an error occurs anywhere within the installation process. If any of the installation components contained by the installer class raises an error, all of the installations carried out so far are undone.
- **Uninstall** — **Uninstall** is used to remove installation component resources from the system. Unlike installation, uninstall is not transactional. If one uninstall process fails, the system still attempts to uninstall all of the other resources.

Each of these methods contains default processing that will carry out the required work without modification. If desired, however, you can override and modify the processing in each step. For example, for a **Service** installation component, you might want to modify the default processing that occurs in the **Commit** procedure so that the installation process starts the service after it is successfully installed. You might also modify the **Uninstall** procedure so that each service is stopped before it is uninstalled.

To override default methods for an installation component

1. After adding an installation component to your solution, open the installer class.
2. Locate the installation component you want to modify, and access it in the Code Editor.
3. Override the appropriate method and add the functionality you want. For details, see Overriding Properties and Methods.

> **Tip** Make sure to retain the original processing necessary to install, commit, rollback, or uninstall your resources when you override methods in the **ProjectInstaller** class. In general, you should add code to the existing functions rather than removing existing functionality.

See Also

Adding Installation Components to Your Projects | Configuring Installation Components | Walkthrough: Installing an Event Log Component | Overriding Properties and Methods

Walkthrough: Installing an Event Log Component

The procedures in this topic walk you through the process of setting up an installation component for an instance of the **EventLog** component, including creating the component instance, adding an installer for it, and compiling and running the installer.

To create your EventLog component

1. On the **File** menu, point to **New**, and then click **Project**.
2. In the **New Project** dialog box, choose Visual Basic or Visual C# in the left pane, then choose the **Windows Application** template. Name the project **MyEventLog**.
3. From the **Components** tab of the Toolbox, drag an EventLog component to the form.
4. In the Properties window, set the following values:

   - Set the **Log** property to **Application**.
   - Set the **MachineName** property to the name of the server on which the event log resides. Use a period (**.**) for the local computer.
   - Set the **Source** property to the string of your choice. In this case, you can use the name of your project.

5. Save your files.

   Note For more information on creating **EventLog** component instances and setting their properties, see Logging Application, Server, and Security Events.

To create an installation component for your EventLog component

1. In the designer, click the **EventLog1** component.
2. In the **Description** area at the bottom of the Properties window, click the **Add Installer** link.

   A **ProjectInstaller** class appears in your project, and an installation component is added to it.

You can now install and run your application.

To build your installation

- Save your project and build it.

  Note Because this installer will function correctly without changing the existing methods, you do not need to override the **Install**, **Commit**, **Rollback**, and **Uninstall** methods.

A setup project will install the compiled project files and run the installers needed to run a Windows application.

To create a setup project for your application

1. On the **File** menu, point to **Add Project**, then click **New Project**.
2. In the **Project Types** pane, select the **Setup and Deployment Projects** folder.
3. In the **Templates** pane, select **Setup Project**. Name the project **MyEventLogSetup**.

   A setup project is added to the solution.

Next you will add the output from the Windows Service project to the setup.

To add the project output to the setup

1. In Solution Explorer, right-click **MyEventLogSetup**, point to **Add**, then choose **Project Output**.

   The **Add Project Output Group** dialog box appears.

2. **MyEventLog** is selected in the **Project** box.
3. In the list box, select **Primary Output**, then click **OK**.

   A project item for the primary output of **MyEventLog** is added to the setup project.

Now add a custom action to install the MyNewService.exe file.

To add a custom action to the setup

1. In Solution Explorer, right-click the setup project, point to **View**, then choose **Custom Actions**.

   The **Custom Actions** editor appears.

2. In the **Custom Actions** editor, right-click the **Custom Actions** node and choose **Add Custom Action**.

   The **Select Item in Project** dialog box appears.

3. Double-click the **Application Folder** in the list box to open it, select **Primary Output from MyEventLog (Active)**, and click OK.

   The primary output is added to all four nodes of the custom actions, **Install**, **Commit**, **Rollback**, and **Uninstall**.

4. Build the setup project.
5. To install **MyEventLog.exe**, right-click the setup project in the Solution Explorer and select **Install**.

See Also

Introduction to Installation Components | Launching Event Viewer | Logging Application, Server, and Security Events | System Monitoring Walkthroughs

Reference

This section links you to reference information on various aspects of Visual Basic and Visual C# programming.

In This Section

Visual Basic Language
Describes this programming language, including concepts, new features, and a complete Visual Basic Language and Run-Time Reference.

Visual C# Language
Describes this programming language, including concepts, wizards, samples, and a complete C# Language Specification.

.NET Framework Reference Information
Links to the major starting points of the .NET Framework documentation.

Upgrading Reference for Visual Basic and Visual C#
Describes the tool for upgrading from Visual Basic 6 to Visual Basic .NET and the tool for converting Visual J++ projects to Visual C#.

Legacy ActiveX Control Reference
Links to topics on the Masked Edit ActiveX control and the MSChart ActiveX control.

Related Sections

Visual Basic Code Example Topics
Lists quickstart-like code examples that help you perform common tasks in Visual Basic .NET.

Automation and Extensibility Reference
Contains automation model language reference topics for the integrated development environment (IDE), including debugging.

Development Environment Reference
Covers Visual Studio commands and Devenv command-line switches, and also includes Help topics on all user interface elements.

Language Equivalents
Compares data types, operators, controls, and programmable objects in several programming languages.

User Interface Reference
Contains topics for dialog boxes and windows found in Visual Studio. In the Help table of contents, all topics about user interface elements (windows, dialog boxes, and so forth) are found under the book named **Visual Studio**.

Visual Basic Language

Visual Basic .NET, the next generation of the Visual Basic language, is a fast and easy way to create .NET-based applications, including XML Web services and Web applications.

Visual Basic .NET has many new and improved features that make it a powerful object-oriented programming language, including inheritance, interfaces, and overloading. Other new language features include free threading and structured exception handling. Visual Basic .NET also fully integrates the .NET Framework and the Common Language Runtime, which provide language interoperability, garbage collection, enhanced security, and improved versioning support.

In This Section

Visual Basic Language Tour
Introduces the essential components of Visual Basic .NET, an object-oriented programming language.
Visual Basic Language and Run-Time Reference
Provides a complete reference for the Visual Basic language and its run-time elements.
Visual Basic Compiler Options
Offers information on using the Visual Basic compiler, and a reference to the Visual Basic compiler options.
Visual Basic .NET Language Specification
Provides a complete description of the Visual Basic .NET language, including syntax, semantics, and design.

Related Sections

Introducing Visual Studio .NET
Provides information about new features, a high-level overview of Visual Studio .NET and the .NET Framework, and tips for getting started.
Developing with Visual Studio .NET
Discusses the shared tools that allow you to design, develop, debug, and deploy applications.
Samples and Walkthroughs
Demonstrates how to create rich client applications and Web Forms applications, and how to create and access Web services. Also provides information on the sample applications Duwamish and Fitch and Mather.

Visual Basic Language Tour

As with any modern programming language, Visual Basic supports a number of common programming constructs and language elements. This language tour walks through all the essential elements of programming with Visual Basic.

In This Section

What's New in the Visual Basic Language
Provides information on inheritance, interfaces, overloading, multithreading, structured exception handling, custom attributes and common language specification compliance.

Visual Basic Language Walkthroughs
Provides step-by-step instructions for many programming tasks, including declaring and raising events, handling events, defining classes, using file I/O, calling Windows APIs, creating interfaces, building a common language runtime-based stack class, and multithreading.

Language Changes in Visual Basic
Provides an overview of the new features contained in Visual Basic .NET.

Program Structure and Code Conventions
Introduces the typical Visual Basic program structure, provides a simple Visual Basic program, "Hello World," and discusses Visual Basic code conventions.

Visual Basic Language Features
Provides an overview of the essential components of Visual Basic .NET.

Object-Oriented Programming in Visual Basic
Covers creating objects from the classes you define, and using those objects to simplify your coding and increase code reuse.

COM Interop
Explains the interoperability issues associated with creating and using COM objects with Visual Basic .NET.

Multithreading in Visual Basic .NET
Shows how to write code that performs multiple tasks simultaneously, to improve the performance and responsiveness of your applications.

Processing Drives, Folders, and Files
Covers file I/O operations using the **System.IO** model, **FileOpen** and **Write** functions, and the File System Object model.

Exception Handling
Covers using structured and unstructured exception-handling techniques to detect and possibly recover from program errors during execution.

Related Sections

Visual Basic Language and Run-Time Reference
Provides a complete reference for the Visual Basic language and its run-time elements.

Visual Basic Compiler Options
Offers information on using the Visual Basic compiler, and a reference to the Visual Basic compiler options.

Visual Basic .NET Language Specification
Provides a complete description of the Visual Basic .NET language, including syntax, semantics, and design.

What's New in the Visual Basic Language

Visual Basic .NET has many new and improved language features — such as inheritance, interfaces, and overloading — that make it a powerful object-oriented programming language. As a Visual Basic developer, you can now create multithreaded, scalable applications using explicit multithreading. Other new language features in Visual Basic .NET include structured exception handling, custom attributes, and common language specification (CLS) compliance.

The CLS is a set of rules that standardizes such things as data types and how objects are exposed and interoperate. Visual Basic .NET adds several features that take advantage of the CLS. Any CLS-compliant language can use the classes, objects, and components you create in Visual Basic .NET. And you, as a Visual Basic user, can access classes, components, and objects from other CLS-compliant programming languages without worrying about language-specific differences such as data types. CLS features used by Visual Basic .NET programs include assemblies, namespaces, and attributes.

Visual Basic .NET supports many new or improved object-oriented language features such as inheritance, overloading, the **Overrides** keyword, interfaces, shared members, and constructors.

Also included are structured exception handling, delegates, and several new data types.

What's New in Visual Basic .NET 2003

Visual Basic .NET 2003 has added functionality that simplifies bit manipulation and loop variable declaration.

Bit Shift Operators

Visual Basic .NET now supports arithmetic left and right shift operations on integral data types (**Byte**, **Short**, **Integer**, and **Long**). Arithmetic shifts are not circular, which means the bits shifted off one end of the result are not reintroduced at the other end. The corresponding assignment operators are provided as well. For details, see Bit Shift Operators and Assignment Operators.

Loop Variable Declaration

Visual Basic .NET now allows you to declare a loop variable as part of a **For** or **For Each** loop. You can include an **As** clause for the variable in the **For** or **For Each** statement, provided no variable of that name has been declared outside the loop. The scope of a loop variable declared in this manner is the loop itself. For details, see For...Next Statements and For Each...Next Statements.

What's New in Visual Basic .NET 2002

Visual Basic .NET 2003 also includes the following features, which were introduced in Visual Basic .NET 2002.

Inheritance

Visual Basic .NET supports *inheritance* by allowing you to define classes that serve as the basis for derived classes. Derived classes inherit and can extend the properties and methods of the base class. They can also override inherited methods with new implementations. All classes created with Visual Basic .NET are inheritable by default. Because the forms you design are really classes, you can use inheritance to define new forms based on existing ones. For details, see Inheritance.

Exception Handling

Visual Basic .NET supports *structured exception handling*, using an enhanced version of the **Try...Catch...Finally** syntax supported by other languages such as C++. Structured exception handling combines a modern control structure (similar to **Select Case** or **While**) with exceptions, protected blocks of code, and filters. Structured exception handling makes it easy to create and maintain programs with robust, comprehensive error handlers. For details, see Exception Handling.

Overloading

Overloading is the ability to define properties, methods, or procedures that have the same name but use different data types. Overloaded procedures allow you to provide as many implementations as necessary to handle different kinds of data, while giving the appearance of a single, versatile procedure. For details, see Overloaded Properties and Methods.

Overriding Properties and Methods

The **Overrides** keyword allows derived objects to override characteristics inherited from parent objects. Overridden members have the same arguments as the members inherited from the base class, but different implementations. A member's new implementation can call the original implementation in the parent class by preceding the member name with **MyBase**. For details, see Overriding Properties and Methods.

Constructors and Destructors

Constructors are procedures that control initialization of new instances of a class. Conversely, *destructors* are methods that free system resources when a class leaves scope or is set to **Nothing.** Visual Basic .NET supports constructors and destructors using the **Sub New** and **Sub Finalize** procedures. For details, see Object Lifetime: How Objects are Created and Destroyed.

Data Types

Visual Basic .NET introduces three new data types. The **Char** data type is an unsigned 16-bit quantity used to store Unicode characters. It is equivalent to the .NET Framework **System.Char** data type. The **Short** data type, a signed 16-bit integer, was named **Integer** in earlier versions of Visual Basic. The **Decimal** data type is a 96-bit signed integer scaled by a variable power of 10. In earlier versions of Visual Basic, it was available only within a **Variant**. For details, see Data Types.

Interfaces

Interfaces describe the properties and methods of classes, but unlike classes, do not provide implementations. The **Interface** statement allows you to declare interfaces, while the **Implements** statement lets you write code that puts the items described in the interface into practice. For details, see Interfaces in Visual Basic .NET.

Delegates

Delegates — objects that can call the methods of objects on your behalf — are sometimes described as type-safe, object-oriented function pointers. You can use delegates to let procedures specify an event handler method that runs when an event occurs. You can also use delegates with multithreaded applications. For details, see Delegates and the AddressOf Operator.

Shared Members

Shared members are properties, procedures, and fields that are shared by all instances of a class. Shared data members are useful when multiple objects need to use information that is common to all. Shared class methods can be used without first creating an object from a class. For details, see Shared Members.

References

References allow you to use objects defined in other assemblies. In Visual Basic .NET, references point to assemblies instead of type libraries. For details, see References and the Imports Statement.

Namespaces

Namespaces prevent naming conflicts by organizing classes, interfaces, and methods into hierarchies. For details, see Namespaces.

Assemblies

Assemblies replace and extend the capabilities of type libraries by, describing all the required files for a particular component or application. An assembly can contain one or more namespaces. For details, see Assemblies.

Attributes

Attributes enable you to provide additional information about program elements. For example, you can use an attribute to specify which methods in a class should be exposed when the class is used as a XML Web service. For details, see Attributes.

Multithreading

Visual Basic .NET allows you to write applications that can perform multiple tasks independently. A task that has the potential of holding up other tasks can execute on a separate thread, a process known as *multithreading*. By causing complicated tasks to run on threads that are separate from your user interface, multithreading makes your applications more responsive to user input. For details, see Multithreaded Applications.

See Also

What's New in Visual Basic and Visual C# | What's New in Visual Studio .NET | Language Changes in Visual Basic | Visual Basic Language and Run-time Reference | Visual Basic .NET Language Specification | Exception Handling | Overloaded Properties and Methods | Overriding Properties and Methods | Object Lifetime: How Objects are Created and Destroyed | Interfaces in Visual Basic .NET | Delegates and the AddressOf Operator | Shared Members | References and the Imports Statement | Namespaces | Assemblies | Attributes | Multithreaded Applications

Visual Basic Language Walkthroughs

Walkthroughs give step-by-step instructions for common scenarios, which makes them a good place to start learning about the product or a particular feature area.

In This Section

Declaring and Raising Events
Illustrates how events are declared and raised in Visual Basic .NET.

Handling Events
Shows how to handle events using either the standard **WithEvents** keyword or the new **AddHandler/RemoveHandler** keywords.

Defining Classes
Describes how to declare a class and its fields, properties, methods, and events.

Calling Windows APIs
Explains how to use **Declare** statements and call Windows APIs. Includes information about using attributes to control marshaling for the API call and how to expose an API call as a method of a class.

Creating and Implementing Interfaces
Shows how interfaces are declared and implemented in Visual Basic .NET.

Multithreading
Shows how to create a multithreaded application that searches a text file for occurrences of a word.

Structured Exception Handling
Demonstrates how to create and maintain programs with comprehensive error handling, and how to use **Try...Catch...Finally**.

Creating COM Objects with Visual Basic .NET
Demonstrates how to create COM objects in Visual Basic .NET, both with and without the COM class template.

Implementing Inheritance with COM Objects
Demonstrates how to use Visual Basic 6.0 to create a COM object containing a class, and then use it as a base class in Visual Basic .NET.

Creating Your Own Collection Class
Describes how to use inheritance to create a simple, strongly typed collection inherited from the CollectionBase class.

Manipulating Files and Folders in Visual Basic .NET
Demonstrates how to use Visual Basic functions to determine information about a file, search for a string in a file, and write to a file.

Manipulating Files Using .NET Framework Methods
Demonstrates how to use .NET Framework methods to determine information about a file, search for a string in a file, and write to a file.

Interacting with Files and Folders in Visual Basic .NET
Illustrates how to interact with files and folders by creating an application that examines text files in a folder and returns information about the files.

Related Sections

Visual Basic Walkthroughs
Provides links to Visual Basic walkthroughs covering distributed applications, data, Web services, Windows Forms, Web Forms, component and control authoring, Framework Services, upgrading, internationalization, and accessibility.

Language Changes in Visual Basic

While earlier versions of Visual Basic are targeted for Microsoft Windows client applications, Visual Basic .NET is intended for creating XML Web service applications as well. For this purpose, Visual Basic .NET generates managed code for the common language runtime. This entails changes to the language itself.

The changes in Visual Basic .NET are intended to:

- Simplify the language and make it more consistent.
- Add new features requested by users.
- Make code easy to read and maintain.
- Help programmers avoid introducing coding errors.
- Make applications more robust and easier to debug.

The following topics describe language changes from previous versions in Visual Basic .NET.

In This Section

Programming Element Support Changes Summary
Provides an alphabetical list of programming elements that have changed or are no longer supported in Visual Basic .NET, and includes their replacements.

Array Changes
Provides links to topics covering changes to array bounds, size declarations, and the **ReDim** statement.

Data Type Changes
Provides links to topics covering changes to data type declaration, usage, and conversion, as well as the **Integer** and **Universal** types.

Declaration Changes
Provides links to topics covering changes to declaration syntax, string length, structures, and the scope of variables inside blocks.

Function Changes
Provides links to topics covering changes to the **Format** function, functions related to dates and times, and functions that include the **String** ($) suffix.

Miscellaneous Changes
Provides links to topics covering changes to operators and file handling.

Object and Component Changes
Provides links to topics covering changes to object creation, binary compatibility, and interfaces and classes.

Procedure Changes
Provides links to topics covering changes to parameter passing, procedure calling sequences, and procedure declarations.

Property Changes
Provides links to topics covering changes to default properties, property procedures, and property arguments.

Control Flow Changes
Provides links to topics covering changes to execution flow, and new support for structured exception handling.

Related Sections

What's New in the Visual Basic Language
Includes information on inheritance, interfaces, overloading, multithreading, structured exception handling, custom attributes, and common language specification compliance.

Introduction to Visual Basic .NET for Visual Basic Veterans
Covers changes to Visual Basic that include, among others, Web functionality, projects, forms, constants, the **Circle**, **Line**, and **Pset** methods, and the integrated development environment (IDE).

Visual Basic Language and Run-Time Reference
Provides a complete reference for the Visual Basic language and its run-time elements.

Visual Basic .NET Language Specification
Provides a complete description of the Visual Basic .NET language, including syntax, semantics, and design.

Programming Element Support Changes Summary

Visual Basic .NET changes the manner in which it supports various programming elements, mostly for interoperability with the common language runtime. Many Visual Basic 6.0 elements are renamed, reclassified, or combined with other programming elements for Visual Basic .NET. Several elements are no longer supported, because the common language runtime includes functionality that makes them unnecessary.

For additional information on changes to Visual Basic — including the integrated development environment (IDE), Web functionality, projects, forms, constants, and the **Circle**, **Line**, and **Pset** methods — see Introduction to Visual Basic .NET for Visual Basic Veterans.

The following table lists changed programming elements and their replacements.

Visual Basic 6.0 programming element	Visual Basic .NET equivalent	Namespace, class, or run-time library location
Abs function	Abs Method	System Namespace, Math Class
AscB function	Asc Function	Visual Basic Run-Time Library Members, Strings module
As Any keyword phrase	Not supported in Visual Basic .NET. See Declaration Syntax Changes in Visual Basic	Does not apply
Atn function	Atan Method	System Namespace, Math Class
Calendar property	CurrentCulture Property	System.Globalization Namespace, CultureInfo Class
ChDir statement	ChDir Function	Visual Basic Run-Time Library Members, FileSystem module
ChDrive statement	ChDrive Function	Visual Basic Run-Time Library Members, FileSystem module
Chr$, **ChrB**, functions	Chr Function	Visual Basic Run-Time Library Members, Strings module
Close statement	FileClose Function	Visual Basic Run-Time Library Members, FileSystem module
Cos function	Cos Method	System Namespace, Math Class
Currency data type	Decimal Data Type; see Data Type Changes in Visual Basic	Visual Basic Run-Time Library Members, VariantType Enumeration
CVDate function	DateValue Function	Visual Basic Run-Time Library Members, DateAndTime module
CVError function	Error Statement	Does not apply
Date function, **Date** statement	Now Property, Today Property	Visual Basic Run-Time Library Members, DateAndTime module
Date$ function	DateString Property	Visual Basic Run-Time Library Members, DateAndTime module
Debug.Assert method	Assert, Fail methods	System.Diagnostics Namespace, Debug Class
Debug.Print method	Write, WriteIf, WriteLine, and WriteLineIf methods	System.Diagnostics Namespace, Debug Class
Deftype statements	Not supported in Visual Basic .NET. See Data Type Changes in Visual Basic	Does not apply
DeleteSetting statement	DeleteSetting Function	Visual Basic Run-Time Library Members, Interaction module
DoEvents function	DoEvents Method	System.Windows.Forms Namespace, Application Class
Empty keyword	Nothing	Does not apply
Eqv operator	= Operator; see Boolean Operator Changes in Visual Basic	Does not apply
Exp function	Exp Method	System Namespace, Math Class
FileCopy statement	FileCopy Function	Visual Basic Run-Time Library Members, FileSystem module

Get statement	FileGet Function	Visual Basic Run-Time Library Members, FileSystem module
GoSub statement	Not supported in Visual Basic .NET; use the Return Statement. See Control Statement Changes in Visual Basic	Does not apply
Initialize event	Not supported in Visual Basic .NET; use **Sub New**. See Class and Interface Changes in Visual Basic and Using Constructors and Destructors	Does not apply
Imp operator	Not supported in Visual Basic .NET. See Not and Or operators, and Boolean Operator Changes in Visual Basic	Does not apply
Input #, **Input$** statements, **Input$**, **InputB**, **InputB$** functions	Input Function	Visual Basic Run-Time Library Members, FileSystem module
Instancing property	Not supported in Visual Basic .NET. See Class and Interface Changes in Visual Basic	Does not apply
InStrB function	InStr Function	Visual Basic Run-Time Library Members, Strings module
IsEmpty function	IsNothing Function	Visual Basic Run-Time Library Members, Information module
IsMissing function	Not supported in Visual Basic .NET. See Procedure Declaration Changes in Visual Basic	Does not apply
IsNull function	IsDbNull Function	Visual Basic Run-Time Library Members, Information module
IsObject function	IsReference Function	Visual Basic Run-Time Library Members, Information module
Kill statement	Kill Function	Visual Basic Run-Time Library Members, FileSystem module
LCase$ function	LCase Function	Visual Basic Run-Time Library Members, Strings module
Left$, **LeftB**, **LeftB$** functions	Left Function	Visual Basic Run-Time Library Members, Strings module
LenB function	Len Function	Visual Basic Run-Time Library Members, Strings module
Let, **Set** assignment statements	Not supported in Visual Basic .NET; the new **Set** statement is unrelated to the older one. See Default Property Changes in Visual Basic	Does not apply
Line Input # statement	LineInput Function	Visual Basic Run-Time Library Members, FileSystem module
Lock statement	Lock, Unlock Functions	Visual Basic Run-Time Library Members, FileSystem module
Log function	Log Method	System Namespace, Math Class
LSet, **RSet** statements	LSet Function, PadRight, PadLeft; see Data Type Changes in Visual Basic	Visual Basic Run-Time Library Members, Strings module and System Namespace, String Class
LTrim$ function	LTrim Function	Visual Basic Run-Time Library Members, Strings module
MidB function	Mid Function	Visual Basic Run-Time Library Members, Strings module
MidB statement	Mid Statement	Visual Basic Run-Time Library Members, Strings module
MkDir statement	MkDir Function	Visual Basic Run-Time Library Members, FileSystem module
Name statement	Rename Function	Visual Basic Run-Time Library Members, FileSystem module
Now function	Now Property	Visual Basic Run-Time Library Members, DateAndTime module
Null keyword	Nothing	Does not apply

Oct$ function	Oct Function	Visual Basic Run-Time Library Members, Conversion module
On ... GoSub construction	Not supported in Visual Basic .NET; use Select Case Statement. See Control Statement Changes in Visual Basic	Does not apply
On ... GoTo construction	Not supported in Visual Basic .NET; use Select Case Statement. See Control Statement Changes in Visual Basic	Does not apply
Open statement	FileOpen Function	Visual Basic Run-Time Library Members, FileSystem module
Option Base statement	Not supported in Visual Basic .NET. See Array Bound Changes in Visual Basic	Does not apply
Option Private Module statement	Not supported in Visual Basic .NET; use Module Statement. See Class and Interface Changes in Visual Basic	Does not apply
Print # statement	Print, PrintLine Functions	Visual Basic Run-Time Library Members, FileSystem module
Property Get, **Property Let**, **Property Set** statements	Not supported in Visual Basic .NET. See Property Changes in Visual Basic	Does not apply
Put statement	FilePut Function	Visual Basic Run-Time Library Members, FileSystem module
Reset statement	Reset Function	Visual Basic Run-Time Library Members, FileSystem module
Right$, **RightB** functions	Right Function	Visual Basic Run-Time Library Members, Strings module
RmDir statement	RmDir Function	Visual Basic Run-Time Library Members, FileSystem module
Round function	Round Method	System Namespace, Math Class
RSet, **LSet** statements	RSet Function; see Data Type Changes in Visual Basic	Visual Basic Run-Time Library Members, Strings module
RTrim$ function	RTrim Function	Visual Basic Run-Time Library Members, Strings module
SaveSetting statement	SaveSetting Function	Visual Basic Run-Time Library Members, Interaction module
Scale method	Not supported in Visual Basic .NET	Does not apply
Set, **Let** assignment statements	Not supported in Visual Basic .NET; the new **Set** statement is unrelated to the older one. See Default Property Changes in Visual Basic	Does not apply
SetAttr statement	SetAttr Function	Visual Basic Run-Time Library Members, FileSystem module
Sgn function	Sign Function	System Namespace, Math Class
Sin function	Sin Method	System Namespace, Math Class
Sqr function	Sqrt Function	System Namespace, Math Class
String function	String Constructor; see String ($) Function Changes in Visual Basic	System Namespace, String Class
String ($) functions	Not supported in Visual Basic .NET. See String ($) Function Changes in Visual Basic	Does not apply
Terminate event	Not supported in Visual Basic .NET; use **Sub Dispose** and **Sub Finalize**. See Class and Interface Changes in Visual Basic and Using Constructors and Destructors	Does not apply
Time function, **Time** statement	TimeOfDay Property; see DateTime Structure, Date Data Type	Visual Basic Run-Time Library Members, DateAndTime module
Time$ function	TimeString Property	Visual Basic Run-Time Library Members, DateAndTime module

Timer function	Timer Property	Visual Basic Run-Time Library Members, DateAndTime module
Trim$ function	LTrim, RTrim, and Trim Functions	Visual Basic Run-Time Library Members, Strings module
Type statement	Not supported in Visual Basic .NET; use Structure Statement. See Structure Declaration Changes in Visual Basic	Does not apply
UCase$ function	UCase Function	Visual Basic Run-Time Library Members, Strings module
Unlock statement	Lock, Unlock Functions	Visual Basic Run-Time Library Members, FileSystem module
Variant data type	Object Data Type; see Universal Data Type Changes in Visual Basic	Does not apply
Wend keyword	While...End While Statements and End Statement; see Control Statement Changes in Visual Basic	Does not apply
Width # statement	FileWidth Function	Visual Basic Run-Time Library Members, FileSystem module
Write # statement	Write, WriteLine Functions	Visual Basic Run-Time Library Members, FileSystem module

See Also

Language Changes in Visual Basic | Introduction to Visual Basic .NET for Visual Basic Veterans

Array Changes

The following topics describe changes to arrays between Visual Basic 6.0 and Visual Basic .NET.

In This Section

Array Bound Changes in Visual Basic
Discusses changes to array bounds and default bounds.
Array Size Declaration Changes in Visual Basic
Describes changes to the specification of array sizes.
ReDim Statement Changes in Visual Basic
Discusses changes to declarations that include the **ReDim** statement.

Related Sections

Programming Element Support Changes Summary
Provides an alphabetical list of programming elements that have changed or are no longer supported in Visual Basic .NET, and includes their replacements.
Language Changes in Visual Basic
Provides an overview of new and changed features in Visual Basic .NET.

Array Bound Changes in Visual Basic

Visual Basic .NET updates the declaration of array bounds to provide interoperability with arrays in other programming languages.

Visual Basic 6.0

In Visual Basic 6.0, the default lower bound of every dimension of an array is 0. You can change this to 1 with the **Option Base** statement. You can also override the default lower bound in individual array declarations.

If you leave the default at 0, the number of elements in the array is equal to the upper bound plus one. The following declaration reserves 21 elements for the array `Weight`:

```
Dim Weight(20) As Single
```

Visual Basic .NET

In Visual Basic .NET, the lower bound of every array dimension is 0, and you cannot declare it to be otherwise. The **Option Base** statement is not supported.

The number you specify for each dimension in the declaration is the upper bound, and the initial element count is equal to the upper bound plus one. The declaration in the preceding example reserves 21 elements for `Weight`, with subscripts 0 through 20.

You can also specify a *zero-length array*, which does not contain any elements, by declaring one of its upper bounds to be –1.

See Also

Declaration Syntax Changes in Visual Basic | Array Size Declaration Changes in Visual Basic | Arrays | Dim Statement | UBound Function | Programming Element Support Changes Summary

Array Size Declaration Changes in Visual Basic

Visual Basic .NET updates array size declaration for interoperability with the common language runtime.

Visual Basic 6.0

In Visual Basic 6.0, you can specify the size of an array in its declaration, as in the following example:

```
Dim Month(0 To 11) As Integer
```

This causes the array to have a fixed size, which you cannot change with the **ReDim** statement.

Visual Basic .NET

In Visual Basic .NET, an array cannot have a fixed size. The preceding example can be rewritten as either of the following declarations:

```
Dim Month(11) As Integer   ' Reserves 12 elements -- (0) through (11).
Dim Month() As Integer = New Integer(11) {}
```

These declarations are equivalent. Each specifies an initial size, which you can change during execution with the **ReDim** statement. To initialize the elements, you can use the following syntax:

```
Dim Month() As Integer = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}
```

If you specify –1 for one of the dimensions, the array does not contain any elements. A **ReDim** statement can change an array back and forth between empty and nonempty.

Number of Dimensions

Although an array's size can change in Visual Basic .NET, the number of dimensions must be fixed. The following example declares a three-dimensional array:

```
Dim Point( , , ) As Double
```

The **ReDim** statement can set or change the size of each dimension, but the array always remains three-dimensional.

See Also

Arrays | Dim Statement | New | ReDim Statement | Programming Element Support Changes Summary

ReDim Statement Changes in Visual Basic

Visual Basic .NET updates declaration with the **ReDim** statement for simplification and improved readability.

Visual Basic 6.0

In Visual Basic 6.0, you can use the **ReDim** statement to serve as the initial declaration of a dynamic array. The array does not have to be declared anywhere else.

Visual Basic 6.0 also allows you to change the rank of an array with **ReDim**, as in the following example:

```
Dim X(10) As Single
ReDim X(10, 10)
```

Visual Basic .NET

In Visual Basic .NET, you cannot use **ReDim** as a declaration. You must declare an array just like any other variable, using **Dim** or an equivalent statement, before it can appear in a **ReDim** statement.

You cannot change the rank of an array in Visual Basic .NET. The code in the preceding example causes an error.

See Also

ReDim Statement | Dim Statement | Programming Element Support Changes Summary

Data Type Changes

The following topics describe changes to data types between Visual Basic 6.0 and Visual Basic .NET.

In This Section

Data Type Changes in Visual Basic
Describes data type declaration, usage, and conversion.
Integer Data Type Changes in Visual Basic
Covers integer data type updates.
Universal Data Type Changes in Visual Basic
Covers universal data type updates.

Related Sections

Programming Element Support Changes Summary
Provides an alphabetical list of programming elements that have changed or are no longer supported in Visual Basic .NET, and includes their replacements.
Language Changes in Visual Basic
Provides an overview of new and changed features in Visual Basic .NET.

Data Type Changes in Visual Basic

Visual Basic .NET updates data types for interoperability with other programming languages and the common language runtime. The changes affect data type declaration, usage, and conversion.

Visual Basic 6.0

In Visual Basic 6.0, you use the **Deftype** statements — **DefBool**, **DefByte**, **DefCur**, **DefDate**, **DefDbl**, **DefDec**, **DefInt**, **DefLng**, **DefObj**, **DefSng**, **DefStr**, and **DefVar** — to set a variable's default type.

You use the **Currency** data type for calculations involving money and for fixed-point calculations.

A **Date** is stored in a **Double** format using eight bytes.

If two **Variant** variables containing integers are multiplied, an overflow condition causes the data type of the result to be changed to **Double**.

You can use the **LSet** and **RSet** statements to copy a variable of one user-defined type to another variable of a different user-defined type.

Visual Basic .NET

The **Deftype** statements are not supported in Visual Basic .NET. Nor is the **Currency** data type. Instead, use the new **Decimal** data type, which can handle more digits on both sides of the decimal point, for all money variables and calculations. **Decimal** is also directly supported by the common language runtime.

In Visual Basic .NET, the **Date** data type uses the common language runtime **DateTime** data type, which is an eight-byte integer value. Because of these different representations, there is no implicit conversion between the **Date** and **Double** data types. To convert between **Double** and the Visual Basic 6.0 representation of **Date**, use the **ToOADate** and **FromOADate** methods of the **DateTime** structure in the **System** namespace.

If multiplication of two **Object** variables containing integers results in an overflow, the result is changed to the 64-bit **Long** data type.

You cannot use **LSet** and **RSet** to assign one data type to another. Doing so entails a type-unsafe operation, particularly with structures, which could result in unverifiable code.

See Also

Decimal Data Type | Date Data Type | DateTime Structure | TimeSpan Structure | Val Function | Type Conversion Functions | Data Type Summary | Double Data Type | Long Data Type | Programming Element Support Changes Summary

Integer Data Type Changes in Visual Basic

Visual Basic .NET updates the integer data types for interoperability with other programming languages and with the common language runtime.

The following table shows correspondences between integer types in Visual Basic 6.0 and Visual Basic .NET.

Integer Size	Visual Basic 6.0 type and identifier type character	Visual Basic .NET type and identifier type character	Common language runtime type
8 bits, signed	(none)	(none)	**System.SByte**
16 bits, signed	**Integer** (**%**)	**Short** (none)	**System.Int16**
32 bits, signed	**Long** (**&**)	**Integer** (**%**)	**System.Int32**
64 bits, signed	(none)	**Long** (**&**)	**System.Int64**

On 32-bit systems, 32-bit integer operations are faster than either 16-bit or 64-bit integer operations. This means that in Visual Basic .NET, **Integer** is the most efficient and fundamental numeric type. You can improve performance in your applications by changing your **Long** declarations to **Integer** when you migrate to Visual Basic .NET.

> **Note** If you are interfacing with components created on platforms other than Visual Basic .NET, you must take care that your data types correspond to those of the other components. For example, if you use a **Declare** statement to refer to an external procedure created in Visual Basic 6.0, and the procedure defines an **Integer** argument (two bytes in Visual Basic 6.0), you must identify that argument as **Short** in the **Declare** statement, because that is the two-byte integer type in Visual Basic .NET.

See Also

Data Type Summary | Integer Data Type | Short Data Type | Long Data Type | Type Characters | Programming Element Support Changes Summary

Universal Data Type Changes in Visual Basic

Visual Basic .NET updates the universal data type for interoperability with the common language runtime.

Visual Basic 6.0

In Visual Basic 6.0, **Variant** serves as the universal data type. This means that you can store data of any type in a variable of type **Variant**.

Visual Basic .NET

In Visual Basic .NET, **Object** is the universal data type. A variable of type **Object** can hold data of any type. The **Variant** type is not supported, and all its functionality is supplied by **Object**.

Variant is still a reserved word in Visual Basic .NET, even though it has no syntactical use. This helps avoid confusion with its former meanings.

The **VarType** function returns a member of the **VariantType** enumeration that provides the data type classification of an object variable. You can also use classes in the **System** namespace to obtain numeric data type information for an **Object** instance, as shown in the following code:

```
Dim SomeObj As Object
' ... ... ... ...  SomeObj is assigned some value during processing.
' ... ... ... ...  Now we want to find out the data type of SomeObj.
Dim Dtype As Integer      ' To hold numeric data type result.
Dtype = Type.GetTypeCode(SomeObj.GetType())
```

See Also

Data Types | Value Types and Reference Types | Object Data Type | System Namespace | Object Class | Type Class | VarType Function | VariantType Enumeration | Programming Element Support Changes Summary

Declaration Changes

The following topics describe changes to declarations between Visual Basic 6.0 and Visual Basic .NET.

In This Section

Declaration Syntax Changes in Visual Basic
Describes multiple variable declaration and return type declaration.
String Length Declaration Changes in Visual Basic
Discusses the specification of string length.
Structure Declaration Changes in Visual Basic
Discusses structures, also known as user-defined types (UDTs).
Variable Scope Changes in Visual Basic
Discusses the scope of variables inside a block.

Related Sections

Programming Element Support Changes Summary
Provides an alphabetical list of programming elements that have changed or are no longer supported in Visual Basic .NET, and includes their replacements.
Language Changes in Visual Basic
Provides an overview of new and changed features in Visual Basic .NET.

Declaration Syntax Changes in Visual Basic

Visual Basic .NET introduces several changes related to the declaration of programming elements.

Multiple Variable Declaration

Visual Basic .NET revises simultaneous declaration of several variables for simplification.

Visual Basic 6.0

In Visual Basic 6.0, you can declare variables of different types in the same statement, but you must specify the data type of each variable or it defaults to **Variant**. The following example shows multiple declarations and their resulting data types:

```
Dim I, J As Integer             ' I is Variant, J is Integer.
Dim L As Integer, M As Integer  ' L is Integer, M is Integer.
Dim N As Integer, X As Double   ' N is Integer, X is Double.
```

Visual Basic .NET

In Visual Basic .NET, you can declare multiple variables of the same data type without having to repeat the type keyword. The declarations equivalent to those in the preceding example are as follows:

```
Dim I                           ' I is Object.
Dim J As Integer                ' J is Integer.
' -- OR --
Dim I As Object, J As Integer   ' I is Object, J is Integer.
Dim L, M As Integer             ' L is Integer, M is Integer.
Dim N As Integer, X As Double   ' N is Integer, X is Double.
```

External Procedure Declaration

Visual Basic 6.0

In Visual Basic 6.0, when you declare a reference to an external procedure with the **Declare** statement, you can specify **As Any** for the data type of any of the arguments and the return type. The **As Any** keywords disable type checking and allow any data type to be passed in or returned.

Visual Basic .NET

Visual Basic .NET does not support the **Any** keyword. In a **Declare** statement, you must specifically declare the data type of every argument and the return if **Option Strict** is **On**. This improves type safety. You can overload your procedure declaration to accommodate various argument data types. You cannot overload only on return types, but you can use the argument type overloads to vary the return type, or you can turn **Option Strict Off**.

Line Label Declaration

Visual Basic 6.0

In Visual Basic 6.0, a line number can directly precede a statement on the same line, without any separating character.

Visual Basic .NET

Visual Basic .NET requires that every line label be followed by a colon (**:**). A statement can optionally follow the colon on the same line, or the line label and colon can stand alone on the line.

See Also

Universal Data Type Changes in Visual Basic | Procedure Calling Sequence Changes in Visual Basic | Declare Statement | Rules of Procedure Overloading | Option Strict Statement | Data Types | Programming Element Support Changes Summary

String Length Declaration Changes in Visual Basic

Visual Basic .NET updates string length declaration for interoperability with the common language runtime.

Visual Basic 6.0

In Visual Basic 6.0, you can specify the length of a string in its declaration. This causes the string to have a fixed length, as in the following example:

```
Dim Name As String * 30
```

Visual Basic .NET

In Visual Basic .NET, you cannot declare a string to have a fixed length unless you use the VBFixedStringAttribute Class attribute in the declaration. The code in the preceding example causes an error.

You declare a string without a length. When your code assigns a value to the string, the length of the value determines the length of the string, as in the following example:

```
Dim Name As String
' ...
Name = "Name is now 30 characters long" ' Length can be changed later.
```

See Also

String Data Type | Dim Statement | Attributes Used in Visual Basic .NET | Programming Element Support Changes Summary

Structure Declaration Changes in Visual Basic

Visual Basic considers structures and user-defined types (UDTs) to be the same type of programming element. Visual Basic .NET updates structure declaration for unification and improved readability.

Visual Basic 6.0

In Visual Basic 6.0, you declare a structure using the **Type ... End Type** construction. The structure and its members all default to public access. Explicit access declaration is optional. The following example shows a valid structure declaration:

```
Type Employee
   EmpNumber As Integer     ' Defaults to Public access.
   EmpOfficePhone As String
   EmpHomePhone As String   ' Cannot be declared Private inside Type.
End Type
```

Visual Basic .NET

In Visual Basic .NET, the **Type** statement is not supported. You must declare structures using the **Structure** statement as part of a **Structure ... End Structure** construction. Every member of a structure must have an access modifier, which can be **Public**, **Friend**, or **Private**. You can also use the **Dim** statement, which defaults to public access. The structure in the preceding example can be declared as follows:

```
Structure Employee
   Public EmpNumber As Integer     ' Must declare access, even if Public.
   Dim EmpOfficePhone As String    ' Still defaults to Public access.
   Private EmpHomePhone As String  ' Can be made Private inside Structure.
End Structure
```

Visual Basic .NET unifies the syntax for structures and classes. Structures support most of the features of classes, including methods.

See Also

Structure Statement | Dim Statement | Public | Private | Programming Element Support Changes Summary

Variable Scope Changes in Visual Basic

Visual Basic .NET updates local variable scope to support block scope and improve structured programming.

Visual Basic 6.0

In Visual Basic 6.0, any variable declared inside a procedure has procedure scope, so it can be accessed anywhere else within the same procedure. If the variable is declared inside a block — that is, a set of statements terminated by an **End**, **Loop**, or **Next** statement — the variable is still accessible outside the block. The following example illustrates procedure scope:

```
For I = 1 To 10
   Dim N As Long
' N has procedure scope although it was declared within a block.
   N = N + Incr(I)
Next
W = Base ^ N
' N is still visible outside the block it is declared in.
```

Visual Basic .NET

In Visual Basic .NET, a variable declared inside a block has block scope, and it is not accessible outside the block. The preceding example can be rewritten as follows:

```
Dim N As Long = 0
' N is declared outside the block and has procedure scope.
For I As Integer = 1 To 10
   N = N + Incr(I)
Next I
W = Base ^ N
' N is visible outside the block but I is not.
```

Because the **For** statement declares `I` as part of the **For** block, `I` has block scope only.

However, a variable's lifetime is still that of the entire procedure. This is true whether the variable has block scope or procedure scope. If you declare a variable inside a block, and if you enter that block several times during the lifetime of the procedure, you should initialize the variable to avoid unexpected values.

See Also

Choice of Variable Scope | Programming Element Support Changes Summary

Function Changes

The following topics describe changes to functions between Visual Basic 6.0 and Visual Basic .NET.

In This Section

Format Function Changes in Visual Basic
Discusses date/time, numeric, and string format changes to the **Format** function.
Date and Time Changes in Visual Basic
Discusses changes to the **Date**, **Date$**, **Time**, and **Time$** functions.
String ($) Function Changes in Visual Basic
Discusses functions that include the **String** ($) suffix.

Related Sections

Programming Element Support Changes Summary
Provides an alphabetical list of programming elements that have changed or are no longer supported in Visual Basic .NET, and includes their replacements.
Language Changes in Visual Basic
Provides an overview of new and changed features in Visual Basic .NET.

Format Function Changes in Visual Basic

The Visual Basic .NET **Format** function now follows the common language runtime specification for formatting data. For details on data formatting in the .NET Framework, see Formatting Types.

The following sections detail the changes Visual Basic .NET makes to the date/time, numeric, and string formats.

Date/Time Format

Visual Basic 6.0

In Visual Basic 6.0, to display a short or long date, you use either the "ddddd" or "dddddd" format specifier. The **DayOfWeek** ("w") and **WeekOfYear** ("ww") specifiers display the day considered to be the first day of the week, and the week considered to be the first week of the year. The lowercase "m" character displays the month as a number without a leading zero. The **Quarter** specifier ("q") displays the quarter of the year as a number from 1 – 4.

To display a minute as a number with or without leading zeros, you use either the "Nn" or the "N" format specifier. The characters "Hh" display the hour as a number with leading zeros, and "ttttt" displays the time as a complete time. To display either an uppercase or lowercase "A" or "P" with any hour before or after noon, you use either "AM/PM", "am/pm", "A/P", "a/p", or "AMPM."

Visual Basic .NET

In Visual Basic .NET, "ddddd" and "dddddd" behave the same as "dddd", displaying the full name of the day. They do not display the short date and long date. **DayOfWeek** ("w") and **WeekOfYear** ("ww") are not supported; instead, use the **DatePart** function as follows:

```
Format(Datepart(DateInterval.Weekday, Now))
Format(Datepart(DateInterval.WeekOfYear, Now))
```

"M" and "m" apply to different things, so both are now case sensitive. Use uppercase "M" only for the month in the date portion of a date/time format, and lowercase "m" only for the minutes in the time portion.

The **Quarter** format specifier is not supported; instead, use the **DatePart** function as follows:

```
Format(DatePart(DateInterval.Quarter, Now))
```

To display the minute as a number with or without leading zeros, use "m" or "mm". The "ttttt" format is no longer supported. "H" and "h" apply to different things, so both are now case sensitive. Use uppercase "H" only for a 24-hour clock, and lowercase "h" only for a 12-hour clock. The AM/PM formats are replaced with "t" and "tt."

Numeric Format

Visual Basic 6.0

In Visual Basic 6.0, the **Format** function converts strings to numbers for you.

Negative numbers with an empty negative format string display an empty string.

Trailing decimal points are displayed.

Visual Basic 6.0 supports four format sections – positive, negative, zero, and null.

The exponent for formatting scientific notation supports the "#" placeholder.

The **Short Date/Time** specifier ("c"), displays a date and time in the ddddd ttttt format.

Visual Basic .NET

In Visual Basic .NET, the **Format** function does not convert strings to numbers for you. Thus, the first of the following two lines of code is invalid in Visual Basic .NET, while the second is valid:

```
Format("1.234", "#.#")   ' Displays "#.#".
Format(CSng("1.234"),"#.#")   ' Displays "1.234".
```

Negative numbers with an empty negative format string display a minus sign, as in the following example:

```
Format(-1, ";")   ' Displays "-".
```

Trailing decimal points are not displayed, as in the following examples:

```
Format(123, "###.")   ' Displays "123"
Format(123, "###.#")   ' Displays "123"
```

Visual Basic .NET supports three format sections – positive, negative, and zero. If a non-zero value rounds to zero according to the first or second format section, the resulting zero is formatted according to the third section.

The exponent for formatting scientific notation does not support the "#" placeholder; use the "0" (zero) placeholder instead. Thus, the first of the following two lines of code is invalid in Visual Basic .NET, while the second is valid:

```
Format(123, "#e+#")   ' Displays "12e+3".
Format(123, "#e+0")   ' Displays "1e+2".
```

The short date/time specifier ("c") is now reserved for **Currency** formatting; for date formatting, use the **General Date/Time** ("g") format.

String Format

Visual Basic 6.0

In Visual Basic 6.0, you create expressions for user-defined format strings with the @, &, <, >, and ! specifiers.

Visual Basic .NET

Visual Basic .NET eliminates support for user-defined format strings, so the @, &, <, >, and ! format specifiers have no meaning and are no longer supported.

See Also

Date and Time Changes in Visual Basic | Format Function | DatePart Function | Programming Element Support Changes Summary

Date and Time Changes in Visual Basic

Visual Basic .NET updates the **Date**, **Date$**, **Time**, and **Time$** functions, and replaces the **Now** and **Timer** functions with properties.

Visual Basic 6.0

In Visual Basic 6.0, the **Date** and **Time** functions return the system date and time in the four-byte **Date** format. Similarly, the **Date** and **Time** statements use the Visual Basic 6.0 format to set the system date and time.

The **Date$** and **Time$** functions return the system date and time in **String** format.

The **Now** and **Timer** functions specify the current date and time and the number of seconds elapsed since midnight.

Visual Basic .NET

Visual Basic .NET replaces **Date** and **Time** with the **Today** and **TimeOfDay** properties, which use the eight-byte common language runtime (CLR) **DateTime** structure. This corresponds with the **Date** data type in Visual Basic .NET. You can use **Today** and **TimeOfDay** to both set and fetch the system date and time.

Visual Basic .NET replaces the **Date$** and **Time$** functions with the **DateString** and **TimeString** properties. You can use **DateString** and **TimeString** to both set and fetch the system date and time.

The **Now** and **Timer** functions are replaced by read-only properties of the same names. Their calling sequences remain unchanged.

See Also

Format Function Changes in Visual Basic | Format Function | Today Property | TimeOfDay Property | DateTime Structure | Date Data Type | DateString Property | TimeString Property | Now Property | Timer Property | Programming Element Support Changes Summary

String ($) Function Changes in Visual Basic

Visual Basic .NET eliminates the need for some functions to have both a **String** version and a **Variant** version.

Visual Basic 6.0

In Visual Basic 6.0, some functions have two versions, one that returns a **String** value and one that returns a **Variant** value. These function pairs are differentiated by a dollar sign ($) suffix for the **String** version; for example, **Chr** and **Chr$**.

Visual Basic .NET

Visual Basic .NET replaces each function pair with a single function. The **Variant** versions are discontinued, and you can call the **String** versions either with or without the $ suffix.

Although only **Trim** is defined, Visual Basic .NET accepts **Trim$** because the $ works as an identifier type character for the **String** data type.

See Also

Format Function Changes in Visual Basic | Date and Time Changes in Visual Basic |
Programming Element Support Changes Summary | LTrim, RTrim, and Trim Functions

Miscellaneous Changes

The following topics describe changes to operators and file handling between Visual Basic 6.0 and Visual Basic .NET.

In This Section

Boolean Operator Changes in Visual Basic
Describes Boolean operators and logical expressions.

File Handling Changes in Visual Basic
Discusses expansion of file handling capabilities, and compatibility with previous versions of Visual Basic file I/O functions.

Related Sections

Programming Element Support Changes Summary
Provides an alphabetical list of programming elements that have changed or are no longer supported in Visual Basic .NET, and includes their replacements.

Language Changes in Visual Basic
Provides an overview of new and changed features in Visual Basic .NET.

Boolean Operator Changes in Visual Basic

Visual Basic .NET removes two Boolean operators and adds two others to improve performance.

Visual Basic 6.0

In Visual Basic 6.0, Boolean operators — **And**, **Or**, **Not**, and **Xor** — always evaluate all the expressions in their operands.

You use the **Eqv** and **Imp** operators to perform logical equivalences and implications on two expressions.

Visual Basic .NET

In Visual Basic .NET, the **And**, **Or**, **Not**, and **Xor** operators still evaluate all expressions contributing to their operands. Visual Basic .NET also introduces two new operators, **AndAlso** and **OrElse**, that can reduce execution time by *short-circuiting* logical evaluations. If the first operand of an **AndAlso** operator evaluates to **False**, the second operand is not evaluated. Similarly, if the first operand of an **OrElse** operator evaluates to **True**, the second operand is not evaluated.

> **Note** You should be careful using short-circuiting operators. If the second operand contains a procedure call, you cannot rely on that procedure being called every time the operator executes.

The **Eqv** and **Imp** operators are not supported. Use the equals (**=**) comparison operator in place of **Eqv** for both logical and bitwise evaluations. You can replace the logical **Imp** operator with an expression using the **Not** and **Or** operators, as shown in the following example:

```
Result = A Imp B         ' True unless A is True and B is False.
Result = (Not A) Or B    ' Same as A Imp B.
```

You can replace a bitwise **Imp** in the same manner, using **Not** and **Or** on numeric operands.

See Also

And Operator | AndAlso Operator | Or Operator | OrElse Operator | Not Operator | Xor Operator | TimeSpan Structure | Programming Element Support Changes Summary

File Handling Changes in Visual Basic

Visual Basic .NET expands file-handling capabilities while providing compatibility with previous versions of Visual Basic file I/O functions.

Visual Basic 6.0

In Visual Basic 6.0, file handling is accomplished using various file I/O functions such as **Open**, **Input**, **Output**, and **Append**. The File System Object provides an object-oriented method of working with files.

Visual Basic .NET

In Visual Basic .NET, file handling is accomplished through the **System.IO** namespace, which duplicates and expands upon the File System Object. In addition, the **System.IO.File** namespace includes functions that provide compatibility with the older Visual Basic file I/O functions.

The **FileStream** class provides access to standard input and output files and error devices.

The following table lists file-related enumerations available in Visual Basic .NET.

Enumeration	Purpose
FileAccess Enumeration	Defines constants for read, write, or read/write access to a file
FileMode Enumeration	Specifies how the operating system opens a file
FileShare Enumeration	Provides constants for controlling the level of access other FileStream members have to the same file
FileAttributes Enumeration	Provides access to stored attributes, such as whether a file is a directory, is encrypted, hidden, or read-only, or is a system file or a temporary file

See Also

Processing Drives, Folders, and Files | File Access Through StreamReader and StreamWriter Classes | File Access Types, Functions, and Statements | System.IO Namespace | File Class | FileStream Class | FileAccess Enumeration | FileMode Enumeration | FileShare Enumeration | FileAttributes Enumeration | Programming Element Support Changes Summary

Object and Component Changes

The following topics describe changes to objects and components between Visual Basic 6.0 and Visual Basic .NET.

In This Section

Object Creation Changes in Visual Basic
Covers implicit object creation.
Binary Compatibility Changes in Visual Basic
Discusses updates to the mechanism for preserving compatibility with older versions of a component.
Class and Interface Changes in Visual Basic
Discusses declaration and usage of classes and interfaces.

Related Sections

Programming Element Support Changes Summary
Provides an alphabetical list of programming elements that have changed or are no longer supported in Visual Basic .NET, and includes their replacements.
Language Changes in Visual Basic
Provides an overview of new and changed features in Visual Basic .NET.

Object Creation Changes in Visual Basic

Visual Basic .NET revises object creation for improved readability and to support constructors with arguments.

Visual Basic 6.0

In Visual Basic 6.0, an object variable declared with the keywords **As New** is initialized to **Nothing**, meaning that no object has yet been created. Every time the variable is encountered during execution, it is evaluated before the code executes. If the variable contains **Nothing**, an object of the appropriate class is created prior to execution of the code that uses it.

Visual Basic .NET

In Visual Basic .NET, there is no implicit object creation. If an object variable contains **Nothing** when it is encountered, it is left unchanged and no instance is automatically created.

You can create an object with the same statement that declares the object variable. Each of the following two lines of code creates an object from a class, `EmpObj`, that is already defined in the application. The two lines are treated as equivalent, with the first being taken as shorthand for the second:

```
Dim Emp As New EmpObj           ' Shorthand for next line.
Dim Emp As EmpObj = New EmpObj  ' EmpObj created; Emp points to EmpObj.
```

In both cases, an instance of the `EmpObj` class is created as soon as the **Dim** statement is executed, and `Emp` is initialized to a reference to the new object.

Parameterized Constructors

Some classes have constructors that take arguments. If you are creating an object from such a class, you can include its arguments in the declaration. The following example shows two equivalent declarations that pass arguments to the object's constructor, with the first being taken as shorthand for the second:

```
Dim Q As New Quark(12, 0.0035)          ' Shorthand for next line.
Dim Q As Quark = New Quark(12, 0.0035)  ' Parameterized constructor.
```

See Also

Object Lifetime: How Objects are Created and Destroyed | Dim Statement | New | Nothing | Programming Element Support Changes Summary

Binary Compatibility Changes in Visual Basic

Visual Basic .NET updates the mechanism for preserving compatibility with older versions of a component.

Visual Basic 6.0

In Visual Basic 6.0, the Binary Compatibility option allows you to automatically retain class and interface identifiers from a previous version of a component when you compile a new version. The new version contains both the old identifiers and the new identifiers, and client applications can continue using the old interface to your component without having to be updated.

Visual Basic .NET

In Visual Basic .NET, binary compatibility is accomplished using attributes. This gives you direct control over the information placed in your compiled component, such as class and interface identifiers, virtual table offsets, and any appropriate COM attributes. Because you can explicitly select attributes for preservation, the Binary Compatibility option is not supported.

See Also

Attributes Overview | Attributes Used in Visual Basic .NET | Global Attributes | ComClassAttribute Class | Programming Element Support Changes Summary

Class and Interface Changes in Visual Basic

Visual Basic .NET introduces a number of changes that affect the declaration and usage of classes and interfaces.

Implements Statement

Visual Basic 6.0

In Visual Basic 6.0, a class definition implicitly includes a default interface, which defines the public members of the class. This interface can be referenced by another class. In particular, a class can specify another class name in an **Implements** statement.

Visual Basic .NET

In Visual Basic .NET, classes can have methods, and you can define interfaces explicitly. The properties and methods of a class belong exclusively to the class itself. The **Implements** statement can specify only an interface, not another class.

Global Classes

Visual Basic 6.0

When you create a new class in Visual Basic 6.0, the allowable values of the **Instancing** property include **GlobalSingleUse** and **GlobalMultiUse**. These indicate that other components can invoke the properties and methods of the new class as if they were global functions. An instance of the class is implicitly created the first time one of its members is called. This functionality enables a server to expose properties and methods that can be accessed without qualification.

Visual Basic .NET

Visual Basic .NET provides the same functionality by allowing you to expose standard module members from a server. You can add properties and methods of a shared class to the **System** namespace using the **Imports** statement. Visual Basic .NET does not support the **Instancing** property. You can achieve the effect of **GlobalMultiUse** with **Public** class access and the appropriate constructor access.

Option Private Module Statement

Visual Basic 6.0

In Visual Basic 6.0, the **Option Private Module** statement indicates that the entire module is private, and its components are not accessible outside its project.

Visual Basic .NET

Visual Basic .NET does not support the **Option Private Module** statement. This specification is handled by access modifiers in the **Module** statement.

Default Members

Visual Basic 6.0

In Visual Basic 6.0, you can specify that any particular method or data member is the default member of a class.

Visual Basic .NET

In Visual Basic .NET, a default member of a class or structure can only be a property that takes one or more arguments. This enables interoperability with other programming languages.

Initialization and Termination

Visual Basic 6.0

In Visual Basic 6.0, class modules use the **Initialize** and **Terminate** events to perform any necessary actions at the time of creation and destruction. The **Initialize** event is raised when the object is first encountered following a **New** statement, and

Terminate is raised as soon as the last reference to the object is released. However, you can also call these event methods directly at any time during execution.

Visual Basic .NET

Visual Basic .NET uses the constructor/destructor model, and does not support **Class_Initialize** and **Class_Terminate**. A class uses the **Sub** statement to declare its constructor. This enables support of inheritance, which requires parameterized constructors. It also protects the constructor and destructor from being called except at creation and termination.

Indeterminate Lifetime

Object lifetime is determined by the creation and termination of the object instance. The program determines the creation time of the objects it declares, but termination involves a more complex mechanism.

Visual Basic 6.0

In Visual Basic 6.0, every object instance maintains a reference count. When the last reference to an instance is released and the count goes to zero, the object is terminated immediately.

Visual Basic .NET

In Visual Basic .NET, a destructor is not necessarily called as soon as the last reference is released. This is because the common language runtime maintains a reference tree instead of individual reference counts. The garbage collector traces the reference tree in the background. If it finds an object or group of objects that have no reference from any currently executing code, it calls the destructors of all such objects. It is impossible to predict either the order of this destruction or the time it takes the garbage collector to trace the reference tree. Therefore, the lifetime of an object is indeterminate.

See Also

Implements Statement | Imports Statement | Object Data Type | Class Statement | Structure Statement | Module Statement | Sub Statement | New | System Namespace | Object Lifetime: How Objects are Created and Destroyed | Automatic Memory Management | Programming Element Support Changes Summary

Procedure Changes

The following topics describe changes to procedures between Visual Basic 6.0 and Visual Basic .NET.

In This Section

Parameter Passing Mechanism Changes in Visual Basic
Describes passing arguments to procedures.
Procedure Calling Sequence Changes in Visual Basic
Discusses procedure calls and returns.
Procedure Declaration Changes in Visual Basic
Describes optional procedure arguments and static local variables.

Related Sections

Programming Element Support Changes Summary
Provides an alphabetical list of programming elements that have changed or are no longer supported in Visual Basic .NET, and includes their replacements.
Language Changes in Visual Basic
Provides an overview of new and changed features in Visual Basic .NET.

Parameter Passing Mechanism Changes in Visual Basic

Visual Basic .NET introduces several changes affecting the mechanism of passing arguments to procedures.

Default Passing Mechanism

Visual Basic 6.0

In Visual Basic 6.0, if you do not specify **ByVal** or **ByRef** for a procedure argument, the passing mechanism defaults to **ByRef**. This allows the variable passed into the procedure to be modified in the calling program.

Exposing a variable to modification can lead to a pitfall. In the following example, the passing mechanism defaults to **ByRef**, the value of `ElapsedSeconds` gets altered by **MinutesPastHour**, and `ElapsedSeconds` is displayed incorrectly by **MsgBox**:

```
Function MinutesPastHour(Seconds As Integer) As Integer
   Dim Hours As Integer = Seconds \ 3600
   Seconds = Seconds Mod 3600
   MinutesPastHour = Seconds \ 60
End Function
' ...
ElapsedSeconds = CInt(Timer( ))  ' Integer seconds since midnight.
ExtraMinutes = MinutesPastHour(ElapsedSeconds)
MsgBox "Total seconds: " & ElapsedSeconds & _
       "; minutes past hour: " & ExtraMinutes
```

Passing an argument **ByRef** allows a procedure to change it in the calling program, which can lead to unexpected behavior. And if that procedure calls another procedure and passes the same argument **ByRef**, the chances of unintentionally changing the original variable are increased.

Visual Basic .NET

When you declare a procedure in Visual Basic .NET, the passing mechanism defaults to **ByVal** for every argument. This protects arguments against modification. The declaration in the preceding example can be rewritten as follows:

```
Function MinutesPastHour(ByVal Seconds As Integer) As Integer
```

Passing `Seconds` by value prevents the procedure from accessing the variable in the calling program, and helps avoid the pitfall just described.

Although **ByVal** is the default mechanism, specifying it explicitly for every argument removes uncertainty and makes your code easier to read.

ByRef Property Arguments

Visual Basic 6.0

In Visual Basic 6.0, a property passed to a procedure as a **ByRef** argument is copied into the procedure but not copied out. This means that any modification to such a property argument is not reflected back to the original property in the calling program, even though it was passed **ByRef**.

Visual Basic .NET

In Visual Basic .NET, a property argument passed **ByRef** is copied both into and out of the procedure. The following example demonstrates how a property can be changed by a procedure:

```
Sub Reduce(ByRef Height As Single)
' ... ... ... ... ... ... ...  ' Code to modify Height argument.
End Sub
' ...
Dim Sq As Square = New Square  ' Square has property Side of type Single.
Reduce(Sq.Side)                ' Side is changed when Reduce returns.
```

When a procedure modifies a property argument, the value of the original property is not changed immediately in the calling program. Instead, it is copied out when the procedure returns.

ParamArray Arguments

Visual Basic 6.0

In Visual Basic 6.0, a procedure can specify the **ParamArray** keyword on the last of its arguments to accept an array of **Variant** arguments. You cannot declare the passing mechanism of these arguments. They are always passed **ByRef**.

Visual Basic .NET

In Visual Basic .NET, **ParamArray** arguments are always passed **ByVal**. The arguments in the array must all be of the data type of the **ParamArray** argument.

See Also

Procedure Declaration Changes in Visual Basic | Procedure Calling Sequence Changes in Visual Basic | Parameter Arrays | Function Statement | Sub Statement | ByVal | ByRef | ParamArray | Programming Element Support Changes Summary

Procedure Calling Sequence Changes in Visual Basic

Visual Basic .NET introduces several changes affecting procedure-calling sequences. These changes improve syntax consistency.

Parentheses in Procedure Calls

Visual Basic 6.0

In Visual Basic 6.0, parentheses are required around the argument list in **Function** calls. In **Sub** calls, they are required if you use the **Call** statement and forbidden if you do not. The following example shows valid calling statements:

```
Y = Sqrt(X)
Call DisplayCell(2, 14, Value)
DisplayCell 2, 14, Value  ' Variation on previous statement.
```

Visual Basic .NET

In Visual Basic .NET, parentheses are always required around a non-empty argument list in any procedure call. In **Sub** calls, the **Call** statement is optional. The preceding example can be rewritten as follows:

```
Y = Sqrt(X)
DisplayCell(2, 14, Value)
```

If you are calling a procedure without supplying any arguments, you can include empty parentheses or leave them out altogether.

Return Statement

Visual Basic 6.0

In Visual Basic 6.0, you use the **Return** statement only to branch back to the code following a **GoSub** statement. Both statements must be in the same procedure.

Visual Basic .NET

In Visual Basic .NET, the **GoSub** statement is not supported, and you can use the **Return** statement to return control to the calling program from a **Function** or **Sub** procedure.

See Also

Declaration Syntax Changes in Visual Basic | Procedure Declaration Changes in Visual Basic |
Parameter Passing Mechanism Changes in Visual Basic | Control Statement Changes in Visual Basic | Return Statement |
Call Statement | Programming Element Support Changes Summary

Procedure Declaration Changes in Visual Basic

Visual Basic .NET introduces several changes related to the declaration of procedures.

Optional Arguments

Visual Basic 6.0

In Visual Basic 6.0, you can declare a procedure argument as **Optional** without specifying a default value. If an optional argument is of the **Variant** type, the procedure code can use the **IsMissing** function to determine whether the argument is present.

Visual Basic .NET

In Visual Basic .NET, every optional argument must declare a default value, which is passed to the procedure if the calling program does not supply that argument. The **IsMissing** function is not needed to detect a missing argument, and it is not supported. The following example shows an optional argument declaration:

```
Sub Calculate(Optional ByVal Switch As Boolean = False)
```

Static Local Variables

Visual Basic 6.0

In Visual Basic 6.0, you can declare a procedure with the **Static** modifier. This causes every local variable within the procedure to be static and to retain its value between calls.

Visual Basic .NET

In Visual Basic .NET, the **Static** keyword is not supported in a **Function** or **Sub** statement. You must individually declare each local variable you want to be **Static**.

See Also

Procedure Calling Sequence Changes in Visual Basic | Parameter Passing Mechanism Changes in Visual Basic | Optional | Static | Programming Element Support Changes Summary

Property Changes

The following topics describe changes to properties between Visual Basic 6.0 and Visual Basic .NET.

In This Section

Default Property Changes in Visual Basic
Covers changes to default property support.
Property Changes in Visual Basic
Covers changes to property procedures and arguments.

Related Sections

Programming Element Support Changes Summary
Provides an alphabetical list of programming elements that have changed or are no longer supported in Visual Basic .NET, and includes their replacements.
Language Changes in Visual Basic
Provides an overview of new and changed features in Visual Basic .NET.

Default Property Changes in Visual Basic

Visual Basic .NET updates default property support for simplification and improved readability.

Visual Basic 6.0

In Visual Basic 6.0, default properties are supported on objects. On a **Label** control, for example, **Caption** is the default property, and the two assignments in the following example are equivalent:

```
Dim L As Label
L = "Important"
L.Caption = "Important"
```

While default properties enable a certain amount of shorthand in writing Visual Basic code, they have several drawbacks:

- They can make code more difficult to read. In the preceding example, if you are not familiar with the **Label** control, you cannot tell from the first assignment whether the string `"Important"` is being stored directly in the variable `L` or in a default property.
- Given an object that you plan to use in your code, it is not always easy to discover whether it has a default property, and if so, which property that is.
- Default properties make the **Set** statement necessary in the Visual Basic language. The following example shows how **Set** is needed to indicate that an object reference, rather than a default property, is to be assigned:

```
Dim L1 As Label, L2 As Label
L1 = "Saving"  ' Assign a value to L1's Caption property.
L2 = L1        ' Replace L2's Caption property with L1's.
Set L2 = L1    ' Replace L2 with an object reference to L1.
```

Visual Basic .NET

In Visual Basic .NET, default properties are not supported unless they take arguments. Because of this syntax change, the **Let** and **Set** statements are not needed to specify what is to be assigned, and they are not used in assignment statements. The **Text** property replaces the **Caption** property on the **Label** control, and the preceding example can be rewritten as follows:

```
Dim L1, L2 As Label   ' Both become type Label in Visual Basic .NET.
L1.Text = "Saving"    ' Assign Text property.
L2.Text = L1.Text     ' Copy Text property.
L2 = L1               ' Copy object reference.
```

Let is still a reserved word in Visual Basic .NET, even though it has no syntactical use. This helps avoid confusion with its former meanings. Visual Basic .NET uses the **Set** statement for property procedures that set the value of a property.

Parameterized Properties

Default properties that take arguments are not ambiguous, and they are supported in Visual Basic .NET. Default properties appear most commonly on collection classes. In the **System.Windows.Forms** namespace, for example, the **Form** class supports the following hierarchy:

Form object

Controls property (returns a **ControlCollection** object for this form)

ControlCollection object (default property is **Item**)

Item property (returns a **Control** object for one item in the collection)

Control object

The **Form.Controls** property returns a **ControlCollection** object, and the **ControlCollection.Item** property returns a **Control** object. The following example shows both valid and invalid use of default properties in Visual Basic .NET:

```
Dim F As Form   ' Assume F has been created and initialized.
' ...
F.Controls.Item(0).Text = "Stop"   ' Valid -- no default properties used.
F.Controls(0).Text = "Stop"   ' Valid -- Item is parameterized.
' ...
F(0).Text = "Stop"   ' INVALID -- Form does not have a default property.
F.Controls(0) = "Stop"   ' INVALID -- No default property on Control.
```

Default Property Declaration

In Visual Basic .NET, you specify a property as the default property by beginning its declaration with the **Default** keyword. If you overload the property name, you must specify **Default** in every overload declaration. You cannot declare a default property to be **Shared** or **Private**.

See Also

Property Changes in Visual Basic | Control.Text Property | Label Class | Set Statement | System.Windows.Forms Namespace | Form Class | Control.ControlCollection Class | Control Class | Default | Programming Element Support Changes Summary

Property Changes in Visual Basic

Visual Basic .NET updates the declaration of property procedures and arguments for simplification and interoperability with other programming languages.

Visual Basic 6.0

In Visual Basic 6.0, you use the **Property Get**, **Property Let**, and **Property Set** statements to get and set property values.

Passing a variable the **ByRef** property argument allows a procedure to change that variable in the calling code.

Visual Basic .NET

Visual Basic .NET introduces a unified property declaration syntax that includes the procedures for getting and setting the property's value. This guarantees consistency of property attributes such as access level and overloading. The following example shows the declaration of a property that does not take arguments:

```
Private MonthNum As Integer = 1  ' Internal storage for property value.
Property Month( ) As Integer
   Get
      Return MonthNum
   End Get
   Set(ByVal Value As Integer)
      If Value < 1 Or Value > 12 Then
         ' Error processing for invalid value.
      Else
         MonthNum = Value
      End If
   End Set
End Property   ' Month
```

You can use any name you want for the argument to **Set**. If you do not supply an argument, `Value` is automatically generated.

This syntax change makes the **Property Get** and **Property Set** statements unnecessary, and they are not supported. Because Visual Basic .NET does not allow default properties without arguments, **Property Let** and **Property Set** are not needed to distinguish whether an object reference or a default property is to be assigned. Therefore, the **Property Let** statement also is not supported.

Visual Basic .NET does not support **ByRef** property arguments. If a property procedure has a **ByRef** argument and changes the variable underlying that argument, the property can behave in an unexpected manner. Therefore, all declarations of parameterized properties must specify **ByVal** for the arguments.

See Also

Default Property Changes in Visual Basic | ByVal | Programming Element Support Changes Summary

Control Flow Changes

The following topics describe changes to control flow between Visual Basic 6.0 and Visual Basic .NET.

In This Section

Control Statement Changes in Visual Basic
Covers changes to statements controlling execution flow.
Exception Handling Changes in Visual Basic
Discusses new support for structured exception handling, and continued support for unstructured exception handling.

Related Sections

Programming Element Support Changes Summary
Provides an alphabetical list of programming elements that have changed or are no longer supported in Visual Basic .NET, and includes their replacements.
Language Changes in Visual Basic
Provides an overview of new and changed features in Visual Basic .NET.

Control Statement Changes in Visual Basic

Visual Basic .NET updates several statements controlling the flow of execution.

Visual Basic 6.0

In Visual Basic 6.0, the **GoSub** statement calls a sub-procedure within a procedure. The **On ... GoSub** and **On ... GoTo** constructions, also known as computed **GoSub** and computed **GoTo**, provide compatibility with early versions of BASIC.

The **While ... Wend** construction loops through code while a specified condition is true.

Visual Basic .NET

In Visual Basic .NET, you can call procedures with the **Call** statement, and the **GoSub** statement is not supported. You can perform multiple branching with **Select...Case** statements, and the **On ... GoSub** and **On ... GoTo** constructions are not supported. However, Visual Basic .NET still supports the **On Error** statement.

Visual Basic .NET retains the **While ... Wend** construction, but replaces the **Wend** keyword with the **End** statement. The **Wend** keyword is not supported.

See Also

Procedure Calling Sequence Changes in Visual Basic | Call Statement | Function Statement | Sub Statement | Return Statement | Select...Case Statements | On Error Statement | End Statement | Programming Element Support Changes Summary

Exception Handling Changes in Visual Basic

Visual Basic .NET adds support for structured exception handling, and continues support for unstructured exception handling.

Visual Basic 6.0

In Visual Basic 6.0, you use unstructured exception handling to deal with errors in your code. Placing the **On Error** statement at the beginning of a block of code handles any errors that occur within that block. Unstructured exception handling also employs the **Error** and **Resume** statements.

Visual Basic .NET

In Visual Basic .NET, structured exception handling code detects and responds to errors during execution by combining a control structure with exceptions, protected blocks of code, and filters. Structured exception handling is done through a **Try** statement, which is made up of three kinds of blocks: **Try**, **Catch**, and **Finally**. A **Try** block is a statement block that contains the statement to be executed. A **Catch** block is a statement block that handles an exception. A **Finally** block contains statements to be run when the **Try** statement is exited, regardless of whether an exception has occurred. The **Throw** statement, used in conjunction with a **Catch** block, raises an exception that is represented by an instance of a type derived from the **System.Exception** class.

See Also

Structured Exception Handling | Unstructured Exception Handling | Try...Catch...Finally Statements (Conceptual) | Try...Catch...Finally Statements (Language Reference) | Throw Statement | On Error Statement | Error Statement | Resume Statement | Exception Class | Programming Element Support Changes Summary

Program Structure and Code Conventions

This section introduces the typical Visual Basic program structure, provides a simple Visual Basic program, "Hello World", and discusses Visual Basic code conventions. Code conventions are suggestions that focus not on a program's logic but on its physical structure and appearance. Following them makes your code easier to read, understand, and maintain. Code conventions can include, among others:

- Standardized formats for labeling and commenting code.
- Guidelines for spacing, formatting, and indenting code.
- Naming conventions for objects, variables, and procedures.

The following topics present a set of programming guidelines for Visual Basic programs, along with examples of good usage.

In This Section

Structure of a Visual Basic Program
Provides an overview of the elements that make up a Visual Basic program.
The Visual Basic Version of Hello World!
Provides a step-by-step recreation of the classic program "Hello World," redone in Visual Basic .NET.
Visual Basic Naming Conventions
Includes general guidelines for naming procedures, constants, variables, arguments, and objects.
Breaking and Combining Statements in Code
Shows how to divide long statements into multiple lines and combine short statements on one line.
Special Characters in Code
Shows how and where to use non-numeric and non-alphabetic characters.
Comments in Code
Discusses how to add descriptive comments to your code.
Visual Basic Limitations
Discusses the removal of known coding limits within Visual Basic.
Conditional Compilation
Provides links to topics about how to compile particular blocks of code selectively while directing the compiler to ignore others.

Related Sections

Typographic and Code Conventions
Provides standard coding conventions for Visual Basic .NET.
Coding Techniques and Programming Practices
Provides an overview of coding techniques throughout Visual Studio .NET.
Editing Code, HTML, and Text
Provides an overview of the Visual Studio .NET Code Editor.

Structure of a Visual Basic Program

A Visual Basic program has standard building blocks.

Visual Basic code is stored in project modules. Projects are composed of files, which are compiled into applications. When you start a project and open the code editor, you see some code already in place and in the correct order. Any code that you write should follow this sequence:

1. **Option** statements
2. **Imports** statements
3. The **Main** procedure
4. **Class**, **Module**, and **Namespace**, if applicable, statements

In addition, a program may contain conditional compilation statements. These can be placed anywhere in the module. Many programmers prefer to put them at the end.

If you enter statements in a different order, compilation errors may result.

Option Statements

Option statements establish ground rules for subsequent code, helping prevent syntax and logic errors. The **Option Explicit** statement ensures that all variables are declared and spelled correctly, which cuts back on time spent debugging later. The **Option Strict** statement helps prevent logic errors and data loss that can occur when you work between variables of different types. The **Option Compare** statement specifies the way strings are compared to each other, either by their **Binary** or **Text** arrangement.

Imports Statements

Imports statements allow you to name classes and other types defined within the imported namespace without having to qualify them.

Main Procedure

The **Main** procedure is the "starting point" for your application — the first procedure that is accessed when you run your code. **Main** is where you would put the code that needs to be accessed first. In **Main**, you can determine which form is loaded first when the program is started, find out if a copy of your application is already running on the system, establish a set of variables for your application, or open a database that the application requires. There are four varieties of **Main**:

- `Sub Main()`
- `Sub Main(ByVal CmdArgs() As String)`
- `Function Main() As Integer`
- `Function Main(ByVal CmdArgs() As String) As Integer`

The most common variety of this procedure is **Sub Main( )**. For more information, see The Visual Basic Version of Hello World!.

Class, Module, and Namespace Statements

Classes and modules make up most of the code in your source file. They contain most of the code you write — mainly **Sub**, **Function**, **Method** and **Event** statements — along with variable declarations and other code needed to make your application run.

Conditional Compilation Statements

Conditional compilation statements can appear anywhere within the module. They are set to execute if certain conditions are met during run time. You can also use them for debugging your application, since conditional code runs in debugging mode only.

See Also

Visual Basic Naming Conventions | Visual Basic Limitations | Option Compare Statement | Option Explicit Statement | Option Strict Statement | Imports Statement | Class Statement | Module Statement | Conditional Compilation Overview | Anatomy of the Visual Basic Code Behind Windows Forms | Sub Statement | The Visual Basic Version of Hello World!

The Visual Basic Version of Hello World!

The following console program is the Visual Basic version of the traditional "Hello World!" program, which displays the string `Hello World!`.

```
' A "Hello World!" program in Visual Basic.
Module Hello
   Sub Main()
      MsgBox("Hello World!")   ' Display message on computer screen.
   End Sub
End Module
```

The important points of this program are the following:

- Comments
- The **Main** procedure
- Input and output
- Compilation and execution

Comments

The first line of the example contains a comment:

```
' A "Hello World!" program in Visual Basic.
```

The single quote (') means that the rest of the line is a comment, and will be ignored by the compiler. You can make an entire line a comment, or you can append a comment to the end of another statement, as follows:

```
      MsgBox("Hello World!")   ' Display message on computer screen.
```

The Main Procedure

Every Visual Basic program must contain a procedure called `Main`. This procedure serves as the starting point and overall control for your application. It is called when your module is loaded.

> **Note** If you declare the `Main` procedure in a class, you must use the **Shared** keyword. In a module, `Main` does not need to be **Shared**.

There are four ways to declare the `Main` procedure, as follows:

- The simplest way is to declare a **Sub** procedure that does not take arguments or return a value, as in the earlier console program example:

  ```
  Sub Main()
  ```

- `Main` can also return an **Integer** value, which the operating system uses as the exit code for your program. Other programs can test this code by examining the Windows ERRORLEVEL value. To return an exit code, you must declare `Main` as a **Function** procedure instead of a **Sub** procedure, as follows:

  ```
  Function Main() As Integer
     MsgBox("Hello World!")   ' Display message on computer screen.
     Return 0   ' Zero usually means successful completion.
  End Function
  ```

- `Main` can also take a **String** array as an argument. Each string in the array contains one of the command-line arguments used to invoke your program. You can take different actions depending on their values. To obtain the command-line arguments, you declare `Main` as follows:

  ```
  Function Main(ByVal CmdArgs() As String) As Integer
  ```

```
    Dim ArgNum As Integer   ' Index of individual command-line argument.
    If CmdArgs.Length > 0 Then   ' See if there are any arguments.
       For ArgNum = 0 To UBound(CmdArgs)
          ' Examine CmdArgs(ArgNum) for settings you need to handle.
       Next ArgNum
    End If
    MsgBox("Hello World!")   ' Display message on computer screen.
    Return 0   ' Zero usually means successful completion.
 End Function
```

- You can declare `Main` to examine the command-line arguments but not return an exit code, as follows:

```
Sub Main(ByVal CmdArgs() As String)
```

For more information on the `Main` procedure, see Structure of a Visual Basic Program.

Input and Output

This example uses the standard Visual Basic run-time library, which is available through the **Microsoft.VisualBasic** namespace. You can use all the procedures and properties of **Microsoft.VisualBasic** without having to import it in your program.

The `Main` procedure calls the **MsgBox** function to display a message box containing the string `Hello World!`:

```
        MsgBox("Hello World!")   ' Display message on computer screen.
```

For more information on Windows Forms, see Microsoft.VisualBasic Namespace.

Compilation and Execution

You can compile the "Hello World!" program using either the Visual Studio .NET integrated development environment (IDE) or the command line.

To compile and run the program from the command line

1. Create the source file using any text editor and save it with a file name such as `Hello.vb`.
2. To invoke the compiler, enter the following command:

```
vbc Hello.vb
```

3. If your program does not contain any compilation errors, the compiler creates a `Hello.exe` file.
4. To run the program, enter the following command:

```
Hello
```

You can optionally include the **/main** command-line compiler option in the `vbc` command to specify the namespace and module supplying `Main`.

To compile and run the program from the IDE

1. Create a Visual Basic console application project.
2. Copy the code into the project.
3. Choose the appropriate Build command from the **Build** menu, or F5 to build and run (corresponding to Start in the **Debug** menu).

For more information on the Visual Basic compiler and its options, see Building From the Command Line.

See Also

Structure of a Visual Basic Program | Microsoft.VisualBasic Namespace | Building From the Command Line | Array.Length Property | /main

Visual Basic Naming Conventions

When you name an element in your Visual Basic .NET application, the first character of that name must be an alphabetic character, a digit, or an underscore. The following suggestions also apply to naming:

- Begin each separate word in a name with a capital letter, as in `FindLastRecord` and `RedrawMyForm`.
- Begin function and method names with a verb, as in `InitNameArray` or `CloseDialog`.
- Begin class and property names with a noun, as in `EmployeeName` or `CarAccessory`.
- Begin interface names with the prefix "I", followed by a noun or a noun phrase, like `IComponent`, or with an adjective describing the interface's behavior, like `IPersistable`. Do not use the underscore, and use abbreviations sparingly, because abbreviations can cause confusion.
- Begin event handler names with a noun describing the type of event followed by the "`EventHandler`" suffix, as in "`MouseEventHandler`".
- In names of event argument classes, include the "`EventArgs`" suffix.
- If an event has a concept of "before" or "after," use a prefix in present or past tense, as in "`ControlAdd`" or "`ControlAdded`".
- For long or frequently used terms, use abbreviations to keep name lengths reasonable, for example, "HTML", instead of "Hypertext Markup Language". In general, variable names greater than 32 characters are difficult to read on a monitor set to a low resolution. Also, make sure your abbreviations are consistent throughout the entire application. Randomly switching in a project between "HTML" and "Hypertext Markup Language" will lead to confusion.
- Avoid using names in an inner scope that are the same as names in an outer scope. Errors will result if the wrong variable is accessed. If a conflict occurs between a variable and the keyword of the same name, you must identify the keyword by preceding it with the appropriate type library. For example, if you have a variable called `Date`, you can use the intrinsic **Date** function only by calling **System.Date**.

See Also

Keywords as Element Names in Code | Program Structure and Code Conventions |
Visual Basic Language and Run-Time Reference

Keywords as Element Names in Code

Any program element — such as a variable, class, or member — can have the same name as a restricted keyword. For example, you can create a variable named `Loop`. However, to refer to your version of it — which has the same name as the restricted **Loop** keyword — you must either qualify it by preceding it with its full namespace, or enclose it in square brackets (**[ ]**), as in the following examples:

```
MyForm.Loop.Visible = True   ' Qualified with the full namespace.
[Loop].Visible = True   ' Square brackets also work.
```

If you do not, Visual Basic assumes use of the intrinsic **Loop** keyword and produces an error, as in the following example:

```
Loop.Visible = True   ' Causes an error.
```

You can use square brackets when referring to forms and controls, and when declaring a variable or defining a procedure with the same name as a restricted keyword. It can be easy to forget to qualify names or include square brackets, and thus introduce errors into your code and make it harder to read. For this reason, it is recommended that you not use restricted keywords as the names of program elements. However, if a future version of Visual Basic defines a new keyword that conflicts with an existing form or control name, you can use this technique when updating your code to work with the new version.

> **Note** Your program also may include element names provided by other referenced assemblies. If these names conflict with restricted keywords, placing square brackets around them forces Visual Basic to accept them.

See Also

Visual Basic Naming Conventions | Program Structure and Code Conventions | Keywords

Conditional Compilation

In *conditional compilation*, particular blocks of code in a program are compiled selectively while others are ignored.

In This Section

Conditional Compilation Overview
Explains what conditional compilation is used for, and discusses the **#Const** and **#If...Then...#Else** directives.

Declaring Conditional Compilation Constants
Shows how to declare conditional compilation constants in the **Property Pages** dialog box, at the command line, or in your code.

Collapsing and Hiding Sections of Code
Explains how to specify a block of code that you can expand or collapse using the Visual Studio .NET outlining feature.

Related Sections

Program Structure and Code Conventions
Provides suggestions for making your code easy to read and maintain.

Compiling to MSIL
Explains how your source code is translated into Microsoft Intermediate language (MSIL) by the compiler.

Compiling MSIL to Native Code
Provides an overview of the .NET framework JIT (just-in-time) compiler that converts MSIL to native (CPU-specific) code.

Conditional Compilation Overview

You can use conditional compilation to select particular sections of code to compile while excluding other sections. For example, you may want to write debugging statements that compare the speed of different approaches to the same programming task, or you may want to localize an application for multiple languages. Conditional compilation statements are designed to run during compile time, not at run time.

You declare a conditional compiler constant in code with the **#Const** directive, and you denote blocks of code to be conditionally compiled with the **#If...Then...#Else** directive. For example, to create French and German language versions of the same application from the same source code, you embed platform-specific code segments in **#If...Then** statements using the predefined constants `FrenchVersion` and `GermanVersion`. The following example demonstrates how:

```
#If FrenchVersion Then
   ' <code specific to the French language version>.
#ElseIf GermanVersion Then
   ' <code specific to the German language version>.
#Else
   ' <code specific to other versions>.
#End If
```

If you set the value of the `FrenchVersion` constant to **True** at compile time, the conditional code for the French version is compiled. If you set the value of the `GermanVersion` constant to **True**, the compiler uses the German version. If neither is set to **True**, the code in the last **Else** block runs.

> **Note** Autocompletion will not function when editing code and using conditional compilation directives if the code is not part of the current branch.

See Also

Declaring Conditional Compilation Constants | Collapsing and Hiding Sections of Code | Building from the Command Line | #Const Directive | #If...Then...#Else Directives

Declaring Conditional Compilation Constants

You can set conditional compilation constants in one of three ways:

- In the Property Pages dialog box
- At the command line when using the command-line compiler
- In your code

Conditional compilation constants have a special scope and cannot be accessed from standard code. The scope of a conditional compilation constant is dependant on the way it is set. The following table lists the scope of constants declared using each of the three ways mentioned above.

How Constant is Set	Scope of Constant
Property Pages dialog box	Public to all files in the project
Command line	Public to all files passed to the command-line compile r
#Const statement in code	Private to the file in which it is declared

To set constants in the Property Pages dialog box

- Before creating your executable file, set constants in the Property Pages dialog box by following the steps provided in Setting Visual Basic Project Properties.

To set constants at the command line

- Use the **/d** switch to enter conditional compilation constants, as in the following example:

  ```
  vbc MyProj.vb /d:conFrenchVersion=-1:conANSI=0
  ```

  No space is required between the **/d** switch and the first constant.

Command-line declarations override declarations entered in the Property Pages dialog box, but do not erase them. Arguments set in Property Pages remain in effect for subsequent compilations.

When writing constants in the code itself, there are no strict rules as to their placement, since their scope is the entire module in which they are declared.

To set constants in your code

- Place the constants in the declaration block of the module in which they are used. This helps keep your code organized and easier to read.

See Also

Conditional Compilation Overview | Collapsing and Hiding Sections of Code | Building From the Command Line | #If...Then...#Else Directives | #Const Directive

Collapsing and Hiding Sections of Code

The **#Region** directive enables you to collapse and hide sections of code in Visual Basic .NET files. **#Region** lets you specify a block of code that you can expand or collapse when using the Outlining and Hiding Code feature of the Visual Studio .NET Code editor. The ability to hide code selectively makes your files more manageable and easier to read.

#Region directives support code block semantics such as **#If...#End If**. This means they cannot begin in one block and end in another; the start and end must be in the same block. **#Region** directives are not supported within functions.

Syntax

The syntax for **#Region** is as follows:

```
#Region "identifier string"
'code region
#End Region
```

The `identifier string` requires the quotes and acts as the title of the region when it is collapsed. The region is collapsed by default.

To collapse and hide a section of code

- Place the section of code between the **#Region** and **#End Region** statements, as in the following example:

```
#Region "This is the code to be collapsed"
    Private components As System.ComponentModel.Container
    Dim WithEvents Form1 As System.Windows.Forms.Form

   Private Sub InitializeComponent()
      components = New System.ComponentModel.Container
      Me.Text = "Form1"
   End Sub
#End Region
```

Multiple #Region Blocks

The **#Region** block can be used multiple times in a code file; thus, users can define their own blocks of procedures and classes that can, in turn, be collapsed. **#Region** blocks also can be nested within other **#Region** blocks.

> **Note** Hiding code does not prevent it from being compiled, and will not affect the **#If...#End If** statements.

See Also

Conditional Compilation Overview | Declaring Conditional Compilation Constants | #Region Directive | #If...Then...#Else Directives | Outlining and Hiding Code

Breaking and Combining Statements in Code

When writing your code, you may at times create lengthy statements that necessitate horizontal scrolling in the Code Editor. While this does not affect the way your code runs, it makes it difficult for you or anyone else to read the code as it appears on the monitor. In such cases, you should consider breaking the single long statement into several lines.

At other times, you may want to consolidate statements on a single line; for instance, when you have several very short statements and want to conserve space. This feature can also be handy when organizing variables or commands within a module.

To break a single statement into multiple lines

- Use the line-continuation sequence, which is a space followed by an underscore (_), at the point at which you want the line to break. In the following example, the statement is broken into four lines with line-continuation sequences terminating all but the last line:

```
Data1.RecordSource = _
  "SELECT * FROM Titles, Publishers" _
& "WHERE Publishers.PubId = Titles.PubID" _
& "AND Publishers.State = 'CA'"
```

Using this sequence makes your code easier to read, both online and when printed.

> **Note** The line-continuation sequence is necessarily the last thing on a line. You cannot follow it with anything else on the same line.

Some limitations exist as to where the line-continuation sequence can be used, such as in the middle of an argument name. You can break an argument list with the line-continuation sequence, but the individual names of the arguments must remain intact.

> **Note** You cannot continue a comment by using a line-continuation sequence. Once a comment begins, the compiler does not examine the characters for special meaning. For a multiple-line comment, repeat the comment symbol (') on each line.

While placing each statement on a separate line is the recommended method, Visual Basic also allows you to place multiple statements on the same line.

To place multiple statements on the same line

- Separate the statements with a colon (**:**), as in the following example:

```
Text1.Text = "Hello" : Red = 255 : Text1.BackColor = Red
```

See Also

Program Structure and Code Conventions | Statements

Labeling Statements

Statement blocks are made up of lines of code delimited by colons. Lines of code preceded by an identifying string or integer are said to be *labeled*. Statement labels are used to mark a line of code to identify it for use with statements such as **On Error Goto**.

> **Note** Labels can be used only on executable statements inside methods.

The following lines of code are labeled with `Jump` and `120`, respectively:

```
Jump: FileOpen(1, "MYFILE", OpenMode.Input)
120: FileClose (1)
```

Labels may be either valid Visual Basic .NET identifiers — such as those that identify programming elements — or integer literals. A label must appear at the beginning of a line of source code and must be followed by a colon, regardless of whether it is followed by a statement on the same line.

The compiler identifies labels by checking whether the beginning of the line matches any already-defined identifier. If it does not, the compiler assumes it is a label.

Labels have their own declaration space and do not interfere with other identifiers. A label's scope is the body of the method. Label declaration takes precedence in any ambiguous situation.

To label a line of code

- Place an identifier, followed by a colon, at the beginning of the line of source code.

See Also

Statements | Statements Overview | Program Structure and Code Conventions

Special Characters in Code

At times, you need to use special characters in your code, that is, characters that are not on the standard alphanumeric list. The punctuation and special characters in the Visual Basic .NET character set have various uses, from organizing program text to defining the tasks that the compiler or the compiled program carries out. They do not specify an operation to be performed.

Parentheses

Use parentheses when defining a procedure, such as a **Sub** or **Function**. You must enclose all procedure arguments in parentheses. You also use parentheses for putting variables or arguments into logical groups.

Separators

Separators do what their name suggests: they separate sections of code. In Visual Basic, the separator character is the colon (**:**). Use separators when you want to place multiple statements on a single line instead of separate lines, saving space and improving the readability of your code. The following example shows three statements separated by colons (**:**):

```
a = 5: b = 10: c = 15
```

Concatenation

Use the "**&**" operator for *concatenation*, or linking strings together. Do not confuse it with the **+** operator, which adds together numeric values. Using the "**+**" operator to concatenate may cause incorrect results when operating on two numeric values. The following code provides an example:

```
Var1 = "10.01"
Var2 = 11
Result = Var1 + Var2   ' Result = 21.01
Result = Var1 & Var2   ' Result = 10.0111
```

Member Access Operators

To access a member of a type, you use the dot (**.**) or exclamation point (**!**) operator between the type name and the member name. The type can be an enumeration, structure, interface, or class, and the member can be a field, property, event, or method.

Dot (.) operator

Use the **.** operator as a member access operator for properties, events, fields, and methods, as in the following example:

```
Dim MyForm As New System.Windows.Forms.Form
MyForm.Text = "This is my form"   ' Accesses Text member (property) of Form class (on MyForm
object).
MyForm.CenterToScreen()   ' Accesses CenterToScreen member (method) on MyForm.
```

Exclamation point (!) operator

Use the **!** operator (in a class or interface only) as a dictionary access operator. The class or interface must have a default property that accepts a single **String** argument. The identifier immediately following the **!** operator becomes the string argument to the default property, as in the following example:

```
Public Class HasDefault   ' Exposes a default property.
   Default Public ReadOnly Property Index(ByVal S As String) As Integer
      Get
         Return 1000 + CInt(S)
      End Get
   End Property   ' Index
End Class   ' HasDefault
' ...
Public Class TestHasDefault
   Public Sub CompareAccess()
      Dim HD As HasDefault = New HasDefault()
```

```
        Dim IndexStr As String = "5"
        MsgBox("Traditional access returns " & HD.Index(IndexStr) & vbCrLf & _
               "Default property access returns " & HD(IndexStr) & vbCrLf & _
               "Dictionary access returns " & HD!IndexStr)
    End Sub
End Class   ' TestHasDefault
```

The three output lines of **MsgBox** all display the value 1005. The first line uses the traditional access to property `Index`, the second makes use of the fact that `Index` is the default property of class `HasDefault`, and the third uses dictionary access to the class.

Note that the second operand of the **!** operator must be a valid identifier or keyword. Therefore, the following change to the last line of the **MsgBox** call generates an error because "5" is not an identifier or keyword:

```
               "Dictionary access returns " & HD!"5")
```

> **Note** References to default collections must be explicit. In particular, you cannot use the **!** operator on a late-bound variable.

The **!** character is also used as the **Single** type character.

See Also

Program Structure and Code Conventions | Type Characters

Comments in Code

As you read the code examples, you often encounter the comment symbol ('). This symbol tells the Visual Basic compiler to ignore the text following it, or the *comment*. Comments are brief explanatory notes added to code for the benefit of those reading it.

It is good programming practice to begin all procedures and functions with a brief comment describing the functional characteristics of the procedure (what it does), for your own benefit and the benefit of anyone else who examines the code. You should separate the implementation details (how the procedure does it) from comments that describe the functional characteristics. When you include implementation details in the description, remember to update them when you update the function.

Comments can follow a statement on the same line, or occupy an entire line. Both are illustrated in the following code:

```
' This is a comment beginning at the left edge of the screen.
Text1.Text = "Hi!"   ' This is an inline comment.
```

If your comment requires more than one line, use the comment symbol on each line:

```
' The text of this comment is too long to fit on a single line, so we
' break it into two lines. Some comments might need three or more lines.
```

The following chart provides general guidelines for what types of comments should be listed in a particular section of code. These are suggestions; Visual Basic does not enforce "rules" for adding comments. Write what works best, both for you and for anyone else who reads your code:

Section heading	Comment description
Purpose	Describes what the procedure does (not how it does it)
Assumptions	Lists each external variable, control, open file, or other element accessed by the procedure
Effects	Lists each affected external variable, control, or file, and the effect it has (only if it is not obvi ous)
Inputs	Specifies the purpose of the argument
Returns	Explains the values returned by functions

Remember the following points:

- Every important variable declaration should include an inline comment describing the use of the variable being declared.
- Variables, controls, and procedures should be named clearly enough that inline commenting is needed only for complex implementation details.
- Comments cannot follow a line-continuation sequence on the same line.

You can add or remove comment symbols for a block of code by selecting two or more lines of code and choosing the **Comment** and **Uncomment** buttons on the **Edit** toolbar.

> **Note** You can also add comments to your code by preceding the text with the **REM** keyword. However, the **'** symbol and the **Comment**/**Uncomment** buttons are easier to use and require less space and memory.

See Also

Program Structure and Code Conventions | Rem Statement

Visual Basic Limitations

Earlier versions of Visual Basic enforced boundaries in code, such as the length of variable names, the number of variables allowed in modules, and module size. In Visual Basic .NET, these restrictions have been removed, giving you greater freedom in writing and arranging your code. Physical limits still exist, but they are set so high that there is virtually no chance you will exceed them and cause compile errors within your code.

See Also

Program Structure and Code Conventions

Visual Basic Language Features

The following topics introduce the essential components of Visual Basic .NET, an object-oriented programming language. After creating the interface for your application using forms and controls, you will need to write the code that defines the application's behavior. As with any modern programming language, Visual Basic supports a number of common programming constructs and language elements.

If you've programmed in other languages, much of the material covered in this section may seem familiar. While most of the constructs are similar to other languages, the event-driven nature of Visual Basic introduces some subtle differences.

If you're new to programming, the material in this section serves as an introduction to the basic building blocks for writing code. Once you understand the basics, you can create powerful applications using Visual Basic.

In This Section

Visual Basic .NET and the .NET Framework
Describes the role of Visual Basic .NET in the .NET Framework.
Declared Elements
Covers variables, constants, enumerations, structures, properties, methods, procedures, procedure arguments, function returns, events, delegates, interfaces, and classes.
Data Types
Describes programming elements that refer to what kind of data can be held and how that data is stored.
Variables
Discusses how to declare variables and use them to refer to objects.
Arrays
Discusses making your code more compact and powerful by declaring and using arrays.
Objects
Provides an overview of objects and classes; how they are used; their relationships to each other; and the properties, methods, and events they expose.
Strings
Covers declaring and manipulating variables that contain a series of characters.
Constants and Enumerations
Discusses storing unchanging values for repeated use.
Enumerations Overview
Discusses storing and naming sets of related constant values.
Operators
Describes the code units that manipulate value-returning elements, and how to use them efficiently.
Expressions
Shows how to combine elements with operators to yield new values.
Statements
Provides an overview of the two basic units of instruction in Visual Basic: declaration statements and executable statements.
Procedures
Describes **Sub**, **Function,** and **Property** procedures, as well as advanced topics such as recursive and overloaded procedures.
Control Flow
Shows how to regulate the flow of your program's execution.

Related Sections

Object-Oriented Programming in Visual Basic
Covers object-oriented programming basics.
Visual Basic Language and Run-Time Reference
Provides a complete reference for the Visual Basic language and its run-time elements.

Visual Basic .NET and the .NET Framework

Visual Basic .NET is designed around the .NET Framework, which provides enhanced security, memory management, versioning, and deployment support. The .NET Framework also enables interoperability between objects you create with any .NET programming language. This means you can create objects with Visual Basic .NET that are easy to use from other .NET languages, and you can use objects from other .NET languages just like you use objects created with Visual Basic .NET.

In This Section

Assemblies
Defines assemblies and describes how to use them.
References and the Imports Statement
Discusses how to reference objects in other assemblies.
Namespaces
Describes how namespaces organize objects within assemblies.
Attributes
Discusses how .NET attributes provide additional information about entities defined inside a Visual Basic .NET application.

Related Sections

Adding and Removing References
Covers adding references so you can write code against external components, and describes how to remove unnecessary references.
Getting Started
Provides a hands-on introduction to and overview of the .NET Framework.
Programming with the .NET Framework
Discusses how to use the .NET Framework when developing Visual Studio .NET applications.
Language Changes in Visual Basic
Provides a list and descriptions of changed language elements.

Assemblies

Assemblies form the fundamental unit of deployment, version control, reuse, activation scoping, and security permissions for a .NET-based application. Assemblies take the form of an executable (.exe) file or dynamic link library (.dll) file, and are the building blocks of the .NET Framework. They provide the common language runtime with the information it needs to be aware of type implementations. You can think of an assembly as a collection of types and resources that form a logical unit of functionality and are built to work together.

With Visual Basic .NET, you use the contents of assemblies, and add references to them, in much the same way as you use type libraries with previous versions of Visual Basic. What makes assemblies different from .exe or .dll files in earlier versions of Windows, however, is that they contain all the information you would find in a type library, plus information about everything else necessary to use the application or component.

Assembly Manifest

Within every assembly is an *assembly manifest*. Similar to a table of contents, the assembly manifest contains the following:

- The assembly's identity (its name and version).
- A file table describing all the other files that make up the assembly, including, for example, any other assemblies you created that your .exe or .dll file relies on, or even bitmap or Readme files.
- An *assembly reference list*, which is a list of all external dependencies — .dlls or other files your application needs that may have been created by someone else. Assembly references contain references to both global and private objects. Global objects reside in the global assembly cache, an area available to other applications, somewhat like the System32 directory. The **Microsoft.VisualBasic** namespace is an example of an assembly in the global assembly cache. Private objects must be in a directory at either the same level as or below the directory in which your application is installed.

Because assemblies contain information about content, versioning, and dependencies, the applications you create with Visual Basic .NET do not rely on registry values to function properly. Assemblies reduce DLL conflicts and make your applications more reliable and easier to deploy. In many cases, you can install a .NET-based application simply by copying its files to the target computer.

References

To use an assembly, you must add a reference to it, as described in Adding and Removing References. Next, you use the **Imports** statement to choose the namespace of the items you want to use, as described in References and the Imports Statement. Once an assembly is referenced and imported, all the accessible classes, properties, methods, and other members of its namespaces are available to your application as if their code were part of your source file. A single assembly can contain multiple namespaces, and each namespace can contain a different grouping of items, including other namespaces.

For information on creating assemblies, see Creating and Using Assemblies.

See Also

Creating and Using Assemblies | Assemblies (.NET Framework) | Namespaces | References and the Imports Statement | Adding and Removing References | Imports Statement | Viewing Assembly Contents | Language Changes in Visual Basic | Microsoft.VisualBasic Namespace

Creating and Using Assemblies

An assembly is one or more .exe or .dll files that make up a Visual Studio .NET-based application. Assemblies are created automatically when you compile Visual Studio .NET source files.

To create an assembly

- Compile your application by choosing **Build** from the **Build** menu, or by building it from the command line using the command line compiler. For details about building assemblies from the command line, see Building From the Command Line.

To add a reference to another assembly

- Choose the **Add Reference** command from the **Project** menu, and select the assembly you want to use. For details, see Adding and Removing References.

To use objects in another assembly

- Specify the fully qualified name for the object, or use an alias that has been defined for the object using an **Imports** statement. For details about fully qualified names, see Namespaces. For details about adding references and using the **Imports** statement, see References and the Imports Statement.

See Also

Assemblies (Visual Basic .NET) | Assemblies | Creating Assemblies | Namespaces | References and the Imports Statement | Adding and Removing References | Imports Statement | Viewing Assembly Contents | Language Changes in Visual Basic | Building From the Command Line

References and the Imports Statement

You can make external objects available to your project by choosing the **Add Reference** command from the **Project** menu. References in Visual Basic .NET can point to assemblies, which are like type libraries but contain more information.

Assemblies include one or more namespaces. When you add a reference to an assembly, you can also add an **Imports** statement to a module that controls the visibility of that assembly's namespaces within the module. The **Imports** statement provides a scoping context that lets you use only the portion of the namespace necessary to supply a unique reference.

The **Imports** statement has the following syntax:

```
Imports [| aliasname =] Namespace
```

Aliasname refers to a short name you can use within code to refer to an imported namespace. *Namespace* is a namespace available through either a project reference, through a definition within the project, or through a previous **Imports** statement.

A module may contain any number of **Imports** statements. They must appear after any **Option** statements, if present, but before any other code.

> **Note** Do not confuse project references with the **Imports** statement or the **Declare** statement. Project references make external objects, such as objects in assemblies, available to Visual Basic .NET projects. The **Imports** statement is used to simplify access to project references, but does not provide access to these objects. The **Declare** statement is used to declare a reference to an external procedure in a dynamic-link library (DLL).

Using Aliases with the Imports Statement

The **Imports** statement makes it easier to access methods of classes by eliminating the need to explicitly type the fully qualified names of references. Aliases let you assign a friendlier name to just one part of a namespace. For example, the carriage return-line feed sequence that causes a single piece of text to be displayed on multiple lines is part of the **ControlChars** class in the **Microsoft.VisualBasic** namespace. To use this constant in a program without an alias, you would need to type the following code:

```
    MsgBox("Some text" & Microsoft.VisualBasic.ControlChars.crlf _
                       & "Some more text")
```

Imports statements must always be the first lines immediately following any **Option** statements in a module. The following code fragment shows how to import and assign an alias to the **Microsoft.VisualBasic.ControlChars** namespace:

```
Imports CtrlChrs=Microsoft.VisualBasic.ControlChars
```

Future references to this namespace can be considerably shorter:

```
    MsgBox("Some text" & CtrlChrs.crlf & "Some more text")
```

If an **Imports** statement does not include an alias name, elements defined within the imported namespace can be used in the module without qualification. If the alias name is specified, it must be used as a qualifier for names contained within that namespace.

See Also

Adding and Removing References | Namespaces | Assemblies (Visual Basic .NET) | Creating and Using Assemblies | Imports Statement | Language Changes in Visual Basic | ControlChars (Object Browser) | Microsoft.VisualBasic Namespace

Namespaces

Namespaces organize the objects defined in an assembly. Assemblies can contain multiple namespaces, which can in turn contain other namespaces. Namespaces prevent ambiguity and simplify references when using large groups of objects such as class libraries.

For example, Visual Studio .NET defines the **ListBox** class in the **System.Windows.Forms** namespace. The following code fragment shows how to declare a variable using the fully qualified name for this class:

```
Dim LBox As System.Windows.Forms.ListBox
```

Visual Studio .NET namespaces address a problem sometimes known as *namespace pollution*, in which the developer of a class library is hampered by the use of similar names in another library. These conflicts with existing components are sometimes called *name collisions*.

For example, if you create a new class named `ListBox`, you can use it inside your project without qualification. However, if you want to use the Visual Studio .NET **ListBox** class in the same project, you must use a fully qualified reference to make the reference unique. If the reference is not unique, Visual Basic .NET produces an error stating that the name is ambiguous. The following code example demonstrates how to declare these objects:

```
'   Define a new object based on your ListBox class.
Dim LBC as New ListBox
'   Define a new Windows.Forms ListBox control.
Dim MyLB as New System.Windows.Forms.ListBox
```

The following illustration shows two namespace hierarchies, both containing an object named `ListBox`.

By default, every executable file you create with Visual Basic .NET contains a namespace with the same name as your project. For example, if you define an object within a project named `ListBoxProject`, the executable file, ListBoxProject.exe, contains a namespace called `ListBoxProject`.

Multiple assemblies can use the same namespace. Visual Basic .NET treats them as a single set of names. For example, you can define classes for a namespace called `SomeNameSpace` in an assembly named `Assemb1`, and define additional classes for the same namespace from an assembly named `Assemb2`.

Fully Qualified Names

Fully qualified names are object references that are prefixed with the name of the namespace where the object is defined. You can use objects defined in other projects if you create a reference to the class (by choosing **Add Reference** from the **Project** menu) and then use the fully qualified name for the object in your code. The following code fragment shows how to use the fully qualified name for an object from another project's namespace:

```
Dim LBC As New ListBoxProject.Form1.ListBox()
```

Fully qualified names prevent naming conflicts because the compiler can always determine which object is being used. However, the names themselves can get long and cumbersome. To get around this, you can use the **Imports** statement to define an *alias* — an abbreviated name you can use in place of a fully qualified name. For example, the following code example creates aliases for two fully qualified names, and uses these aliases to define two objects:

```
Imports LBControl = System.Windows.Forms.ListBox

Imports MyListBox = ListBoxProject.Form1.ListBox

Dim LBC As LBControl

Dim MyLB As MyListBox
```

If you use the **Imports** statement without an alias, you can use all the names in that namespace without qualification provided they are unique to the project. If your project contains **Imports** statements for namespaces that contain items with the same name, you must fully qualify that name when you use it. Suppose, for example, your project contained the following two **Imports**

statements:

```
Imports MyProj1 ' This namespace contains a class called Class1.
Imports MyProj2 ' This namespace also contains a class called Class1.
```

If you attempt to use `Class1` without fully qualifying it, Visual Basic .NET produces an error stating that the name `Class1` is ambiguous.

Namespace Level Statements

Within a namespace, you can define items such as modules, interfaces, classes, delegates, enumerations, structures, and other namespaces. You cannot define items such as properties, procedures, variables and events at the namespace level, these items must be declared within containers such as modules, structures, or classes.

See Also

Assemblies (Visual Basic .NET) | Creating and Using Assemblies | References and the Imports Statement | Imports Statement | Language Changes in Visual Basic | ListBox Class | System.Windows.Forms Namespace

Attributes

Attributes are keyword-like tags in which you can specify additional information about entities defined in Visual Basic .NET applications. Attributes, which are saved with an assembly's metadata, annotate programming elements such as types, fields, methods, and properties. Other applications, such as the Visual Basic .NET or ASP.NET compilers, can refer to the information in attributes to determine how objects should be used.

The following sections provide more information about attributes and how they are defined, applied, and retrieved.

In This Section

Attributes Overview
Provides information about attributes and their functionality and capabilities.
Attributes Used in Visual Basic .NET
Lists some of the attributes you are most likely to use in Visual Basic .NET.
Common Uses for Attributes
Describes some of the more common ways that attributes are used.
Application of Attributes
Discusses how attributes are applied to program elements.
Global Attributes
Provides information about attributes that apply to modules or assemblies.
Custom Attributes
Shows how to define your own attributes.

Related Sections

Object-Oriented Programming in Visual Basic
Provides information about object-oriented programming and how it is used.
Metadata and Self-Describing Components
Provides detailed information about the kinds of metadata used in Visual Studio .NET, including attributes.

Attributes Overview

Attributes are descriptive tags that provide additional information about programming elements such as types, fields, methods, and properties. Other applications, such as the Visual Basic compiler, can refer to the extra information in attributes to determine how these items can be used.

Attributes and Metadata

Attributes are saved with the *metadata* of Visual Basic .NET assemblies. Metadata is information that describes every element managed by the runtime. This includes information required for debugging and garbage collection, as well as security attributes, marshaling data, extended class and member definitions, version binding, and any other information the runtime requires.

With attributes, you specify the metadata in much the same way as you use keywords like **Public** and **Private** to provide information about access levels. However, unlike keywords, most attributes are not language-specific. Using attributes, you can extend the capabilities of the Visual Basic language without requiring changes to the compiler.

Functionality and Capabilities of Attributes

Some key points about attributes include:

- You can apply one or more attributes to entire assemblies, modules, or smaller program elements such as classes and properties.
- Attributes can accept arguments in the same way as methods and properties.
- Attributes fall into two groups: those that are used by the .NET Framework and Visual Basic .NET, and custom attributes that are meaningful only to specific applications.
- While the .NET Framework contains many useful attributes, you can define your own custom attributes if necessary.
- Custom attributes are defined in attribute classes based on the **System.Attribute** class. Attribute classes themselves use **AttributeUsage** to provide additional information about how the attribute can be used.
- The process of retrieving metadata from attributes is called *reflection*. Reflection involves using tools that allow objects to retrieve and examine metadata about their own members.

See Also

Common Uses for Attributes | Application of Attributes | Global Attributes | Custom Attributes | Defining Your Own Attributes | Attributes used in Visual Basic .NET | AttributeUsageAttribute Class | Attribute Class | Attributes and Design-Time Support

Attributes Used in Visual Basic .NET

The following table lists some of the more common attributes used in Visual Basic .NET. For information about a specific attribute, type the attribute name in the Index or Search field of the help system.

Attribute	Purpose
COMClassAttribute Class	Indicates to the compiler that the class should be exposed as a COM object. Specific to Visual Basic .NET.
VBFixedStringAttribute Class	Specifies the size of a fixed-length string in a structure for use with file input and output functions. Specific to Visual Basic .NET.
VBFixedArrayAttribute Class	Specifies the size of a fixed array in a structure for use with file input and output functions. Specific to Visual Basic .NET.
WebMethodAttribute Class	Makes a method callable using the **SOAP** protocol. Used in XML Web services.
SerializableAttribute Class	Indicates that a class can be serialized.
MarshalAsAttribute Class	Determines how a parameter should be marshaled between the managed code of Visual Basic .NET and unmanaged code such as a Windows API. Used by the common language runtime.
AttributeUsageAttribute Class	Specifies how an attribute can be used.
DllImportAttribute Class	Indicates that the attributed method is implemented as an export from an unmanaged DLL.

Attributes Specific to Visual Basic .NET

Three attributes are specific to Visual Basic .NET: **COMClassAttribute**, **VBFixedStringAttribute**, and **VBFixedArray**.

COMClassAttribute

Use **COMClassAttribute** to simplify the process of creating COM components from Visual Basic .NET. COM objects are considerably different from .NET assemblies, and without **COMClassAttribute**, you need to follow a number of steps to generate a COM object from Visual Basic .NET. For classes marked with **COMClassAttribute**, the compiler performs many of these steps automatically.

VBFixedStringAttribute

Use **VBFixedStringAttribute** to force Visual Basic .NET to create a fixed-string array. Arrays are of variable length by default, and this attribute is useful when storing strings to files. The following code demonstrates this:

```
Structure Worker
   ' The runtime uses VBFixedString to determine
   ' if the field should be written out as a fixed size.
   <VBFixedString(10)> Public LastName As String
   <VBFixedString(7)> Public Title As String
   <VBFixedString(2)> Public Rank As String
End Structure
```

VBFixedArrayAttribute

Use **VBFixedArrayAttribute** to declare arrays that are fixed in size. Like Visual Basic .NET strings, arrays are of variable length by default. This attribute is useful when serializing or writing data to files.

See Also

Global Attributes | Extending Metadata Using Attributes | Applying Attributes | Retrieving Attribute Information | Retrieving Information Stored in Attributes | Writing Custom Attributes | COMClassAttribute Class | VBFixedArrayAttribute Class | VBFixedStringAttribute Class | System.Runtime.InteropServices Namespace | WebMethodAttribute Class | SerializableAttribute Class | MarshalAsAttribute Class | AttributeUsageAttribute Class | DllImportAttribute Class

Common Uses for Attributes

In using the common language runtime and class libraries, you probably will need to use attributes at some point. The following list includes a few of the common uses of attributes in code:

- Marking methods using the **WebMethod** attribute in XML Web services to indicate that the method should be callable over the SOAP protocol. For more information, see WebMethodAttribute Class.
- Describing how to marshal method parameters when interoperating with native code. For more information, see MarshalAsAttribute Class.
- Describing the COM properties for classes, methods, and interfaces.
- Marking components as COM so the Visual Basic compiler will generate the additional code required to create a COM component. For more information, see COMClassAttribute Class.
- Calling unmanaged code using the DllImportAttribute Class.
- Describing your assembly in terms of title, version, description, or trademark.
- Describing which members of a class to serialize for persistence.
- Describing how to map between class members and XML nodes for XML serialization.
- Describing the security requirements for methods.
- Specifying characteristics used to enforce security.
- Controlling optimizations by the just-in-time (JIT) compiler so the code remains easy to debug.

You can use attributes in many additional ways, and you can also create custom attributes.

See Also

Attributes and Design-Time Support | WebMethodAttribute Class | MarshalAsAttribute Class | COMClassAttribute Class | DllImportAttribute Class | Attributes Overview | Application of Attributes | Global Attributes | Custom Attributes | Defining Your Own Attributes | Attributes used in Visual Basic .NET

Application of Attributes

You apply attributes by adding attribute blocks to program elements such as properties, methods, events, classes, and assemblies. An attribute block consists of angle brackets ("< >") enclosing a list of comma-separated attribute declarations. An attribute declaration consists of an optional attribute modifier such as **Module** or **Assembly**, an attribute name, a (possibly overloaded) list of required positional parameters, and a list of optional named arguments. You must place attributes with modifiers in an attributes section at the top of a source file. For example, the following code sets an assembly attribute that defines a title for the assembly, and a module attribute that indicates that the module is compliant with the common language specification (CLS):

```
Option Explicit On
Imports System.Reflection
<Assembly: AssemblyTitleAttribute("Production assembly 4"), _
Module: CLSCompliant(True)>
Namespace MyNameSpace
```

Assembly attributes also can be applied via the AssemblyInfo.vb file, which is added to projects automatically by the Visual Studio .NET user interface. This file contains assembly-level attributes with either default or empty values.

When applied to program elements such as properties, attributes precede the element. For example, the following code applies an attribute to a class definition:

```
<CustomAttr(Update:=True)> Class Class1
```

By convention, all attribute names end with the word "Attribute" to distinguish them from other items in the .NET Framework. However, you do not need to specify the attribute suffix when using attributes. For example, if you had an attribute named `CustomAttrAttribute`, then specifying `<CustomAttr(Update:=True)>` is equivalent to specifying `<CustomAttrAttribute(Update:=True)>`.

Attribute Arguments

Attributes use optional, required, positional, and named arguments in much the same way that objects use these types of arguments. Positional arguments are arguments that are specified in the order they are declared for the constructor of an attribute. For example, the following code calls the **Sub New** constructor of an attribute with two values:

```
<CustomAttr(True, False)>
```

Arguments passed to **Sub New** in attribute classes are often used to initialize the values of fields and properties.

You can use named arguments to directly set the value of properties and fields. You specify a named argument by appending ":=" to the name of the argument, followed by the value being specified. Unlike positional arguments, named arguments can be specified in any order. For example, the following code sets the value **True** for the `Update` field, and **False** for the `Keep` field:

```
<CustomAttr(Update:=True, Keep:=False)>
```

> **Note** Attribute arguments have one important difference from arguments used with standard method calls. You must use positional arguments for arguments that are used with the **Sub New** constructor of attribute classes. Named arguments can only be used to set the values of fields and properties of attribute classes.

Required arguments are those that must always be specified. Optional arguments are those that can be skipped using a comma as a placeholder (when using positional arguments), or simply omitted (when using named arguments).

Attribute Examples

The following procedures provide examples of attribute declarations.

To use the MarshalAs attribute to control how parameters are marshaled

1. Add an **Imports** statement for the **System.Runtime.InteropServices** namespace to the top of your source code:

   ```
   Imports System.Runtime.InteropServices
   ```

2. Prefix parameters with the **MarshalAs** attribute and specify the data type the target requires. For example, the following code marshals two parameters as the data type long pointer to a string (**LPStr**) for a Windows API function:

```
Declare Auto Sub CopyFile Lib "Kernel32.Lib" ( _
   <MarshalAs(UnmanagedType.LPStr)> ByVal existingfile As String, _
   <MarshalAs(UnmanagedType.LPStr)> ByVal newfile As String, _
   ByVal failifexists As Boolean
)
```

The common language runtime uses the **MarshalAs** attribute to determine how a parameter should be marshaled between the managed code of Visual Basic .NET and the unmanaged code in a Windows API call.

To expose a method to remote Web clients

1. From the **File** menu, click **Project**, select the **ASP.NET Web Service** template, and add an **Imports** statement for the **System.Web** namespace:

```
Imports System.Web.Services
```

2. 7Define a method and use the **WebMethod** attribute to make it callable from remote Web clients:

```
<WebMethod()> Public Function HelloWorld() As String
   HelloWorld = "Hello World..."
End Function
```

Making an XML Web service method **Public** is not sufficient to expose it to Web clients. You must explicitly apply the **WebMethod** attribute to a method before it can be called by a remote Web client.

See Also

Attributes Overview | Common Uses for Attributes | Custom Attributes | Defining Your Own Attributes | WebMethodAttribute Class | Imports Statement | MarshalAsAttribute Class | System.Web Namespace | Argument Passing by Position and by Name

Global Attributes

Most attributes are attached to specific language elements such as classes or methods. However, some attributes are global — they can apply to an entire assembly or module. You specify assembly-level attributes using the following syntax:

```
<Assembly: Attribute1, Assembly: Attribute2..., Assembly: Attributen>
```

You specify module-level attributes using similar syntax:

```
<Module: Attribute1, Module: Attribute2..., Module: Attributen>
```

You place global attributes in source code after any top-level directives, such as **Option Explicit** and **Imports** statements, but before any type or namespace declarations. Global attributes can appear in multiple source files in a project, but they are generally placed in the file AssemblyInfo.vb file created automatically with Visual Basic .NET projects.

Assembly Attributes

Assembly attributes are values that provide information about an assembly. They fall into the following categories:

- Assembly identity attributes
- Informational attributes
- Assembly manifest attributes
- Strong name attributes

Assembly Identity Attributes

Three attributes (with a strong name, if applicable) determine the identity of an assembly: name, version, and culture. These attributes form the full name of the assembly and are required when referencing it in code. You can set an assembly's version and culture using attributes. The name value, however, is set by the compiler or the Assembly Linker (Al.exe) when the assembly is created, based on the file containing the assembly manifest. The **AssemblyFlags** attribute specifies whether multiple copies of the assembly can coexist.

The following table shows the identity attributes.

Attribute	Purpose
AssemblyName Class	Fully describes an assembly's identity
AssemblyVersionAttribute Class	Specifies the version of an assembly
AssemblyCultureAttribute Class	Specifies which culture the assembly supports
AssemblyFlagsAttribute Class	Specifies whether an assembly supports side-by-side execution on the same machine, in the same process, or in the same application domain

The following code applies the version and culture attributes to an assembly:

```
'Set version number for assembly.
<Assembly:AssemblyVersionAttribute("4.3.2.1")>
'Set culture as German.
<Assembly:AssemblyCultureAttribute("de")>
```

Informational Attributes

You can use informational attributes to provide additional company or product information for an assembly. The following table shows the informational attributes defined in the **System.Reflection** namespace.

Attribute	Purpose
AssemblyProductAttribute Class	Defines a product name custom attribute for an assembly manifest
AssemblyTrademarkAttribute Class	Defines a trademark custom attribute for an assembly manifest
AssemblyInformationalVersionAttribute Class	Defines an assembly informational version custom attribute for an assembly manifest

AssemblyCompanyAttribute Class	Defines a company name custom attribute for an assembly manifest
AssemblyCopyrightAttribute Class	Defines a copyright custom attribute for an assembly manifest
AssemblyFileVersionAttribute Class	Instructs the compiler to use a specific version number for the Win32 file version resource

Assembly Manifest Attributes

You can use assembly manifest attributes to provide information in the assembly manifest, including title, description, default alias, and configuration. The following table shows the assembly manifest attributes defined in the **System.Reflection** namespace.

Attribute	Purpose
AssemblyTitleAttribute Class	Defines an assembly title custom attribute for an assembly manifest
AssemblyDescriptionAttribute Class	Defines an assembly description custom attribute for an assembly manifest
AssemblyConfigurationAttribute Class	Defines an assembly configuration custom attribute (such as retail or debug) for an assembly manifest
AssemblyDefaultAliasAttribute Class	Defines a friendly default alias for an assembly manifest

Strong Name Attributes

Strong names are unique identifiers that protect the identity and integrity of an assembly. You can use strong name attributes to set a strong name for an assembly. The following table shows the strong name attributes defined in the **System.Reflection** namespace.

Attribute	Purpose
AssemblyDelaySignAttribute Class	Boolean that indicates whether you want to reserve space in the executable file for the strong name signature, but defer the actual signing until some later stage. For more information, see Delay Signing an Assembly.
AssemblyKeyFileAttribute Class	Indicates the file that contains a key. The location of the KeyFile should be relative to the project output directory, which is %Project Directory%\obj\<configuration>. For example, if your KeyFile is located in the project directory, you would specify the AssemblyKeyFile attribute as [assembly: AssemblyKeyFile("..\\..\\mykey.snk")]
AssemblyKeyNameAttribute Class	Refers to a key that has been installed in the Crypto Service Provider (CSP) on your machine. You must specify a key in order for your file to be signed.

If the KeyFile and the KeyName values are both specified, the following processing occurs:

- If the KeyName can be found in the CSP, that key is used.
- If the KeyName does not exist and the KeyFile does exist, the key in the KeyFile is installed into the CSP and used.
- For more information, see Assembly Security Considerations.

Signing Assemblies

You can sign an assembly in two different, but complementary ways: with a strong name or using the File Signing Tool (Signcode.exe). Signing an assembly with a strong name adds a public key encryption to the file containing the assembly manifest. Strong name signing ensures name uniqueness, prevents name spoofing and provides callers with some identity when a reference is resolved. For more information, see Signing an Assembly with a Strong Name.

To sign an assembly with a strong name

1. Use the Strong Name Tool (Sn.exe) to generate a key file containing a key pair.
2. Add strong name attributes to the AssemblyInfo.vb file created automatically with Visual Basic .NET projects. You can edit this file by double clicking its name in the **Solution Explorer**.

The following example uses delay signing to create a strong-named assembly with a public key file called `myKey.snk`.

```
<Assembly: AssemblyKeyFile("myKey.snk")>
<Assembly: AssemblyDelaySign(True)>
```

> **Note** Assembly-level attributes are ignored if you are not creating an assembly, such as when compiling from the command line using the **/target:module** option.

See Also

AssemblyFlagsAttribute Class | AssemblyName Class | AssemblyVersionAttribute Class | AssemblyCultureAttribute Class | AssemblyProductAttribute Class | AssemblyTrademarkAttribute Class | AssemblyInformationalVersionAttribute Class | AssemblyCompanyAttribute Class | AssemblyCopyrightAttribute Class | AssemblyFileVersionAttribute Class | AssemblyTitleAttribute Class | AssemblyDescriptionAttribute Class | AssemblyConfigurationAttribute Class | AssemblyDefaultAliasAttribute Class | AssemblyDelaySignAttribute Class | AssemblyKeyFileAttribute Class | AssemblyKeyNameAttribute Class | Custom Attributes | Defining Your Own Attributes | Application of Attributes | Strong Name Tool (Sn.exe) | Signing an Assembly with a Strong Name | File Signing Tool (Signcode.exe) | Assembly Security Considerations

Custom Attributes

Custom attributes are user-defined attributes that provide additional information about program elements. For example, you might define a custom security attribute that specifies the permissions required by the caller to execute a procedure.

You define custom attributes in attribute classes based on the **System.Attribute** class. Attribute classes themselves use an attribute called **AttributeUsage** to provide information about how the attribute can be used. Specifying **Inherited** = **True** indicates that an attribute can propagate to derived classes. Setting the **AllowMultiple** property to **True** allows you to apply more than one instance of the attribute to a program element. The **AttributeTargets** enumeration lets you define which kinds of program elements your attribute can apply to.

In the following code, the **AttributeUsage** attribute specifies an attribute that can be applied to any type of item, inherited, and applied only once:

```
<AttributeUsage(AttributeTargets.All, Inherited := True, AllowMultiple := False)>
```

You can use the **Or** operator to combine multiple items from the **AttributeTargets** enumeration, as in the following code:

```
<AttributeUsage(AttributeTargets.Class Or AttributeTargets.Method)>
```

In This Section

Defining Your Own Attributes
Explains how to create your own attributes using attribute classes.

Retrieving Custom Attributes
Demonstrates how to retrieve custom attributes using **GetCustomAttribute** or **GetCustomAttributes**.

Examples of Custom Attribute Usage
Provides example code that defines a custom attribute that can be applied only to classes, and shows how to use the new attribute.

Related Sections

Visual Basic .NET and the .NET Framework
Describes the role of Visual Basic .NET in the .NET Framework.

Object-Oriented Programming in Visual Basic
Provides information about object-oriented programming and how it is used.

Metadata and Self-Describing Components
Provides detailed information about the kinds of metadata used in Visual Studio .NET, including attributes.

Defining Your Own Attributes

Using attribute classes, you can create your own custom attributes, and use them in addition to the .NET Framework attributes to provide additional information about program elements.

To define a custom attribute

1. Declare a class and apply the **AttributeUsage** attribute to it. The name of your class is the name of the new attribute, as shown in the following code:

```
<AttributeUsage(AttributeTargets.All)> Class TestAttribute
```

2. Add an **Inherits System.Attribute** statement as the first line of code in the class:

```
Inherits System.Attribute
```

3. Define **Private** fields to store property values:

```
Private m_SomeValue As String
```

4. If appropriate, create a constructor for the attribute:

```
Public Sub New(ByVal Value As String)
   m_SomeValue = Value
End Sub
```

5. Define methods, fields, and properties for the attribute:

```
Public Sub Attr(ByVal AttrValue As String)
   'Add method code here.
End Sub
Public Property SomeValue() As String  ' A named parameter.
   Get
      Return m_SomeValue
   End Get
   Set(ByVal Value As String)
      m_SomeValue = Value
   End Set
End Property
```

6. End the class with the **End Class** construct:

```
End Class
```

See Also

Application of Attributes | Object Lifetime: How Objects Are Created and Destroyed | AttributeUsageAttribute Class

Retrieving Custom Attributes

You can retrieve custom attributes using the **GetCustomAttribute** or **GetCustomAttributes** methods of the **System.Attribute** class.

To retrieve a single instance of a custom attribute from a class

1. Add an **Imports** statement to the top of your source code to import the **Attribute** class from the **System** namespace:

```
Imports System.Attribute
```

2. Create a procedure to retrieve the attribute:

```
Sub RetrieveAttribute

End Sub
```

3. Inside the procedure, declare a variable of type **Attribute**, and another variable of the same type as the attribute you wish to retrieve:

```
Dim Attr As Attribute
Dim CustAttr As CustomAttribute
```

4. Use the **GetType** operator to pass the type of the class and attribute to a call to the **GetCustomAttribute** method, and then assign the returned value to the variable declared as **Attribute**:

```
Attr = GetCustomAttribute(GetType(ThisClass), _
                          GetType(CustomAttribute), False)
```

5. Use the **CType** function to convert the attribute's data type from generic attribute to the specific attribute of the type you retrieved. Then assign the result to the variable declared as the custom attribute type:

```
CustAttr = CType(Attr , CustomAttribute)
```

6. Check to see if the attribute was retrieved using **IsNothing**, and if it was, use the fields, properties, and methods of the attribute:

```
If CustAttr Is Nothing Then
   MsgBox("The attribute was not found.")
Else
   'Get the label and value from the custom attribute.
   MsgBox("The attribute label is: " & CustAttr.Label)
   MsgBox("The attribute value is: " & CustAttr.Value)
   End If
```

In the above example, the **GetAttr** procedure calls the **GetCustomAttribute** method of the **System.Attribute** class to get the custom attribute applied to the class `ThisClass`. **GetCustomAttribute** is a shared method, so you do not need to create an instance of **System.Attribute** first. The **CType** function converts the returned attribute from type **System.Attribute** to the custom attribute type `CustomAttribute`.

See Also

Attribute.GetCustomAttribute Method | Attribute.GetCustomAttributes Method | GetType Operator | CType Function | IsNothing Function | GetAttr Function | Defining Your Own Attributes | Application of Attributes | Retrieving Information Stored in Attributes | AttributeUsageAttribute Class

Examples of Custom Attribute Usage

The following example defines a custom attribute that can be applied only to classes:

```
<AttributeUsage(AttributeTargets.Class)> Public Class CustomAttribute
      Inherits System.Attribute

      'Declare two private fields to store the property values.
      Private m_Label As String
      Private m_Value As Integer

      'The Sub New constructor is the only way to set the properties.
      Public Sub New(ByVal _Label As String, ByVal _Value As Integer)
         m_Label = _Label
         m_Value = _Value
      End Sub

      Public Overridable ReadOnly Property Label() As String
         Get
            Return m_Label
         End Get
      End Property

      Public Overridable ReadOnly Property Value() As Integer
         Get
            Return m_Value
         End Get
      End Property
   End Class
```

Only the constructor for the attribute class can set the properties defined in this attribute. The following code shows how you could use the attribute:

```
' Apply the custom attribute to this class.
<Custom("Some metadata", 66)> Class ThisClass
   ' Add class members here.
End Class
```

See Also

Defining Your Own Attributes | Retrieving Custom Attributes | Application of Attributes | Retrieving Information Stored in Attributes | AttributeUsageAttribute Class

Declared Elements

A *declared element* is a programming element that is defined in a declaration statement. Declared elements include variables, constants, enumerations, structures, properties, methods, procedures, procedure arguments, function returns, events, delegates, interfaces, and classes. Declaration statements include **Dim**, **Const**, **Public**, **Protected**, **Friend**, **Protected Friend**, **Private**, **Static**, **Sub**, **Function**, **Property**, **Structure**, **Enumeration**, **Class**, and **Interface**.

In This Section

Declared Element Names
Describes how to name elements and use alphabetic case.
Declared Element Characteristics
Covers characteristics, such as scope, possessed by declared elements.
Declared Element Reference
Describes how the compiler matches a reference to a declaration and how to qualify a name.

Related Sections

Object-Oriented Programming in Visual Basic
Covers object-oriented programming basics.
Program Structure and Code Conventions
Presents guidelines for making your code easier to read, understand, and maintain.
Declaration Statements
Describes statements that name and define procedures, variables, arrays, and constants.

Declared Element Names

Every declared element has a name, also called an *identifier*, which is what the code uses to refer to it.

An element name:

- Must begin with an alphabetic character or an underscore (_).
- Must contain only alphabetic characters, decimal digits, and underscores.
- Must contain at least one alphabetic character or decimal digit it if begins with an underscore.
- Must not be more than 1023 characters long.

The following example shows some valid and invalid element names:

```
aB123__45   ' Valid.
_567   ' Valid.
_   ' Invalid - consists of an underscore only.
12ABC   ' Invalid - begins with a decimal digit.
xyz$wv   ' Invalid - contains an identifier type character.
```

> **Note** The length limit of 1023 characters also applies to a fully qualified name, such as: `OuterNamespace.MiddleNamespace.InnerNamespace.ThisClass.ThisElement`. The entire preceding string is limited to 1023 characters.

Escaped Names

Generally, an element name must not match any of the keywords reserved by Visual Basic, such as **Case** or **Friend**. However, you can define an *escaped name*, which is enclosed by brackets ([ ]). An escaped name can match any Visual Basic keyword, since the brackets remove any ambiguity. You also use the brackets when you refer to the name later in your code.

Case Sensitivity in Names

Names in Visual Basic .NET are case insensitive. This means that when the compiler compares two names that differ in alphabetic case only, it interprets them as the same name. For example, `ABC` and `abc` are considered to refer to the same declared element.

However, the common language runtime uses case-sensitive binding. Therefore, when you produce an assembly or a DLL and make it available to other assemblies, your names are no longer case insensitive. For example, if you define a class with an element called `ABC`, and other assemblies make use of your class through the common language runtime, they must refer to the element as `ABC`. If you subsequently recompile your class and change the element's name to `abc`, the other assemblies using your class can no longer access that element. Therefore, when you release an updated version of an assembly, you should not change the alphabetic case of any public elements.

Comparison of names is also independent of locale; that is, if two names match in one locale, they are guaranteed to match in all locales.

See Also

Declared Element Characteristics | Declared Element Reference | Declaration Statements

Declared Element Characteristics

A *characteristic* of a declared element is an aspect of that element that affects how code can interact with it. Every declared element has one or more of four characteristics — data type, lifetime, scope, and accessibility — associated with it.

In This Section

Characteristics of Visual Basic Declared Elements
Presents a list of all Visual Basic declared elements and the characteristics associated with each.
Lifetime
Explains how lifetimes are determined, how they begin and end, and how they can be extended.
Scope
Describes the different levels of scope a programming element can have.
Accessibility
Describes the different levels of accessibility a programming element can have.

Related Sections

Data Types
Describes the kinds of data that programming elements can hold and how that data is stored.
Variable Declaration
Explains how to specify the name and characteristics of a variable.

Characteristics of Visual Basic Declared Elements

Every *declared element* (a programming element that is defined in a declaration statement) has one or more of the following characteristics associated with it:

- Data type — the values the element can hold, and how those values are stored
- Lifetime — the period of execution time during which the element is available for use
- Scope — the set of all code that can refer to the element without qualifying its name
- Accessibility — the permission for code to make use of the element

The following table shows the declared elements and the characteristics that apply to each one.

Element	**Data Type**	**Lifetime**	**Scope**	**Accessibility**
Variable	Yes	Yes	Yes	Yes
Constant	Yes	No	Yes	Yes
Enumeration	Yes	No	Yes	Yes
Structure	No	No	Yes	Yes
Property	Yes	No	Yes	Yes
Method	No	No	Yes	Yes
Procedure	No	No	Yes	Yes
Procedure argument	Yes	Yes	Yes	No
Function return	Yes	Yes	Yes	No
Event	No	No	Yes	Yes
Delegate	No	No	Yes	Yes
Interface	No	No	Yes	Yes
Class	No	No	Yes	Yes

Visibility

The concept of visibility is often mentioned. Visibility results from a combination of lifetime, scope, and accessibility. A programming element can be said to be *visible* from a block of code when all three of the following conditions apply:

- The element is within its lifetime — that is, it has been created and not yet terminated.
- The scope of the element includes the block of code referring to it.
- The element's accessibility permits the code to use it.

See Also

Lifetime | Scope | Accessibility

Lifetime

The *lifetime* of a declared element is the period of time during which it is available for use. Variables are the only elements that have lifetime; for this purpose, the compiler treats procedure arguments and function returns as special cases of variables. The lifetime of a variable represents the period of time during which it can hold a value. Its value can change over its lifetime, but it always holds some value.

Different Lifetimes

A variable declared at module level typically exists for the entire time your application is running. A nonshared variable declared in a class or structure exists as a separate copy for each instance of the class or structure in which it is declared; each such variable has the same lifetime as its instance. However, a **Shared** variable has only a single lifetime, which lasts for the entire time your application is running.

Local variables declared with **Dim** exist only while the procedure in which they are declared is executing. This applies also to that procedure's arguments and to any function return. However, if that procedure calls other procedures, the local variables retain their values while the called procedures are running.

Beginning of Lifetime

A local variable's lifetime begins when execution enters the procedure in which it is declared. Every local variable is initialized to the default value for its data type as soon as the procedure begins execution. Numeric variables (including **Byte** and **Char**) are initialized to zero, **Date** variables to midnight of January 1 of the year 1, **Boolean** variables to **False**, and reference type variables (including strings, arrays, and **Object**) to **Nothing**.

Each member of a structure variable is initialized as if it were a separate variable. Similarly, each element of an array variable is initialized individually.

If a variable is declared with an initializer, the specified value is assigned to the variable when its declaration statement is executed, as shown in the following example:

```
Dim X As Double = 18.973   ' X had previously been initialized to 0.
```

Variables declared within a block inside a procedure are initialized to their default values on entry to the procedure. These initializations take effect whether or not the block is ever executed.

End of Lifetime

When a procedure terminates, the values of its local variables are not preserved, and the memory used by the local elements is reclaimed. The next time the procedure is executed, all its local elements are created afresh and the local variables are initialized.

When an instance of a class or structure terminates, its nonshared variables lose their values. Each new instance of the class or structure creates all its nonshared elements and initializes the nonshared variables. **Shared** elements are preserved until your application stops running.

Extension of Lifetime

If a local variable is declared with the **Static** keyword, its lifetime is longer than the execution time of the procedure in which it is declared. If the procedure is inside a module, the static variable survives as long as your application continues running.

If a static variable is declared in a procedure inside a class, the variable's lifetime depends on whether the procedure is shared. If the procedure is declared with the **Shared** keyword, the variable's lifetime continues until your application terminates. If the procedure is nonshared, its static variables are instance members of the class, and their lifetimes are the same as that of the class instance.

In the following example, the `RunningTotal` function calculates a running total by adding a new value to the total of previous values stored in the static variable `ApplesSold`:

```
Function RunningTotal(ByVal Num As Integer) As Integer
   Static ApplesSold As Integer
   ApplesSold = ApplesSold + Num
   Return ApplesSold  ' ApplesSold keeps its current value.
End Function
```

If `ApplesSold` had been declared without using **Static**, the previous accumulated values would not be preserved across calls to the function, and the function would simply return the same value with which it was called.

You can produce the same lifetime by declaring `ApplesSold` at module level. If you change the scope of a variable this way, however, the procedure no longer has exclusive access to it. Because other procedures can access the variable and change its value, the running total would be unreliable and the code could be more difficult to maintain.

See Also

Static Variable Initialization | Declared Element Names | Declared Element Reference | Characteristics of Visual Basic Declared Elements | Scope | Accessibility | Variables | Variable Declaration | Dim Statement | Shared | Static | Nothing

Static Variable Initialization

You can initialize the value of a static local variable as part of its declaration. If you declare an array to be **Static**, you can initialize its rank (number of dimensions), the length of each dimension, and the values of the individual elements.

You can declare static variables with the same name in more than one procedure. If you do this, the Visual Basic compiler considers each such variable to be a separate element. The initialization of one of these variables does not affect the values of the others. The same applies if you define a procedure with a set of overloads and declare a static variable with the same name in each overload.

You can declare a static local variable in a class, that is, inside a procedure in that class. However, you cannot declare a static local variable in a structure.

See Also

Characteristics of Visual Basic Declared Elements | Declared Element Characteristics | Lifetime | Scope | Accessibility | Variables | Variable Declaration | Procedure Overloading | Static

Scope

The *scope* of a declared element is the set of all code that can refer to it without qualifying its name or making it available through an **Imports** statement. An element can have scope at one of the following levels:

- Block scope — available only within the code block in which it is declared
- Procedure scope — available only within the procedure in which it is declared
- Module scope — available to all code within the module, class, or structure in which it is declared
- Namespace scope — available to all code in the namespace

These levels of scope progress from the narrowest (block) to the widest (namespace), where *narrowest scope* means the smallest set of code that can refer to the element without qualification.

You specify the scope of an element when you declare it. The scope can depend on the following factors:

- The region (block, procedure, module, class, or structure) in which you declare the element
- The namespace containing the element's declaration
- The accessibility you declare for the element

Use care when you define variables with the same name but different scope, because doing so can lead to unexpected results. For more information, see Reference Matching.

See Also

Levels of Scope | Imports Statement | Reference Matching

Levels of Scope

A programming element is available throughout the region in which you declare it. All code in the same region can refer to the element without qualifying its name.

Block Scope

A block is a set of statements terminated by an **End**, **Else**, **Loop**, or **Next** statement, for example within a **For...Next** or **If...Then...Else...End If** construction. An element declared within a block can be used only within that block. In the following example, the scope of the integer variable `Cube` is the block between **If** and **End If**, and `Cube` can no longer be referenced when execution passes out of the block:

```
If N < 1291 Then
   Dim Cube As Integer
   Cube = N ^ 3
End If
```

> **Note** Even if the scope of an element is limited to a block, its lifetime is still that of the entire procedure. If you enter the block more than once during the procedure, a block variable retains its previous value. To avoid unexpected results in such a case, it is wise to initialize the variable each time.

Procedure Scope

An element declared within a procedure is not available outside that procedure. Only the procedure that contains the declaration can use it. Elements at this level are also known as *local* elements. You declare them with the **Dim** statement, with or without the **Static** keyword.

Procedure and block scope are closely related. If an element is declared inside a procedure but outside any block within that procedure, the element can be thought of as having block scope, where the block is the entire procedure.

> **Note** All local elements, even if they are **Static** variables, are private to the procedure in which they appear. You cannot declare an element using the **Public** keyword within a procedure.

Module Scope

For convenience, the single term *module level* applies equally to modules, classes, and structures. You can declare elements at this level by placing the declaration statement outside of any procedure or block within the module, class, or structure.

When you make a declaration at the module level, the accessibility you choose determines the scope. The namespace that contains the module, class, or structure also affects the scope.

Elements for which you declare **Private** accessibility are available for reference to every procedure in that module, but not to any code in a different module. The **Dim** statement at module level defaults to **Private** if you do not use any accessibility keywords. However, you can make the scope and accessibility more obvious by using the **Private** keyword in the **Dim** statement.

In the following example, the string variable `strMsg` is available to all procedures defined in the module. When the second procedure is called, it displays the contents of the string variable `strMsg` in a dialog box:

```
' Put the following declaration at module level (not in any procedure).
Private strMsg As String
' ...
Sub InitializePrivateVariable()
   strMsg = "This variable cannot be used outside this module."
End Sub
' ...
Sub UsePrivateVariable()
   MsgBox(strMsg)
End Sub
```

Namespace Scope

If you declare an element at module level using the **Friend** or **Public** keyword, it becomes available to all procedures throughout the namespace in which the element is declared. With the following alteration to the preceding example, the string variable

`strMsg` can be used by code anywhere in the namespace of its declaration:

```
' Include this declaration at module level (not inside any procedure).
Public strMsg As String   ' strMsg is available throughout the namespace.
```

Namespace scope includes nested namespaces. An element available from within a namespace is also available from within any namespace nested inside that namespace.

If your project does not contain any **Namespace** statements, everything in the project is in the same namespace. In this case, namespace scope can be thought of as project scope. **Public** elements in a module, class, or structure are also available to any project that references their project.

See Also

Declared Element Names | Declared Element Reference | Characteristics of Visual Basic Declared Elements | Scope | Lifetime | Accessibility | Variables | Variable Declaration | Choice of Variable Scope | Shadowing | Dim Statement | Public | Protected | Friend | Private | Static

Accessibility

The *accessibility* of a declared element is the ability to use it, that is, the permission for code to read it or write to it. This is determined not only by how you declare the element itself, but also by the accessibility of the element's container. If the containing element is not accessible, none of its contained elements are accessible, even those declared as **Public**. For example, a **Public** variable in a **Private** structure is accessible from inside the class containing the structure, but not from outside that class.

Public

The **Public** keyword in the **Dim** statement declares elements to be accessible from anywhere within the same project, from other projects that reference the project, and from an assembly built from the project. The following code shows a sample **Public** declaration:

```
Public Class ClassForEverybody
```

You can use **Public** only at module, namespace, or file level. This means you can declare a public element in a source file or inside a module, class, or structure, but not within a procedure.

Protected

The **Protected** keyword in the **Dim** statement declares elements to be accessible only from within the same class, or from a class derived from this class. The following code shows a sample **Protected** declaration:

```
Protected Class ClassForMyHeirs
```

You can use **Protected** only at class level, and only when declaring a member of a class.

Friend

The **Friend** keyword in the **Dim** statement declares elements to be accessible from within the same project, but not from outside the project. The following code shows a sample **Friend** declaration:

```
Friend StringForThisProject As String
```

You can use **Friend** only at module, namespace, or file level. This means you can declare a friend element in a source file or inside a module, class, or structure, but not within a procedure.

Protected Friend

The **Protected** and **Friend** keywords together in the **Dim** statement declare elements to be accessible either from derived classes or from within the same project, or both. The following code shows a sample **Protected Friend** declaration:

```
Protected Friend StringForProjectAndHeirs As String
```

You can use **Protected Friend** only at class level, and only when declaring a member of a class.

Private

The **Private** keyword in the **Dim** statement declares elements to be accessible only from within the same module, class, or structure. The following code shows a sample **Private** declaration:

```
Private NumberForMeOnly As Integer
```

You can use **Private** only at module, namespace, or file level. This means you can declare a private element in a source file or inside a module, class, or structure, but not within a procedure.

> **Note** At the module level, the **Dim** statement without any accessibility keywords is equivalent to a **Private** declaration. However, you might want to use the **Private** keyword to make your code easier to read and interpret.

See Also

Declared Element Names | Declared Element Reference | Declared Element Characteristics | Lifetime | Scope | Variables | Variable Declaration | Dim Statement | Public | Protected | Friend | Private | Static

Declared Element Reference

When your code makes reference to a declared element, the Visual Basic compiler matches the name in your reference to the appropriate declaration of that name. If more than one element is declared with the same name, you can control which of those elements is referenced by *qualifying* its name.

Qualifying a name means preceding it in your source statement with information that resolves any possible ambiguity. Qualifying information can include one or more namespaces, a class, a structure, or a module. In the following example, the **Sub** procedure `SomeSub` is declared in more than one place, and the **Call** statement specifies that the procedure that should be called is the one defined in the module `SomeMod` in the namespace `SomeSpace`:

```
Call SomeSpace.SomeMod.SomeSub()   ' SomeSub is qualified.
```

In This Section

Reference Matching
Discusses how the compiler matches a reference to a declaration.
Shadowing
Describes the treatment of elements with the same name.
Names in Forms and Controls
Discusses considerations for naming elements within forms and controls.
Name Ambiguity
Discusses ambiguity when programming elements have the same name.

Related Sections

Data Types
Describes the kinds of data that programming elements can hold and how that data is stored.
Variables
Discusses how to declare variables and use them to refer to objects.
Procedures
Describes **Sub**, **Function,** and **Property** procedures, as well as advanced topics such as recursive and overloaded procedures.

Reference Matching

When the compiler matches a name reference to a name declaration, it uses the matching declaration with the narrowest scope. In the following example, two **Sub** procedures, both called `Perform`, are declared **Public** in different modules in a project. Each one can be specified without qualification within its own module but must be qualified if referenced from anywhere else.

```
' Assume these three modules are all in the same project.
Module Mod1
   Public Sub Perform()  ' Public keyword allows access outside of Mod1.
      ' ...
      MsgBox("Mod1.Perform() now returning")
   End Sub
End Module

Module Mod2
   Public Sub Perform()   ' Might be different from Perform() in Mod1.
      ' ...
      MsgBox("Mod2.Perform() now returning")
   End Sub
   Public Sub DoSomething()
      Perform()   ' Calls Perform in Mod2, the active module.
      Mod1.Perform()   ' Calls Perform in Mod1.
   End Sub
End Module

Module Mod3
   Public Sub CallPerform()
      Perform()   ' This call is ambiguous and generates an error.
      Mod1.Perform()   ' Calls Perform in Mod1.
   End Sub
End Module
```

Members of Other Classes and Structures

When you use a nonshared member of another class or structure, you must first qualify the member name with a variable or expression that points to an instance of the class or structure. In the following example, `DemoClass` is an instance of a class named `Class1`:

```
Dim DemoClass As Class1 = New Class1()
DemoClass.SomeSub[(argumentlist)]
```

You cannot use the class name itself to qualify a member that is not **Shared**. The instance must first be declared as an object variable (in this case `DemoClass`) and then referenced by the variable name.

You can qualify a **Shared** member of a class either with the class name or with a variable or expression that points to an instance of the class.

References

To use **Public** elements defined in another project, you must first set a *reference* to that project's assembly or type library. You can set a reference by choosing **Add Reference** from the **Project** menu, or with the **/reference** command-line compiler option.

For example, you can use the XML object model of .NET. If you set a reference to the **System.Xml** namespace, you can declare and use any of its classes, such as **XmlDocument**. The following declaration provides an example:

```
Dim XDoc As System.Xml.XmlDocument   ' XDoc is an XML document object.
```

Imports

You can use the **Imports** statement to *import* the namespaces containing the modules or classes you want to use. This allows you to refer to the elements defined in an imported namespace without qualifying their names. In the preceding example, you could import the namespace containing the **XmlDocument** class, as follows:

```
Imports System.Xml   ' Make System.Xml available without qualification.
' The Imports statement must precede all your declarations.
Dim XDoc As XmlDocument   ' XDoc is an XML document object.
```

In addition, the **Imports** statement can define an *import alias* for each imported namespace. This can make your source code shorter and easier to read. In the preceding example, you could import with an alias, as follows:

```
Imports XD = System.Xml   ' XD is alias for System.Xml namespace.
' The Imports statement must precede all your declarations.
Dim XDoc As XD.XmlDocument   ' XDoc is an XML document object.
```

The **Imports** statement does not make elements from other projects available to your application; that is, it does not take the place of setting a reference. Importing a namespace simply removes the need to qualify the names defined in that namespace.

You can also use the **Imports** statement to import modules, classes, structures, and enumerations. You can then use the members of such imported elements without qualification. However, nonshared members of classes and structures must always be qualified with a variable or expression that evaluates to an instance of the class or structure.

Classes with the Same Name

When you create a new instance of an object, you might have to qualify the class with the namespace or type library to which it belongs. For example, the **System.Windows.Forms** and **System.Web.UI.WebControls** namespaces both contain a **Label** class. If your application uses both, or if it defines its own **Label** class, you must distinguish the different **Label** objects. Include the namespace or import alias in the variable declaration, as in the following example:

```
Imports WIN = System.Windows.Forms, WEB = System.Web.UI.WebControls
' The Imports statement must precede all your declarations.
Dim WinLabel As New WIN.Label()   ' Uses the Windows.Forms.Label class.
```

See Also

Declared Element Reference | Variables | System Namespace | System.Xml Namespace | System.Windows.Forms Namespace | System.Web.UI.WebControls Namespace | Object Class | XmlDocument Class | /reference | Setting Visual Basic Project Properties | Imports Statement | New | Public | Shared

Shadowing

When two programming elements share the same name, one of them can hide, or *shadow*, the other one. In such a situation, the shadowed element is not available for reference; instead, when your code uses the shared name, the Visual Basic compiler resolves it to the shadowing element.

An element can shadow another element in two different ways. The shadowing element can be declared inside a subregion of the region containing the shadowed element, in which case the shadowing is accomplished through scope. Or a deriving class can redefine a member of a base class, in which case the shadowing is done through inheritance.

Shadowing Through Scope

It is possible for programming elements in the same module, class, or structure to have the same name but different scope. When two elements are declared in this manner and the code refers to the name they share, the element with the narrower scope shadows the other element (block scope is the narrowest).

For example, a module can define a **Public** variable named `Temp`, and a procedure within the module can declare a local variable also named `Temp`. References to `Temp` within the procedure access the local variable, while references to `Temp` outside the procedure access the **Public** variable. In this case, the procedure variable `Temp` shadows the module variable `Temp`.

When it is shadowed by a local variable, a module variable is still accessible from within a procedure by qualifying the variable with the name of the module, class, or structure it is declared in, as illustrated in the following example:

```
Module Mod4
   Public Num As Integer   ' This Num is a module-level variable.

   Sub Show()
     Dim Num As Integer   ' This Num is a local variable.
     Num = 2   ' Local Num has a value of 2.
     MsgBox(CStr(Mod4.Num))   ' Mod4.Num has a value of 1.
   End Sub

   Sub UseModuleLevelNum()
      Num = 1   ' Set module-level Mod4.Num to 1.
      ' ...
      Show()   ' Call Show, which displays module-level Num.
   End Sub
End Module
```

Shadowing Through Inheritance

If a derived class redefines a programming element inherited from a base class, the redefining element shadows the original element. You can shadow any type of declared element, or set of overloaded elements, with any other type. For example, an **Integer** variable can shadow a **Function** procedure. If you shadow a procedure with another procedure, you can use a different argument list and a different return type.

> **Note** If the shadowing element is not accessible from the consuming code, for example if it is declared **Private**, shadowing is defeated and the compiler resolves any reference to the same element it would have if there had been no shadowing. This is the accessible element the fewest derivational steps backward from the shadowing class. If the shadowed element is a procedure, the resolution is to the proximate accessible version with the same name, argument list, and return type.

Do not confuse shadowing with overriding. Both are used when a derived class inherits from a base class, and both redefine one declared element with another. But there are differences between the two. The following table compares shadowing with overriding:

Criterion	Shadowing	Overriding
Purpose	Protecting against a subsequent base class modification introducing a member you hav e already defined in your derived class	Achieving polymorphism by defining a different im plementation of a procedure or property with the s ame calling sequence
Redefined element	Any declared element type	Only a procedure (**Function** or **Sub**) or property
Redefining element	Any declared element type	Only a procedure or property with the identical call ing sequence[1]

Accessibility	Any accessibility	Cannot expand the accessibility of overridden element (for example, cannot override **Protected** with **Public**)
Readability and writability	Any combination	Cannot change readability or writability of overridden property
Keyword usage	**Shadows** recommended in derived class; **Shadows** assumed if neither **Shadows** nor **Overrides** specified	**Overridable** required in base class; **Overrides** required in derived class
Inheritance of redefining element by classes deriving from your derived class	Shadowing element inherited by further derived classes; shadowed element still hidden[2]	Overriding element inherited by further derived classes; overridden element still overridden

[1] The calling sequence consists of the element type (**Function**, **Sub**, or **Property**), name, argument list, and return type. You cannot override a procedure with a property, or the other way around. You cannot override a **Function** procedure with a **Sub** procedure, or the other way around.

[2] If the shadowing element is inaccessible in a further derived class, shadowing is not inherited. For example, if you declare the shadowing element as **Private**, a class deriving from your derived class inherits the original element instead of the shadowing element.

When you access an element from a derived class, you normally do so through the current instance of that derived class, by qualifying the element name with the **Me** keyword. If your derived class shadows the element in the base class, you can access the base class element by qualifying it with the **MyBase** keyword.

How you access a derived class can affect whether your code uses a shadowing element or the element it shadows. In the following example, the variable `BasObj` is declared as the base class. Assigning a `DervCls()` object to it constitutes a widening conversion and is therefore valid. However, the base class cannot access the shadowing version of the variable `Z` in the derived class, so the compiler resolves `BasObj.Z` to the original base class value.

```
Public Class BaseCls
   Public Z As Integer = 100   ' The element to be shadowed.
End Class
Public Class DervCls
   Inherits BaseCls
   Public Shadows Z As String = "*"   ' The shadowing element.
End Class
Public Class UseClasses
   Dim BObj As BaseCls = New DervCls()   ' DervCls widens to BaseCls.
   Dim DObj As DervCls = New DervCls()   ' Access through derived class.
   Public Sub ShowZ()
      MsgBox("Accessed through base class: " & BObj.Z)   ' Outputs 100.
      MsgBox("Accessed through derived class: " & DObj.Z)   ' Outputs "*".
   End Sub
End Class
```

Note If you shadow the same base class element with more than one element in your derived class, the shadowing elements become overloaded versions.

See Also

Declared Element Reference | Choice of Variable Scope | Shadows | Overrides | Widening and Narrowing Conversions | Me | MyBase

Names in Forms and Controls

In a form, all properties, controls, variables, constants, and procedures are members at the module level. Because all form members are in the same scope, no two of them are allowed to have the same name.

For each control, the forms designer automatically generates a variable at module level with the name of the control. Therefore, you cannot declare another variable at module level with the same name.

Within a form module, local variables with the same names as controls on the form shadow those controls. You must qualify the control with a reference to the form or with the **Me** keyword to access any of its properties.

For information on defining two programming elements with the same name, see Name Ambiguity.

See Also

Name Ambiguity | Declared Element Reference | Choice of Variable Scope | Me | Shadowing

Name Ambiguity

When you define two or more programming elements with the same name, a *name ambiguity* can result when the compiler attempts to resolve a reference to that name. For example, if you define two modules, each containing a **Public Function** procedure called `Recalculate`, and call `Recalculate` from a third module without qualifying its name, the compiler generates an error message because it cannot resolve the reference.

You can avoid name ambiguity by giving all your elements unique names. Then you can call any procedure without needing to qualify its name with a namespace or module.

You can resolve name ambiguity by qualifying each name with the appropriate module name and, if necessary, a namespace name. The following example calls the **Sub** procedure `YourSub` and passes `MyVar` as an argument:

```
YourProject.YourModule.YourSub(MyProject.MyModule.MyVar)
```

You can use as many levels of qualifiers as necessary to differentiate identical names.

For more information on names, see Names in Forms and Controls.

See Also

Names in Forms and Controls | Declared Element Reference | Reference Matching

Data Types

The *data type* of a programming element refers to what kind of data it can hold and how that data is stored. Data types apply to all values that can be stored in computer memory or participate in the evaluation of an expression. Every variable, literal, constant, property, procedure argument, and procedure return value has a data type.

A *declared data type* is specified as part of the declaration of a programming element. Unless you use typeless programming, you must declare data types for all your programming elements.

This section describes the declaration and use of Visual Basic .NET data types.

In This Section

Data Type Declaration
Covers declaring data types of various programming elements.
Elementary Data Types
Discusses numeric, character, and miscellaneous types provided by Visual Basic.
Composite Data Types
Summarizes structures, arrays, and classes as data types.
Data Type Implementation
Illustrates how Visual Basic .NET classifies and implements various data types.
Type Conversions
Explains changing between data types, classifications of conversions, special considerations.
Structures: Your Own Data Types
Discusses structures, how to declare them, and how to use them.
Typeless Programming
Provides an overview of programming without declaring data types, type semantics.

Related Sections

Data Type Summary
Lists the elementary data types supplied by Visual Basic .NET.

Data Type Declaration

Every programming element that can hold data has a declared data type. The following list shows how to specify data types of various elements:

- Variable — In a declaration statement; see Data Types in Declaration Statements
- Literal — With a literal type character; see Type Characters
- Constant — In a declaration statement; see Data Types in Declaration Statements
- Property — In a procedure declaration statement; see Property Procedures
- Procedure argument — In the procedure declaration; see Argument Data Type Declaration
- Procedure return value — In the procedure declaration; see Function Procedures

In This Section

Data Types in Declaration Statements
Describes declaring the data types or variables and constants.
Type Characters
Discusses special characters that force the data type of some programming elements.

Related Sections

Data Type Summary
Covers elementary data types supplied by Visual Basic .NET.

Data Types in Declaration Statements

You define a variable or constant with a declaration statement, and you specify its data type with the **As** clause. The declaration statements are **Dim** and **Const**, and the keywords they can use are **Public**, **Protected**, **Friend**, **Private**, **Shared**, and **Static**. Note that **Protected** can be used only on elements being declared within a class.

The following statements demonstrate ways to declare variables:

```
Private I As Integer
Dim Amount As Double
Static YourName As String
Public BillsPaid As Decimal
```

To declare a constant, use **Const** in the declaration, as in the following example:

```
Const Modulus As Single = 4.17825  ' Value cannot be changed.
```

You can declare several variables in a single declaration statement, provided they all have the same accessibility. The following statements demonstrate multiple variable declaration:

```
Public I, J, K As Integer   ' Declare three integer variables.
Dim FirstName, MiddleName, LastName As String   ' Three String variables.
Private S, SArray() As Short   ' One Short variable and one Short array.
Friend B1, B2 As Byte, C1, C2 As Char   ' Two Byte and two Char variables.
```

See Also

Data Types as Classes and Structures | Type Conversions | Efficient Use of Data Types | Dim Statement | Public | Protected | Friend | Private | Shared | Static | Const Statement

Type Characters

In addition to specifying a data type in a declaration statement, you can force the data type of some programming elements with a *type character*. The type character must immediately follow the element name, with no intervening characters of any kind.

The type character is not part of the name of the element. An element defined with a type character can be referenced without the type character.

Identifier Type Characters

Visual Basic supplies a set of *identifier type characters*, which you can use in a declaration to specify the data type of a variable or constant. The following table shows the available identifier type characters with examples of usage.

Identifier type character	Data type	Example
%	**Integer**	`Dim L%`
&	**Long**	`Public M&`
@	**Decimal**	`Const W@ = 37.5`
!	**Single**	`Dim Q!`
#	**Double**	`Static X#`
$	**String**	`Private V$ = "Secret"`

No identifier type characters exist for the **Boolean**, **Byte**, **Char**, **Date**, **Object**, or **Short** data types, or for any composite data types.

Identifier type characters also can force the data type of other identifiers, and you can even use them in expressions. In some cases, you can append the **$** character to a Visual Basic function, for example **Left$** instead of **Left**.

In all cases, the identifier type character must immediately follow the identifier.

Literal Type Characters

A *literal* is a textual representation of a particular value of a type. The form of a literal as it appears in your code usually determines its data type. The compiler treats integral literals as **Integer** unless they are large enough to be **Long**, and it treats nonintegral literals as **Double**. In the following statements, the data type of the literal is not ambiguous:

```
J = 347    ' This literal is of the Integer data type.
X = 9.2    ' This literal is of the Double data type.
B = False  ' This literal is of the Boolean data type.
```

Visual Basic supplies a set of *literal type characters*, which you can use to force a literal to assume a data type other than the one its form indicates. You do this by appending the character to the end of the literal. The following table shows the available literal type characters with examples of usage.

Literal type character	Data type	Example
S	**Short**	`I = 347S`

I	**Integer**	`J = 347I`
L	**Long**	`K = 347L`
D	**Decimal**	`H = 347D`
F	**Single**	`Y = 347F`
R	**Double**	`Z = 347R`
C	**Char**	`Q = "."C`

No literal type characters exist for the **Boolean**, **Byte**, **Date**, **Object**, or **String** data types, or for any composite data types.

Literals can also use the identifier type characters (**%**, **&**, **@**, **!**, **#**, **$**), as can variables, constants, and expressions. However, the literal type characters (**S**, **I**, **L**, **D**, **F**, **R**, **C**) can be used only with literals.

In all cases, the literal type character must immediately follow the literal.

See Also

Constant and Literal Data Types

Elementary Data Types

Visual Basic .NET supplies a set of pre-defined data types that you can use for many of your programming elements. This section describes these types and how to use them.

In This Section

Numeric Data Types
Describes the integral and nonintegral numeric types.
Character Data Types
Describes the **Char** and **String** types.
Miscellaneous Data Types
Describes the **Boolean**, **Date**, and **Object** types.

Related Sections

Data Types
Introduces the Visual Basic .NET data types and describes how to use them.
Data Type Summary
Provides an overview of the elementary data types supplied by Visual Basic .NET.

Numeric Data Types

Visual Basic supplies several numeric data types for handling numbers in various representations. Integral types represent only whole numbers, and nonintegral types represent numbers with both integer and fractional parts.

In This Section

Integral Types
Discusses using **Short**, **Integer**, **Long**, and **Byte** to represent integers.
Nonintegral Types
Discusses using **Decimal**, **Single**, and **Double** to represent numbers with both integer and fractional parts.

Related Sections

Data Types
Introduces the Visual Basic .NET data types and describes how to use them.
Data Type Implementation
Describes how Visual Basic .NET classifies and implements various data types.
Type Conversions
Explains changing between data types, classifications of conversions, special considerations.

Integral Types

Integral data types are those that represent only whole numbers. The signed integral data types are **Short** (16-bit), **Integer** (32-bit), and **Long** (64-bit). If a variable always stores whole numbers rather than numbers with a fractional amount, declare it as one of these types.

The unsigned integral type is **Byte** (8-bit). If a variable contains binary data, or data of unknown nature, declare as this type. If more than eight bits of data are needed, declare the variable as an array of **Byte** elements.

Arithmetic operations are faster with integral types than with other data types. They are fastest with the **Integer** type in Visual Basic .NET. Integral types are especially useful as counter variables in **For ... Next** loops.

Since **Byte** is an unsigned type with a range of 0-255, it cannot represent a negative number. If you use the unary minus (**-**) operator on an expression that evaluates to type **Byte**, Visual Basic converts the expression to **Integer** first.

If you try to set a variable of an integral type to a number outside the range for that type, an error occurs. If you try to set it to a fraction, the number is rounded, as in the following example:

```
Dim K As Integer  ' Valid range is -2147483648 to +2147483647.
' ...
K = 2147483648    ' Causes an error.
K = CInt(5.9)     ' Sets K to 6.
```

Binary data stored in **Byte** variables and arrays is preserved during format conversions. You should not use a **String** variable for binary data, because its contents can be corrupted during conversion between ANSI and Unicode formats. Such conversion can happen automatically when Visual Basic reads or writes files, or when it calls DLLs, methods, and properties.

For integral types, the literal type characters are **S** for **Short**, **I** for **Integer**, and **L** for **Long**. No literal type character exists for **Byte**.

See Also

Arrays | Introduction to International Applications in Visual Basic and Visual C# | Integer Data Type | Long Data Type | Short Data Type | Byte Data Type | Nonintegral Types

Nonintegral Types

Nonintegral data types are those that represent numbers with both integer and fractional parts. The nonintegral numeric data types are **Decimal** (128-bit fixed point), **Single** (32-bit floating point), and **Double** (64-bit floating point). They are all signed types. If a variable can contain a fraction, declare as one of these types.

The **Decimal** data type supports up to 29 significant digits and can represent values up to 7.9228×10^{28}. It is particularly suitable for calculations, such as financial, that require a large number of digits but cannot tolerate rounding errors.

To assign a large value to a **Decimal** variable or constant, you must append the literal type character to a numeric literal if it is too large for the **Long** data type, as the following example shows:

```
Dim DecVar As Decimal
DecVar = 9223372036854775808   ' Overflow; too big for Long data type.
DecVar = 9223372036854775808D  ' No overflow; Decimal data type.
```

Floating point (**Single** and **Double**) numbers have larger ranges than **Decimal** numbers but can be subject to rounding errors. Floating point types support fewer significant digits than **Decimal** but can represent values of greater magnitude.

Floating point values can be expressed as *mmm***E***eee*, in which *mmm* is the mantissa (the significant digits) and *eee* is the exponent (a power of 10). The highest positive value of a **Single** data type is 3.4028235E+38, and the highest positive value of a **Double** data type is 1.79769313486231570E+308.

For nonintegral types, the literal type characters are **D** for **Decimal**, **F** for **Single**, and **R** for **Double**.

See Also

Single Data Type | Double Data Type | Decimal Data Type | Integral Types

Character Data Types

Visual Basic provides *character data types* for dealing with printable and displayable characters. While they both deal with Unicode characters, **Char** holds a single character while **String** contains any number of characters.

Char Type

The **Char** data type is a single two-byte (16-bit) Unicode character. You can use methods like **IsDigit** and **IsPunctuation** on a **Char** variable to determine its Unicode classification.

Visual Basic does not convert directly between **Char** and the numeric types. You can achieve a type conversion with functions such as **AscW** and **ChrW**.

If the type checking switch (**Option Strict**) is on, you must append the literal type character to a single-character string literal to identify it as the **Char** data type, as the following example shows:

```
Option Strict On
 ' ...
Dim CharVar As Char
CharVar = "@"    ' Cannot convert String to Char with Option Strict On.
CharVar = "@"C   ' Successfully assigns single character to CharVar.
```

Keep in mind that the **Char** type is unsigned.

The literal type character for **Char** is **C**.

String Type

The **String** data type is a sequence of zero or more two-byte (16-bit) Unicode characters. If a variable always contains a character string and never a numeric value, declare it to be of type **String**, as in the following example:

```
Private S As String
```

You can then assign strings to this variable and manipulate it using string functions, as in the following example:

```
S = "Database"
S = Microsoft.VisualBasic.Left(S, 4)   ' Set S to "Data".
```

The **String** data type holds Unicode characters, each of which can have a value between 0 and 65,535. A **String** variable or argument is always a *variable-length string*, which grows or shrinks as you assign new data to it.

Strings are sometimes padded with leading or trailing spaces. The **Trim**, **LTrim**, and **RTrim** functions remove these spaces. You might find these functions useful when working with strings.

No literal type character exists for **String**. However, the compiler treats literals enclosed within double quotes (**" "**) as **String**.

See Also

Char Data Type | LTrim, RTrim, and Trim Functions | String Data Type

Miscellaneous Data Types

Visual Basic supplies several data types that are not oriented toward numbers or characters. Instead, they deal with specialized data such as yes/no and date/time values.

In This Section

Boolean Type
Discusses true and false values.
Date Type
Discusses date and time values.
Object Type
Explains the type that is a pointer to an object or data of any type.

Related Sections

Data Types
Introduces Visual Basic .NET data types and describes how to use them.
Data Type Summary
Provides an overview of the elementary data types supplied by Visual Basic .NET.

Boolean Type

The **Boolean** data type is an unsigned value that is interpreted as either **True** or **False**. If a variable can contain only true/false, yes/no, or on/off information, declare it as type **Boolean**.

The default value of **Boolean** is **False**. In the following example, `RunningVB` is a **Boolean** variable, which stores a simple yes/no setting.

```
Dim RunningVB As Boolean
' Check to see if program is running on VB engine.
If ScriptEngine = "VB" Then
   RunningVB = True
End If
```

No literal type character exists for **Boolean**.

See Also

Boolean Data Type

Date Type

The **Date** data type is a 64-bit (8-byte) signed integer. Each increment represents 100 nanoseconds of elapsed time since the beginning (12:00 AM) of January 1 of the year 1 in the Gregorian calendar. A variable or constant of the **Date** data type holds both the date and the time, as in the following example:

```
SomeDate = #8/13/2002 12:14 PM#
```

If you do not include a date in a date/time literal, Visual Basic sets the date part of the value to January 1, 0001. If you do not include a time in a date/time literal, Visual Basic sets the time part of the value to the start of the day, that is, midnight.

When you convert a numeric data type to **Date**, Visual Basic treats its value as a signed integer. If you attempt to convert a negative number, an error results.

No literal type character exists for **Date**. However, the compiler treats literals enclosed within number sign (**# #**) characters as **Date**.

See Also

Date Data Type

Object Type

The **Object** data type is a 32-bit (4-byte) address that points to an object within your application or in some other application. When you declare a variable as **Object**, you can subsequently use it to refer to any object recognized by your application, as in the following example:

```
Dim objDb As Object
Dim MyCollection As New Collection()
' ... Suppose MyCollection has now been populated.
objDb = MyCollection.Item(1)
```

When you declare an object variable, try to use a specific class rather than the generic **Object**, for example **Database** instead of **Object** in the preceding example. You should also use the most specific class available, such as **TextBox** instead of **Control**, so that you can access its properties and methods. You can usually use the **Classes** list in the **Object Browser** to find available class names.

Variables of the **Object** data type can also point to data of any type. This means you can assign a variable, constant, or expression of any data type to an **Object** variable. To determine the data type an **Object** variable currently refers to, you can use the **GetTypeCode** method of the **Type** class, as in the following example:

```
Dim MyObject As Object
' ... Suppose MyObject has now had something assigned to it.
Dim DatTyp As Integer
DatTyp = Type.GetTypeCode(MyObject.GetType())
```

Programming elements declared as **Object** can accept values of any data type. **Object** is treated as a value type when holding a value type, and as a reference type when holding a reference type. In either case, an **Object** variable does not contain the value itself, but rather a pointer to the value. It always uses four bytes in computer memory, but this does not include the storage for the data representing the value of the variable. Because of the code that uses the pointer to locate the data, **Object** variables holding value types are slightly slower to access than explicitly typed variables.

No literal type character exists for **Object**.

Determining Inheritance

You can compare two objects to determine the relationship, if any, between the classes from which they are created. The **IsInstanceOfType** method of the **Type** class returns **True** if the current class inherits from the specified class, or if the current type is an interface supported by the specified class. The following example determines whether one object represents a class inherited from another object's class:

```
Public Class BaseClass
End Class
Public Class DerivedClass : Inherits BaseClass
End Class
Public Class TestTheseClasses
   Public Sub SeeIfRelated()
      Dim BaseObj As Object = New BaseClass()
      Dim DerivedObj As Object = New DerivedClass()
      Dim Related As Boolean
      Related = BaseObj.GetType().IsInstanceOfType(DerivedObj)
      MsgBox(CStr(Related))
   End Sub
End Class ' TestTheseClasses
```

Note the unexpected placement of the two object variables in the use of **IsInstanceOfType**. The supposed base type is used to generate the **Type** class, and the supposed derived type is passed as an argument to **IsInstanceOfType**.

See Also

Type Conversions | Object Variables | Type Class | Type.GetTypeCode Method | Type.IsInstanceOfType Method | Object Data Type

Composite Data Types

In addition to the elementary data types Visual Basic supplies, you can also assemble items of different types to create *composite data types* such as structures, arrays, and classes. You can build composite data types from elementary types and from other composite types. For example, you can define an array of structure elements, or a structure with array members.

A composite type is different from the data type of any of its components. For example, an array of **Integer** elements is not of the **Integer** data type.

There is no single data type comprising all structures. Instead, each definition of a structure represents a unique data type. The same uniqueness is true for classes.

There is also no single data type comprising all arrays. The data type of a particular instance of an array is determined by:

- The fact of being an array
- The rank (number of dimensions) of the array
- The element type of the array

In particular, the length of a given dimension is not part of the instance's data type. This is illustrated in the following example:

```
Dim A As Byte( ) = New Byte(12) {}
Dim B As Byte( ) = New Byte(100) {}
Dim C As Short( ) = New Short(100) {}
Dim D As Short( , )
Dim E As Short( , ) = New Short(4, 10) {}
```

In the preceding example, array variables `A` and `B` are considered to be of the same data type, even though they are initialized to different lengths. Array variables `B` and `C` are not of the same type because their element types are different. `C` and `D` are not of the same type because their ranks are different. `D` and `E` have the same type because their ranks and element types are the same, even though `D` is not yet initialized.

See Also

Arrays | Structures: Your Own Data Types | Data Types as Classes and Structures

Data Type Implementation

The Visual Basic .NET data types can be classified according to whether a variable of a particular type stores its own data or a pointer to the data. This classification affects the way a data type is implemented.

In This Section

Value Types and Reference Types
Explains the difference between value and reference types, and which data types are which.
Data Types as Classes and Structures
Describes how Visual Basic data types are implemented in the underlying .NET Framework.

Related Sections

Data Types
Introduces the Visual Basic .NET data types and describes how to use them.
Data Type Summary
Lists the elementary data types supplied by Visual Basic .NET.

Value Types and Reference Types

A data type is a *value type* if it holds the data within its own memory allocation. A *reference type* contains a pointer to another memory location that holds the data.

Value types include:

- All numeric data types
- **Boolean**, **Char**, and **Date**
- All structures, even if their members are reference types
- 1Enumerations, since their underlying type is always **Byte**, **Short**, **Integer**, or **Long**

Reference types include:

- **String**
- All arrays, even if their elements are value types
- Class types, such as **Form**
- Delegates

You can assign either a reference type or a value type to a variable of the **Object** data type. An **Object** variable always holds a pointer to the data, never the data itself. However, if you assign a value type to an **Object** variable, it behaves as if it holds its own data.

You can find out whether an **Object** variable is acting as a reference type or a value type by passing it to the **IsReference** method on the **Information** class in the **Microsoft.VisualBasic** namespace. **Microsoft.VisualBasic.Information.IsReference** returns **True** if the content of the **Object** variable represents a reference type.

See Also

Data Types as Classes and Structures | Type Conversions | Structures: Your Own Data Types | Efficient Use of Data Types

Data Types as Classes and Structures

Every elementary data type in Visual Basic .NET is supported by a structure or a class in the **System** namespace. The compiler uses each data type keyword as an alias for the underlying structure or class. For example, declaring a variable with the reserved word **Byte** is the same as declaring it with the fully qualified structure name **System.Byte**.

The common language runtime also supports structures and classes that Visual Basic .NET does not supply. For example, the **System.UInt16**, **System.UInt32**, and **System.UInt64** structures provide the unsigned equivalents of **Short**, **Integer**, and **Long**. You can use these unsigned types to declare variables and constants, and you can access the methods .NET implements on these types. However, Visual Basic .NET does not support operations or type conversions involving types it does not supply.

In the .NET Framework, a structure is a value type and a class is a reference type. For this reason, value types such as **Char** and **Integer** are supported by .NET structures, while reference types such as **Object** and **String** are supported by .NET classes.

Since every reference type represents an underlying .NET class, you must use the **New** keyword when you initialize it. The following statement initializes an array:

```
Dim Totals() As Single = New Single(8) {}
```

You can also use the **New** keyword to initialize a value type. This is particularly useful if the type has a constructor that takes parameters.

Data Type Members

Because they are supported by .NET structures and classes, Visual Basic data types have members. These members include constructors, methods, properties, and fields. You can access the members (except the constructors) on a variable the same way you access methods and properties on an object. The following example uses the **Year**, **Month**, and **Day** properties and the **DaysInMonth** method of the **System.DateTime** structure to determine how many days are remaining in the current month:

```
Dim Current As Date   ' Date data type is supported by System.DateTime.
Dim DaysRemaining As Integer   ' Days remaining in current month.
' ...
Current = Now   ' Get current date/time into Current.
DaysRemaining = Date.DaysInMonth(Current.Year, Current.Month) - Current.Day
```

Note that you must qualify a reference to a data type member with either the name of the type (`Date`) or the name of a variable declared to be of that type (`Current`).

Data Type Member Examples

The following code prototypes illustrate some of the useful methods, properties, and fields on the data types:

```
<Char>.IsDigit()  ' Returns True if character is a numeric digit.
<Char>.IsLower()  ' Returns True if character is a lowercase letter.
<Date>.IsLeapYear()  ' Returns True if current year is a leap year.
<Date>.ToUniversalTime()  ' Returns local date/time converted to UTC.
<Double>.IsInfinity()  ' Returns True if contents represent infinity.
<Double>.IsNaN()  ' Returns True if contents are not a number (0/0).
<Long>.MaxValue  ' Constant representing largest positive Int64 value.
<Object>.GetType()  ' Returns Type object representing type of <Object>.
<Object>.GetType().GetTypeCode()  ' Returns type code of <Object>.
<String>.Chars(<index>)  ' Character at position <index> of <String>.
<String>.Length  ' Number of characters currently in <String>.
```

All numeric types, including **Byte** and **Char**, expose the **MaxValue** and **MinValue** public fields, which can be very useful when dealing with these types.

Data Type Member Equivalence

The .NET Framework also supplies a number of methods on data types that might appear to be equivalent to Visual Basic functions and keywords. For example, **System.Decimal.ToSingle()** performs the same type of action as the **CSng** keyword does on a **Decimal** expression. However, Visual Basic does not always use the .NET Framework methods to accomplish conversion or

other operations. **CSng** is not guaranteed to use **System.<*data type*>.ToSingle()**, and therefore the results are not guaranteed to be the same under marginal or boundary conditions.

See Also

Value Types and Reference Types | Structures and Classes | System Namespace | Boolean Structure | Byte Structure | Char Structure | DateTime Structure | Decimal Structure | Double Structure | Int16 Structure | Int32 Structure | Int64 Structure | Object Class | Single Structure | String Class

Type Conversions

The process of changing a value from one data type to another type is called *conversion*. Conversions are either widening or narrowing, depending on the data capacities of the types involved. They are also implicit or explicit, depending on the syntax in the source code.

In This Section

Widening and Narrowing Conversions
Explains conversions classified by whether the destination type can hold the data.
Implicit and Explicit Conversions
Explains conversions classified by whether Visual Basic performs them automatically.
Value Changes During Conversions
Describes how converting to another data type can change the value or data representation.
Conversions Between Strings and Other Types
Illustrates converting between strings and numeric, **Boolean**, or date/time values.
Array Conversions
Steps you through the process of converting between arrays of different data types.

Related Sections

Data Types
Introduces the Visual Basic .NET data types and describes how to use them.
Data Type Summary
Lists the elementary data types supplied by Visual Basic .NET.

Widening and Narrowing Conversions

An important characteristic of a type conversion is whether the result of the conversion is within the range of the destination data type. A *widening conversion* changes a value to a data type that can accommodate any possible value of the original data. A *narrowing conversion* changes a value to a data type that might not be able to hold some of the possible values.

Widening Conversions

The following table shows the standard widening conversions.

Data type	Widens to data types
Byte	**Byte**, **Short**, **Integer**, **Long**, **Decimal**, **Single**, **Double**
Short	**Short**, **Integer**, **Long**, **Decimal**, **Single**, **Double**
Integer	**Integer**, **Long**, **Decimal**, **Single**, **Double**
Long	**Long**, **Decimal**, **Single**, **Double**
Decimal	**Decimal**, **Single**, **Double**
Single	**Single**, **Double**
Double	**Double**
Any enumerated type	Its underlying integer type and any type to which that widen s
Char	**Char**, **String**
Any type	**Object**
Any derived type	Any base type from which it is derived
Any type	Any interface it implements
Nothing	Any data type or object type

Conversions from **Integer** to **Single**, from **Long** to **Single** or **Double**, or from **Decimal** to **Single** or **Double**, might result in loss of precision, but never in loss of magnitude. In this sense they do not incur information loss.

It might seem surprising that a conversion from a derived type to one of its base types is widening. The justification is that the derived type contains all the members of the base type, so it qualifies as an instance of the base type. In the opposite direction, the base type does not have the members defined by the derived type.

Widening conversions always succeed and can always be performed implicitly.

Narrowing Conversions

The standard narrowing conversions include:

- The reverse directions of the widening conversions in the preceding table
- Conversions in either direction between **Boolean** and any numeric type
- Any numeric type to any enumerated type
- Conversions in either direction between a **Char()** array and a **String**
- Conversions in either direction between **String** and any numeric type, **Boolean**, or **Date**
- Conversions from a data type or object type to a type derived from it

Narrowing conversions do not always succeed, meaning that they might fail at run time. An error occurs if the destination data type cannot receive the value being converted. For example, a numeric conversion can result in an overflow. The compiler usually does not allow narrowing conversions to be performed implicitly.

You use a narrowing conversion when you know the source value can be converted to the destination data type without error. For example, if you have a **String** that you know contains either "True" or "False", you can use the **CBool** keyword to convert it to **Boolean**.

See Also

Implicit and Explicit Conversions | Value Changes During Conversions | Conversions Between Strings and Other Types | Typeless Programming | Data Type Summary | Type Conversion Functions | Boolean Data Type | Byte Data Type | Char Data Type | Date Data Type | Decimal Data Type | Double Data Type | Integer Data Type | Long Data Type | Object Data Type | Short Data Type | Single Data Type | String Data Type

Implicit and Explicit Conversions

An *implicit* conversion does not require any special syntax in the source code. In the following example, Visual Basic .NET implicitly converts the value of `K` to single-precision floating point before assigning it to `Q`.

```
Dim K As Integer
Dim Q As Double
' ...
K = 432   ' Integer widens to Double, so Option Strict can be On.
Q = K
```

An *explicit* conversion uses a type conversion keyword. Visual Basic .NET provides several such keywords, which coerce an expression in parentheses to the desired data type. These keywords act like functions, but the compiler generates the code inline, so execution is slightly faster than with a function call.

In the following extension of the preceding example, the **CInt** keyword converts the value of `Q` back to an integer before it is assigned to `K`:

```
Q = Math.Sqrt(Q)   ' Q had been assigned the value 432 from K.
K = CInt(Q)   ' K now has the value 21 (rounded square root of 432).
```

The following table shows the available conversion keywords.

Type conversion keyword	Converts an expression to data type	Allowable data types of expression to be converted
CBool	**Boolean**	Any numeric type (including **Byte** and enumerated types), **String**, **Object**
CByte	**Byte**	Any numeric type, any enumerated type, **Boolean**, **String**, **Object**
CChar	**Char**	**String**, **Object**
CDate	**Date**	**String**, **Object**
CDbl	**Double**	Any numeric type (including **Byte** and enumerated types), **Boolean**, **String**, **Object**
CDec	**Decimal**	Any numeric type (including **Byte** and enumerated types), **Boolean**, **String**, **Object**
CInt	**Integer**	Any numeric type (including **Byte** and enumerated types), **Boolean**, **String**, **Object**
CLng	**Long**	Any numeric type (including **Byte** and enumerated types), **Boolean**, **String**, **Object**
CObj	**Object**	Any type
CShort	**Short**	Any numeric type (including **Byte** and enumerated types), **Boolean**, **String**, **Object**
CSng	**Single**	Any numeric type (including **Byte** and enumerated types), **Boolean**, **String**, **Object**
CStr	**String**	Any numeric type (including **Byte**), **Boolean**, **Char**, **Char()** array, **Date**, **Object**
CType	Type specified following the comma (**,**)	When converting to an elementary type (including an array of an elementary type), the same types as allowed for the corresponding conversion keyword When converting to a composite type, the interfaces it implements and the classes from which it inherits

The CType Keyword

The **CType** keyword operates on two arguments. The first is the expression to be converted, and the second is the destination data type or object class. The following example demonstrates the use of **CType**.

```
K = CType(Q, Integer)  ' Uses CType keyword instead of CInt.
F = CType(W, Label)    ' Coerces W to the specific object class Label.
```

You can use **CType** to convert values to composite data types as well as to elementary types. You can also use it to coerce an object class to the type of one of its interfaces, as in the following example:

```
' Assume class CZone implements interface IZone.
Dim H As Object
Dim CZ As CZone   ' Cannot use a type as first argument of CType.
H = CType(CZ, IZone)   ' Coerces CZone object to its interface IZone.
```

CType can also convert array types, as in the following example:

```
Dim V() As ClassV   ' Array of ClassV objects.
Dim ObArray() As Object   ' Array of objects to be assigned.
' Some object array is assigned to ObArray.
If TypeOf ObArray Is ClassV()   ' Check for run-time type compatibility.
   V = CType(ObArray, ClassV())   ' ObArray can be converted to ClassV.
End If
```

> **Note** Values used with a conversion keyword must be valid for the destination data type, or an error occurs. For example, if you attempt to convert a **Long** to an **Integer**, the value of the **Long** must be within the valid range for the **Integer** data type.
>
> **Caution** Specifying **CType** to convert from one class type to another fails at run time if there is no inheritance relationship between the two types.

Performing an explicit conversion is also known as *casting* an expression to a given data type or object class.

See Also

Widening and Narrowing Conversions | Value Changes During Conversions | Conversions Between Strings and Other Types | Structures: Your Own Data Types | Typeless Programming | Data Type Summary | Type Conversion Functions

Value Changes During Conversions

A conversion from a value type stores a copy of the source value in the destination of the conversion. However, this copy is not an exact image of the source value. The destination data type stores values differently, and even the value being represented might change, depending on the kind of conversion being performed.

Changes During Widening and Narrowing Conversions

Narrowing conversions change the destination copy of the source value with potential information loss. For example, a fractional value is rounded when converted to an integral type, and a numeric type being converted to **Boolean** is reduced to either **True** or **False**.

Widening conversions preserve the value but can change its representation. This happens if you convert from an integral type to **Decimal**, or from **Char** to **String**.

The original source value is not changed as a result of a conversion.

Changes During Reference Type Conversions

A conversion from a reference type copies only the pointer to the value. The value itself is neither copied nor changed in any way. The only thing that changes is the data type of the variable holding the pointer. In the following example, the data type is converted from the derived class to its base class, but the object now pointed to by both variables is unchanged:

```
' Assume class CSquare inherits from class CShape.
Dim Shape As CShape
Dim Square As CSquare = New CSquare
' ...
Shape = Square   ' Widening conversion from derived class to base class.
```

See Also

Widening and Narrowing Conversions | Implicit and Explicit Conversions | Conversions Between Strings and Other Types | Typeless Programming

Conversions Between Strings and Other Types

You can convert a numeric, **Boolean**, or date/time value to a **String**. You can also convert in the reverse direction — from a string value to numeric, **Boolean**, or **Date** — provided the contents of the string can be interpreted as a valid value of the destination data type. If they cannot, a run-time error occurs.

The conversions for all these assignments, in either direction, are narrowing conversions. You should use the type conversion keywords (**CStr**, **CShort**, **CInt**, **CLng**, **CDec**, **CSng**, **CDbl**, **CBool**, **CDate**, and **CType**). The **Format** and **Val** functions give you additional control over conversions between strings and numbers.

Conversion of Numbers to Strings

You can use the **Format** function to convert a number to a formatted string, which can include not only the appropriate digits but also formatting symbols such as a currency sign (such as **$**), thousands separators or *digit grouping symbols* (such as **,**), and a decimal separator (such as **.**). **Format** automatically uses the appropriate symbols according to the **Regional Options** settings specified in the Windows **Control Panel**.

Note that the concatenation (**&**) operator can convert a number to a string implicitly, as shown in the following example:

```
Str = "The total count is " & Count    ' Count is converted to String.
```

Conversion of Strings to Numbers

You can use the **Val** function to explicitly convert the digits in a string to a number. **Val** reads the string until it encounters a character other than a digit, space, tab, line feed, or period. The sequences "&O" and "&H" alter the base of the number system and terminate the scanning. Until it stops reading, **Val** converts all appropriate characters to a numeric value. For example, the following statement returns the value 141.8:

```
Val(" 14 1.8 miles")
```

When Visual Basic converts a string to a numeric value, it uses the **Regional Options** settings specified in the Windows **Control Panel** to interpret the thousands separator, decimal separator, and currency symbol. This means that a conversion might succeed under one setting but not another. For example, `"$14.20"` is acceptable in the English (United States) locale but not in any French locale.

See Also

Widening and Narrowing Conversions | Implicit and Explicit Conversions | Value Changes During Conversions

Typeless Programming

Data Type Summary | Type Conversion Functions | Format Function | Val Function | Introduction to International Applications in Visual Basic and Visual C#

Array Conversions

You can convert an array type to a different array type provided the following conditions are met:

- The ranks of the two arrays must be the same; that is, they must have the same number of dimensions. However, the lengths of the respective dimensions do not need to be the same.
- The data types of the elements of both arrays must be reference types. You cannot convert an **Integer** array to a **Long** array, or even to an **Object** array, because at least one value type is involved.
- A conversion, either widening or narrowing, must be possible between the element types of the two arrays. An example that fails this requirement is an attempted conversion between a **String** array and an array of a class derived from **System.Enum**. These two types have nothing in common, and no conversion of any kind exists between them.

A conversion of one array type to another is widening or narrowing depending on whether the conversion of the respective elements is widening or narrowing.

Conversion to an Object Array

When you declare an **Object** array without initializing it, its element type is **Object** as long as it remains uninitialized. When you set it to an array of a specific class, it takes on the type of that class. However, its underlying type is still **Object**, and you can subsequently set it to another array of an unrelated class. Since all classes derive from **Object**, you can change the array's element type from any class to any other class. In the following example, no conversion exists between types `Student` and **String**, but both derive from **Object**, so all assignments are valid:

```
' Assume Student has already been defined as a class.
Dim MyArray() As Object   ' MyArray is an Object array at this point.
Dim Names() As String = New String(3) {"Name0", "Name1", "Name2", "Name3"}
' ...
MyArray = New Student(3) {}   ' MyArray is now of type Student.
MyArray = Names   ' MyArray is now of type String.
```

If you originally declare an array with a specific class, its underlying element type is that class. If you subsequently set it to an array of another class, there must be a conversion between the two classes. In the following example, `Students` is a `Student` array. Since no conversion exists between **String** and `Student`, the last statement fails:

```
Dim Students() As Student   ' Students is a Student array at this point.
Dim Names() As String = New String(3) {"Name0", "Name1", "Name2", "Name3"}
' ...
Students = New Student(3) {}   ' Initialize to elements of type Student.
Students = Names   ' Fails at compile time -- no conversion exists.
```

See Also

Value Types and Reference Types | Widening and Narrowing Conversions | Arrays | Advanced Features of Arrays

Structures: Your Own Data Types

You can combine data items of different types to create a *structure*. A structure associates one or more *members* with each other and with the structure itself. When you declare a structure, it becomes a composite data type, and you can declare variables of that type.

A structure is a generalization of the user-defined type (UDT) supported by previous versions of Visual Basic. In addition to fields, structures can expose properties, methods, and events. A structure can implement one or more interfaces, and you can declare individual accessibility for each field.

Structures are useful when you want a single variable to hold several related pieces of information. For example, you might want to keep an employee's name, telephone extension, and salary together. You could use several variables for this information, or you could define a structure and use it for a single employee variable. The advantage of the structure becomes apparent when you have many employees and therefore many instances of the variable.

In This Section

Structure Declaration
Shows how to declare a structure and its members.
Structure Variables
Covers assigning a structure to a variable and accessing its members.
Structures and Other Programming Elements
Summarizes how structures interact with arrays, objects, procedures, and each other.
Structures and Classes
Describes the similarities and differences between structures and classes.

Related Sections

Data Types
Introduces the Visual Basic .NET data types and describes how to use them.
Data Type Summary
Lists the elementary data types supplied by Visual Basic .NET.

Structure Declaration

You begin a structure declaration with the **Structure** statement, and you end it with the **End Structure** statement. Between these two statements you must declare at least one member. The members can be of any data type, but you must include at least one nonshared variable or an event.

If you want to keep an employee's name, telephone extension, and salary together in a single variable, you can declare a structure for this information as follows:

```
Private Structure Employee
   Public GivenName As String   ' This employee's given name.
   Public FamilyName As String   ' This employee's family name.
   Public Extension As Long   ' This employee's telephone extension.
   Private Salary As Decimal   ' This employee's annual salary.
   Public Sub GiveRaise(Raise As Double)   ' Raise this employee's salary.
      Salary *= Raise
   End Sub
   Public Event ReviewTime()   ' This employee must be reviewed.
End Structure
```

If you declare a structure as **Private**, as in the preceding example, it is inaccessible from outside an object created from the class defining the structure. The `Salary` field in the preceding example is also **Private**, which means it is inaccessible outside the structure, even from a class that can access the structure. However, the `GiveRaise()` procedure is **Public**, so it can be called from outside.

You can also define **Function** procedures, properties, and events in a structure. You can define one property as the default property, provided it takes at least one argument. You can handle an event with a **Shared Sub** procedure.

You can specify the accessibility of a structure using the **Public**, **Protected**, **Friend**, or **Private** keyword, or you can let it default to **Public**. You must declare every member and specify an accessibility for it. If you use the **Dim** statement without any keywords, the accessibility defaults to **Public**.

> **Note** You cannot initialize any of the structure members in the structure declaration. When you declare a variable to be of a structure type, you assign values to the members by accessing them through the variable.

See Also

Structure Statement | User-Defined Data Type

Structure Variables

Once you have created a structure, you can declare procedure-level and module-level variables as that type. For example, you can create a structure that records information about a computer system, as in the following code:

```
Public Structure SystemInfo
   Public CPU As String
   Public Memory As Long
   Public PurchaseDate As Date
End Structure
```

You can now declare variables of that type, as follows.

```
Dim MySystem, YourSystem As SystemInfo
```

> **Note** In classes and modules, structures declared using the **Dim** statement default to public access. If you intend a structure to be private, make sure you declare it using the **Private** keyword.

Access to Structure Values

To assign and retrieve values from the elements of a structure variable, you use the same syntax as you use to set and get properties on an object. For example, using the variables previously declared as `SystemInfo`, you can access their elements as follows.

```
MySystem.CPU = "486"
Dim TooOld As Boolean
If YourSystem.PurchaseDate < #1/1/1992# Then TooOld = True
```

You can also assign one variable to another if both are of the same structure type. This copies all the elements of one structure to the corresponding elements in the other, as in the following example:

```
YourSystem = MySystem
```

If a structure element is a reference type, such as a **String**, **Object**, or array, the pointer to the data is copied. If `SystemInfo` had included an object variable, then the preceding example would have copied the pointer from `MySystem` to `YourSystem`, and a change to the object's data through one structure would be in effect when accessed through the other structure.

See Also

Structures: Your Own Data Types | Structures and Other Programming Elements | Structures and Classes

Structure Statement

Structures and Other Programming Elements

You can use structures in conjunction with arrays, objects, and procedures, as well as with each other. The interactions use the same syntax as these elements use individually.

> **Note** You cannot initialize any of the structure members in the structure declaration. You can assign values only to members of a variable that has been declared to be of a structure type.

Structures and Arrays

A structure can contain an array as one of its elements, as shown in this example:

```
Public Structure SystemInfo
   Public CPU As String
   Public Memory As Long
   Public DiskDrives() As String   ' Array of changeable size.
   Public PurchaseDate As Date
End Structure
```

You access the values of an array within a structure the same way you access a property on an object, as shown in this example:

```
Dim MySystem As SystemInfo
ReDim MySystem.DiskDrives(3)
MySystem.DiskDrives(0) = "1.44 MB"
```

You can also declare an array of structures, as shown in this example:

```
Dim AllSystems(100) As SystemInfo
```

You follow the same rules to access the components of this data architecture, as shown in this example:

```
ReDim AllSystems(5).DiskDrives(3)
AllSystems(5).CPU = "386SX"
AllSystems(5).DiskDrives(2) = "100M SCSI"
```

Structures and Objects

A structure can contain an object as one of its elements, as shown in the following example:

```
Protected Structure UserInput
   Public UserName As String
   Public InputForm As System.Windows.Forms.Form
   Public UserFileNumber As Integer
End Structure
```

You should use a specific object class in such a declaration, rather than **Object**.

Structures and Procedures

You can pass a structure as a procedure argument, as shown in this example:

```
Public CurrentCPUName As String = "700MHz Pentium compatible"
Public CurrentMemorySize As Long = 256
' ...
Public Sub FillSystem(ByRef SomeSystem As SystemInfo)
   SomeSystem.CPU = CurrentCPUName
   SomeSystem.Memory = CurrentMemorySize
   SomeSystem.PurchaseDate = Now
End Sub
```

The preceding example passes the structure by reference, which allows the procedure to modify its elements so that the changes take effect in the calling code. If you want to protect a structure against such modification, pass it by value.

You can also return a structure from a function procedure, as shown in this example:

```
Dim AllSystems(100) As SystemInfo
' ...
Function FindByDate(ByVal SearchDate As Date) As SystemInfo
   Dim I As Integer
   For I = 1 To 100
      If AllSystems(I).PurchaseDate = SearchDate Then Return AllSystems(I)
   Next I
   ' Process error: system with desired purchase date not found.
End Function
```

Structures Within Structures

Structures can contain other structures, as shown in the following example:

```
Public Structure DriveInfo
   Public Type As String
   Public Size As Long
End Structure
' ...
Public Structure SystemInfo
   Public CPU As String
   Public Memory As Long
   Public DiskDrives() As DriveInfo   ' Array of changeable size.
   Public PurchaseDate As Date
End Structure
' ...
Dim AllSystems(100) As SystemInfo
ReDim AllSystems(1).DiskDrives(3)
AllSystems(1).DiskDrives(0).Type = "Floppy"
```

You can also use this technique to encapsulate a structure defined in one module within a structure defined in a different module.

Structures can contain other structures to an arbitrary depth.

See Also

Structures: Your Own Data Types | Structure Variables | Structures and Classes | Function Procedures | Procedure Arguments | Structure Statement

Structures and Classes

Visual Basic .NET unifies the syntax for structures and classes, with the result that both entities support most of the same features. However, there are also important differences between structures and classes.

Similarities

Structures and classes are similar in the following respects:

- Both are *container* types, meaning that they contain other types as members.
- Both have members, which can include constructors, methods, properties, fields, constants, enumerations, events, and event handlers.
- Members of both can have individualized accessibilities. For example, one member can be declared **Public** and another **Private**.
- Both can implement interfaces.
- Both can have shared constructors, with or without parameters.
- Both can expose a default property, provided that property takes at least one argument.
- Both can declare and raise events, and both can declare delegates.

Differences

Structures and classes differ in the following particulars:

- Structures are value types; classes are reference types.
- Structures use stack allocation; classes use heap allocation.
- All structure members are **Public** by default; class variables and constants are **Private** by default, while other class members are **Public** by default. This behavior for class members provides compatibility with the Visual Basic 6.0 system of defaults.
- A structure must have at least one nonshared variable or event member; a class can be completely empty.
- Structure members cannot be declared as **Protected**; class members can.
- A structure procedure can handle events only if it is a **Shared Sub** procedure, and only by means of the **AddHandler** statement; any class procedure can handle events, using either the **Handles** keyword or the **AddHandler** statement.
- Structure variable declarations cannot specify initializers, the **New** keyword, or initial sizes for arrays; class variable declarations can.
- Structures implicitly inherit from the **ValueType** class and cannot inherit from any other type; classes can inherit from any class or classes other than **ValueType**.
- Structures are not inheritable; classes are.
- Structures are never terminated, so the common language runtime (CLR) never calls the **Finalize** method on any structure; classes are terminated by the garbage collector, which calls **Finalize** on a class when it detects there are no active references remaining.
- A structure does not require a constructor; a class does.
- Structures can have nonshared constructors only if they take parameters; classes can have them with or without parameters.

Every structure has an implicit public constructor without parameters. This constructor initializes all the structure's data members to their default values. You cannot redefine this behavior.

Instances and Variables

Because structures are value types, each structure variable is permanently bound to an individual structure instance. But classes are reference types, and an object variable can refer to various class instances. This distinction affects your usage of structures and classes in the following ways:

- A structure variable implicitly includes an initialization of the members using the structure's parameterless constructor. Therefore, `Dim S As Struct1` is equivalent to `Dim S As Struct1 = New Struct1()`.
- When you assign one structure variable to another, or pass a structure instance to a procedure argument, the current values of all the variable members are copied to the new structure. When you assign one object variable to another, or pass an object variable to a procedure, only the reference pointer is copied.

- You can assign the value **Nothing** to a structure variable, but the instance continues to be associated with the variable. You can still call its methods and access its data members, although variable members are reinitialized by the assignment. In contrast, if you set an object variable to **Nothing**, you dissociate it from any class instance, and you cannot access any members through the variable until you assign another instance to it.
- An object variable can have different class instances assigned to it at different times, and several object variables can refer to the same class instance at the same time. Changes you make to the values of class members affect those members when accessed through another variable pointing to the same instance. Structure members, however, are isolated within their own instance. Changes to their values are not reflected in any other structure variables, even in other instances of the same **Structure** declaration.
- Equality testing of two structures must be performed with a member-by-member test. Two object variables can be compared using the **Equals** method. **Equals** indicates whether the two variables point to the same instance.

See Also

Structures: Your Own Data Types | Structure Variables | Structures and Other Programming Elements | Value Types and Reference Types | Data Types as Classes and Structures | Structure Statement | Understanding Classes | ValueType Class | AddHandler and RemoveHandler | Class Statement | Object.Equals Method

Typeless Programming

By default, the Visual Basic compiler enforces two rules that affect your source code:

- Every local variable must appear in a declaration statement before being used in any other kind of statement.
- Every narrowing type conversion must be specified with a type conversion keyword.

You can change either or both of these rules by resetting corresponding compiler options. If you change the type conversion rule, the compiler operates under *permissive type semantics*, which allows you to make narrowing conversions without conversion keywords.

If you change the variable declaration rule, you can then use *typeless programming*, which permits you to refer to variables without first declaring them. Typeless programming is not usually recommended because it can lead to subtle execution errors involving data types. However, you might find it useful in some circumstances, such as porting an old application that did not declare variables.

In This Section

Implicit and Explicit Declaration
  Describes how to declare variables implicitly and explicitly.
Type Checking
  Explains how and when the compiler permits implicit narrowing conversions.
Object as the Universal Data Type
  Illustrates object-type data storage, conversions from other types, and special values.
Efficient Use of Data Types
  Explains strong typing and when you should use it.

Related Sections

Data Types
  Introduces the Visual Basic .NET data types and describes how to use them.
Data Type Summary
  Lists the elementary data types supplied by Visual Basic .NET.

Implicit and Explicit Declaration

By default, the Visual Basic compiler enforces *explicit declaration*, which requires that you declare every variable before you use it. You can remove this requirement and permit *implicit declaration*. Visual Basic provides a switch that controls explicit declaration. If you turn this switch off, you can use variables without declaring them.

You can set the explicit declaration switch on or off in any of the following ways:

- Setting the appropriate project property in the **Properties** window of the integrated development environment (IDE)
- Specifying the **/optionexplicit** command-line compiler option
- Including the **Option Explicit** statement at the beginning your code

If you use the **Option Explicit** statement, that setting overrides both the project property and compiler option settings, but only for the source code file in which it appears.

Setting **Option Explicit On** has the advantage of forcing type inference to be made at compile time instead of run time. This improves performance.

Implicit Declaration

If you turn explicit declaration off, you can implicitly declare a variable by simply using it in your code. All implicitly declared variables are given the data type **Object**. However, your application is more efficient if you declare all your variables explicitly and with a specific data type. This reduces the incidence of naming-conflict errors and spelling mistakes. It also allows the compiler to detect potential run-time errors such as assigning an **Integer** to a **Short**.

You can write a function in which you do not declare a local variable, as in the following example:

```
Function SafeSqrt(Num)
   TempVal = Math.Abs(Num)   ' Make sure value is positive for square root.
   Return Math.Sqrt(TempVal)
End Function
```

Visual Basic automatically creates `TempVal` as a local variable, which you can use as if you had explicitly declared it. While this is convenient, it can lead to subtle errors in your code if you misspell a variable name. Suppose you had written the function in the preceding example as follows:

```
Function SafeSqrt(Num)
   TempVal = Math.Abs(Num)   ' Make sure value is positive for square root.
   Return Math.Sqrt(TemVal)
End Function
```

At first glance, this code looks the same. But because the `TempVal` variable is misspelled as the argument to `Sqrt`, the compiler creates an additional local variable called `TemVal`, which is never assigned a value, and your function always returns zero.

When Visual Basic encounters a new name, it cannot determine whether you meant to implicitly declare a new variable or you just misspelled an existing one, so it creates a new variable with that name. Alternatively, the compiler might already have that name defined, and your code would use that definition unintentionally.

You can avoid problems with misnamed variables by using explicit declaration.

Explicit Declaration

Had explicit declaration been in effect for the module containing the `SafeSqrt` function in the preceding example, Visual Basic would have recognized `TempVal` and `TemVal` as undeclared variables and generated errors for both of them. As a result, you would then explicitly declare `TempVal`, as follows:

```
Function SafeSqrt(Num As Single) As Single
   Dim TempVal As Single = Math.Abs(Num)
   Return Math.Sqrt(TemVal)
End Function
```

With this revised code, you would understand the problem immediately because Visual Basic would display an error message for the incorrectly spelled `TemVal`. Because explicit declaration helps you catch these kinds of errors, it is recommended that you use it

with all your code.

> **Note** The **Option Explicit** statement operates on a file-by-file basis. It must be at the beginning of every source code file in which you want to control the enforcement of explicit variable declaration.

See Also

Variables | Lifetime | Visual Basic Naming Conventions | Data Type Summary | Option Explicit Statement | /optionexplicit | Setting Visual Basic Project Properties

Type Checking

When converting between data types, the Visual Basic compiler can operate under strict or permissive type semantics. If *strict type semantics* are in effect, only widening conversions are permitted implicitly, and narrowing conversions must be explicit. Under *permissive type semantics*, all widening and narrowing conversions can be done implicitly. Type semantics apply to conversions between all data types, including object types.

Visual Basic .NET provides a switch that controls type checking. By default, this switch is on, and the compiler enforces strict type semantics. If you turn the switch off, you can perform narrowing conversions without using conversion keywords.

You can set the type checking switch on or off in any of the following ways:

- Setting the appropriate project property in the **Properties** window of the integrated development environment (IDE)
- Specifying the **/optionstrict** command-line compiler option
- Including the **Option Strict** statement at the beginning your code

If you use the **Option Strict** statement, that setting overrides both the project property and compiler option settings, but only for the source code file in which it appears.

Setting **Option Strict On** has the advantage of forcing early binding. This improves performance.

See Also

Widening and Narrowing Conversions | Implicit and Explicit Conversions | Option Strict Statement | /optionstrict | Setting Visual Basic Project Properties

Object as the Universal Data Type

The **Object** data type is the root type in Visual Basic .NET. This means that all other data types and object types are derived from it, either directly or ultimately. It also means that any other data type, whether elementary or composite, can be converted to **Object**.

If you have turned **Option Strict** off, you can use **Object** as the universal data type. This is called *loose typing*. The following example illustrates this flexibility:

```
Dim V As Object
V = "17"   ' V contains the 2-character string "17".
V = V - 15   ' V now contains the integer value 2.
V = "H" & V   ' V now contains the 2-character string "H2".
```

While you can perform operations on **Object** variables without much concern for the kind of data they contain, there are some traps you must avoid.

- If you perform arithmetic operations or functions on an **Object**, it must hold numerical data.
- If you are concatenating strings, use the **&** operator instead of the **+** operator.

Changing Data Type

Although an **Object** variable can accept data of any type, it cannot be redimensioned to a different data type. However, you can redimension an **Object** variable to an array of **Object** variables. The following example illustrates both valid and invalid uses of the **ReDim** statement:

```
Dim SomeObj As Object   ' Initial declaration as a single Object.
ReDim SomeObj(8)   ' Valid reallocation for Object only.
ReDim SomeObj(8) As Double   ' INVALID change of data type.
```

The first **ReDim** statement changes `SomeObj` to an array of type **Object**. This is valid only with the **Object** data type. The second **ReDim** statement is invalid because it involves a different data type. You can achieve the desired effect with the following code:

```
Dim SomeArray(8) As Double  ' First allocate a separate array.
SomeObj = SomeArray         ' Assign new array to Object variable.
```

Object as an Alternative to a Structure

Because the **Object** data type can store many different types of data, you can use an **Object** array in many situations where you might expect to use a structure. An **Object** array is somewhat more flexible than a structure, because you can change the type of data you store in each element at any time, and you can make the array dynamic so you can change its size as necessary. However, an **Object** array uses more memory than an equivalent structure.

See Also

Object Variable Declaration | Object Variable Assignment | Object Variable Values | Universal Data Type Changes in Visual Basic | Type Conversions | Efficient Use of Data Types | Data Types | ReDim Statement

Efficient Use of Data Types

Undeclared variables and variables declared without a data type are given the **Object** data type. This makes it easy to write programs quickly, but it can cause them to execute more slowly.

Specifying data types for all your variables is known as *strong typing*. Using strong typing has several advantages:

- It enables IntelliSense™ support for your variables. This allows you to see their properties and other members as you type in the code.
- It allows the compiler to perform type checking. This catches statements that can fail at run time due to errors such as overflow. It also catches calls to methods on objects that do not support them.
- It results in faster execution of your code.

For variables that never contain fractions, the integral data types are more efficient than the nonintegral types. In Visual Basic .NET, **Integer** is the most efficient numeric type.

Use the **Dim** statement to declare a variable of a specific type. You can simultaneously specify its accessibility by using the **Public**, **Protected**, **Friend**, or **Private** keyword, as in the following example:

```
Private X As Double   ' X is a double-precision floating point.
Protected S As String   ' S is a character string.
```

Character Conversion

The **AscW** and **ChrW** functions operate in Unicode. You should use them in preference to **Asc** and **Chr**, which must translate into and out of Unicode.

See Also

Data Types | Data Type Declaration | Numeric Data Types | Variable Declaration | Asc, AscW Functions | Chr, ChrW Functions

Variables

You often need to store values temporarily when performing calculations with Visual Basic. For example, you might want to calculate several values, compare them, and perform different operations on them, depending on the result of the comparison. You need to retain the values if you want to compare them, but you don't need to store them in a property.

Visual Basic, like most programming languages, uses variables for storing values. A *variable* has a name (the word you use to refer to the value the variable contains) and a data type (which determines the kind of data the variable can store). A variable can represent an array if it needs to store an indexed set of closely related data items.

You use assignment statements to perform calculations and assign the result to a variable, as shown in the following code:

```
ApplesSold = 10              ' The value 10 is assigned to the variable.
ApplesSold = ApplesSold + 1  ' The variable is incremented.
```

> **Note** The equal sign (**=**) in this example is an assignment operator, not an equality operator. The value is being assigned to the variable `ApplesSold`.

In This Section

Variable Declaration
Describes how to declare a variable to tell the program about it in advance.

Object Variables
Describes how to use variables to refer to objects.

Related Sections

Arrays
Describes arrays, and how to declare and use them.

Declared Elements
Discusses programming elements that are defined in declaration statements.

Variable Declaration

You declare a variable to specify its name and characteristics. The declaration statement for variables is the **Dim** statement. Its location and contents determine the variable's characteristics.

Declaration Levels

A *local variable* is one that is declared within a procedure. A *module variable* is declared at module level, inside the module but not within any procedure internal to that module.

In a class or structure, the category of a nonlocal variable depends on whether or not it is shared. If it is declared with the **Shared** keyword, it is a *shared variable*, and it exists in a single copy shared among all instances of the class or structure. Otherwise it is an *instance variable*, and a separate copy of it is created for each instance of the class or structure. A given copy of an instance variable is available only to the instance for which it was created.

Declaring Data Type

The **As** clause in the declaration statement allows you to define the data type or object type of the variable you are declaring. You can specify any of the following types for a variable:

- An elementary data type, such as **Boolean**, **Long**, or **Decimal**
- A composite data type, such as an array or structure
- An object type, or class, from Visual Basic or another application, such as **Label** or **TextBox**

You can declare several variables in one statement without having to repeat the data type. In the following statements, the variables `I`, `J`, and `K` are declared as type **Integer**, `L` and `M` as **Long**, and `X` and `Y` as **Single**:

```
Dim I, J, K As Integer            ' All three are Integer variables.
Dim L, M As Long, X, Y As Single  ' L and M are Long, X and Y are Single.
```

For more information on data types, see Data Types. For more information on objects, see Object-Oriented Programming in Visual Basic and Programming with Components.

Declaring Lifetime

The *lifetime* of a variable is the period of time during which it is available for use. A local variable declared with a **Dim** statement exists only as long as its procedure is executing. When the procedure terminates, all its local variables disappear and their values are lost. However, if you declare a local variable using the **Static** keyword, it continues to exist and preserve its value even when the procedure ends.

Module, shared, and instance variables retain their values as long as your application continues to run.

For more information on lifetime, see Lifetime.

Declaring Scope

The *scope* of a variable is the set of all code that can refer to it without qualifying its name. A variable's scope is determined by where the variable is declared. Code located in a given region can use the variables defined in that region without having to qualify their names. When declaring scope, the following rules apply:

- The scope of a module variable is the entire namespace in which the module is defined.
- The scope of a shared or instance variable is the structure or class in which it is declared.
- The scope of a local variable is the procedure in which it is declared.
- However, if you declare a local variable within a block, its scope is that block only. A block is a set of statements terminated by an **End**, **Else**, **Loop**, or **Next** statement; for example, a **For...Next** or **If...Then...Else...End If** construction.

For more information on scope, see Scope.

Declaring Accessibility

A variable's accessibility is determined by which keyword or keywords — **Dim**, **Public**, **Protected**, **Friend**, **Protected Friend**, or **Private** — you use in the declaration statement. You can declare a module, structure, class, or instance variable with any of these

keywords. Within a procedure, only the **Dim** keyword is allowed, and the accessibility is always private.

For more information on accessibility, see Accessibility.

See Also

Choice of Variable Scope | Data Type Summary | Dim Statement | Public | Protected | Friend | Private | Shared | Static | Declared Element Characteristics

Choice of Variable Scope

The *scope* of a variable is the set of all code that can refer to it without qualifying its name. A variable's scope is determined by where the variable is declared.

When considering scope, keep in mind that local variables are a good choice for any kind of temporary calculation. They consume memory only when their procedure is running, and their names are not susceptible to conflict. For example, you can create several different procedures containing a variable called `intTemp`. As long as each `intTemp` is declared as a local variable, each procedure recognizes only its own version of `intTemp`. Any one procedure can alter the value in its local `intTemp` without affecting `intTemp` variables in other procedures.

> **Note** Module, shared, instance, and static variables consume memory resources until your application stops running, so use them only when necessary.

In general, when declaring any variable or constant, it is good programming practice to make the scope as narrow as possible (block scope is the narrowest). This helps conserve memory and minimizes the chances of your code erroneously referring to the wrong variable. Similarly, you should declare a variable to be **Static** only when it is necessary to preserve its value between procedure calls.

See Also

Variable Declaration | Scope | Dim Statement | Public | Protected | Friend | Private | Shared | Static

Object Variables

In addition to storing values, a variable can refer to an object. You assign an object to a variable for the same reasons you assign any value to a variable:

- A variable name is often shorter and easier to remember than the full path of methods or properties necessary to access the object itself.
- Using a variable that refers to an object is more efficient than repeatedly accessing the object itself through the necessary methods or properties.
- You can change a variable to refer to other objects while your code is running.

You can use object variables to shorten the code you have to type. For example, you might have a code segment like the following:

```
' Assume Me is a valid Form, or replace Me with a valid Form.
Me.Activate()   ' Make this the active form.
Me.ActiveControl = Label1   ' Make some valid control the active control.
' ...
Me.ActiveForm.ActiveControl.Text = "Test"
Me.ActiveForm.ActiveControl.Location = New Point(100, 100)
Me.ActiveForm.ActiveControl.Show()
```

You can shorten this code, and speed up execution, if you use an object variable for the control. You should declare the object variable with the specific class that you intend to assign to it (`Control` in the following example). Once you assign an object to the variable, you can treat it exactly the same as you do the object to which it refers. You can set or return the properties of the object or use any of its methods, as shown in the following example:

```
Dim ctrlBal As Control  ' Must declare to be of class Control.
' ...
ctrlBal = Me.ActiveForm.ActiveControl
ctrlBal.Text = "Test"
ctrlBal.Location = New Point(100, 100)
ctrlBal.Show()
```

In This Section

Object Variable Declaration
Discusses declaring object variables and choosing the class you declare them to be.

Object Variable Assignment
Describes how to assign objects to object variables, and combine assignments and declarations.

Object Variable Values
Describes how to return information about what an object variable refers to.

Related Sections

Variable Declaration
Describes how to specify variable names and characteristics, including which type of data the variable can contain.

Object Variable Declaration

You use a normal declaration statement to declare an object variable, but for the data type, you specify either **Object** or the *object class*. The object class is the specific class from which the object is to be instantiated. When you declare a variable with a specific object class, it can access all the methods and properties exposed by that class. If you declare the variable with **Object**, it can use only the members of the **Object** class, unless you turn off the type-checking switch (Option Strict Statement).

Use the following syntax to declare an object variable:

```
Dim variablename As [New] {objectclass | Object}
```

You can also use **Protected**, **Friend**, **Protected Friend**, **Private**, **Shared**, or **Static** for the declaration. The following declarations are valid:

```
Private ObjA As Object     ' Declare ObjA as generic Object data type.
Static ObjB As Label       ' Declare ObjB as Label class type.
Dim ObjC As System.Buffer  ' Declare ObjC as Buffer class type.
```

Sometimes the object class is not known until the procedure runs. In this case, you must declare the object variable with the **Object** data type. This creates a generic reference to any type of object. It also limits your code to the methods and properties of **System.Object** when **Option Strict** is turned on.

When you know the specific object class, you should declare the object variable to be of that class. In the preceding example declarations, if the application uses objects of class **Label**, you should specify **As Label** in the declarations for both `ObjA` and `ObjB`.

Declaring an object variable as a specific object class gives you several advantages:

- Automatic type checking
- Microsoft IntelliSense support in the Code editor
- Improved readability
- Fewer errors in your code
- Faster code execution

Access to Object Variable Members

When **Option Strict** is turned on, an object variable can access only the methods and properties of the class with which you declared it. The following example illustrates this:

```
' Option statements must precede all other source file lines.
Option Strict On
' Imports statement must precede all declarations in the source file.
Imports System.Windows.Forms
Dim P As Object
Dim Q As Label
P = New Label                ' Assign Label class to Object declaration.
Q = New Label                ' Assign Label class to Label declaration.
' ...
Dim J, K As Integer
J = P.Left                   ' Generates a compiler error.
K = Q.Left                   ' Returns left edge of label in pixels.
```

In this example, `P` can use only the members of the **Object** class itself, which do not include the `Left` property. On the other hand, `Q` was declared to be of type **Label**, so it can use all the methods and properties of the **Label** class in the **System.Windows.Forms** namespace.

Flexibility of Object Variables

When working with objects in an inheritance hierarchy, you have a choice of which class to use for declaring your object variables. In making this choice, you must balance flexibility of object assignment against access to a class's members. For example, consider the inheritance hierarchy that leads to the **Form** class:

Object Class

MarshalByRefObject
MarshalByRefComponent
Control Class
RichControl
ScrollableControl
ContainerControl
Form Class

Suppose your application defines a form called `Form2`, which inherits from class **Form**. You can declare an object variable that refers specifically to `Form2`, as follows:

```
Dim MyForm As New Form2    ' Can assign only Form2 objects to MyForm.
```

This limits the variable `MyForm` to objects of class `Form2`, but it also makes all the methods and properties of `Form2` available to `MyForm`, as well as all the members of all the classes from which `Form2` inherits.

You can make an object variable more general by declaring it to be of type **Form**, as follows:

```
Dim AnyForm As Form        ' Can assign any form, but no other control.
```

This allows you to assign any form in your application to `AnyForm`. However, although `AnyForm` can access all the members of class **Form**, it cannot use any of the additional methods or properties defined for specific forms such as `Form2`, unless you turn off **Option Strict**.

See Also

Object Variables | Object Variable Assignment | Object Variable Values | Typeless Programming | System Namespace | System.Windows.Forms Namespace | Object Class | Control Class | Form Class | Label Class | New

Object Variable Assignment

You use a normal assignment statement to assign an object to an object variable. You can assign an object expression or the **Nothing** keyword, as in the following code:

```
Dim MyObject, YourObject As Object
' ...
MyObject = YourObject  ' Assign an object reference.
MyObject = Nothing     ' Discontinue association with any object.
```

Nothing means there is no object currently assigned to the variable. When your code begins running, your object variables are initialized to **Nothing**, except for those whose declarations include assignment.

You can combine assignment with declaration by using the **New** keyword. The following declaration statements declare object variables and assign specific objects to them. The second statement uses one of the overloaded constructors of the **Version** class in the **System** namespace to initialize the object:

```
Dim MyURI As New System.Uri("http://www.microsoft.com")
Dim Major, Minor, Revision As Integer
Dim Ver As New System.Version(Major, Minor, Revision)
```

Setting an object variable to **Nothing** discontinues the association of the variable with any specific object. This prevents you from accidentally changing the object by changing the variable. It also allows you to test whether the object variable points to a valid object, as in the following example:

```
If Not MyObject Is Nothing Then
   ' MyObject refers to a valid object, so your code can use it.
   ' ...
End If
```

If the object your variable refers to is in another application, this test cannot determine whether that application has been closed.

Current Instance of an Object

The *current instance* of an object is the one in which the code is currently executing. Since all code executes inside a procedure, the current instance is the one in which the procedure was invoked.

The **Me** keyword acts as an object variable referring to the current instance. If a procedure is not shared, it can use the **Me** keyword to obtain a pointer to the current instance. Shared procedures cannot be associated with a single instance of an object.

Using **Me** is particularly useful for passing the current instance to a procedure in another module. For example, if you have a number of XML documents and wish to add some standard text to all of them, you could define the following procedure:

```
Sub AddStandardText(XmlDoc As System.Xml.XmlDocument)
   XmlDoc.CreateTextNode("This text goes into every XML document.")
End Sub
```

Every XML document object could then call the procedure and pass its current instance as an argument using the following statement:

```
AddStandardText(Me)
```

See Also

Object Variables | Object Variable Declaration | Object Variable Values | System Namespace | Buffer Class | Version Class | Nothing | New | Me

Object Variable Values

An object variable can refer to data of any type. The value you store in an object variable is kept elsewhere in memory, while the variable itself holds a pointer to the data.

Visual Basic supplies functions that return information about what an object variable refers to, as shown in the following table.

Function	Returns True if the Object variable refers to
IsArray	An array of values, rather than a single value
IsDate	A number or string that can be interpreted as a date and time valu e
IsNumeric	A number, or something that can be interpreted as a number

You can use these functions to avoid submitting an invalid value to an operation or a procedure.

You can also use the **TypeOf** operator to determine whether an object variable currently refers to a specific data type. The **TypeOf...Is** expression evaluates to **True** if the run-time type of the operand is derived from or implements the specified type. The following example uses **TypeOf** on object variables referring to value and reference types:

```
Dim Num As Object = 10          ' Value type (Integer).
Dim Frm As Object = New Form()  ' Reference type.
  ' ...
If TypeOf Num Is Long Then Debug.WriteLine("Num is Long")
If TypeOf Num Is Integer Then Debug.WriteLine("Num is Integer")
If TypeOf Num Is Short Then Debug.WriteLine("Num is Short")
If TypeOf Num Is Object Then Debug.WriteLine("Num is Object")
  ' ...
If TypeOf Frm Is Form Then Debug.WriteLine("Frm is Form")
If TypeOf Frm Is Label Then Debug.WriteLine("Frm is Label")
If TypeOf Frm Is Object Then Debug.WriteLine("Frm is Object")
```

The preceding example writes the following lines to the Debug window:

```
Num is Integer
Num is Object
Frm is Form
Frm is Object
```

The object variable `Num` refers to data of type **Integer**, and `Frm` refers to an object of class **Form**.

You can declare and use an array of **Object** variables. This is useful when you need to handle a variety of data types and object classes. All the elements in an array must have the same declared data type. Declaring this data type as **Object** allows you to store objects and class instances alongside other data types in the array.

See Also

Object Variables | Object Variable Declaration | Object Variable Assignment | System Namespace | System.Windows.Forms Namespace | Object Class | Form Class | Label Class | Data Types | TypeOf

Arrays

When you use arrays, you can refer to multiple variables by the same name, using a number called an *index* or *subscript* to distinguish them from one another. Arrays can shorten and simplify your code, allowing you to create loops that deal efficiently with any number of elements.

In This Section

Arrays Overview
Explains what arrays are and how they work, including information on dimensionality, size, and type.
Array Usage
Provides information on how to use arrays, including multidimensional arrays.
Declaring Array Variables
Provides information on using **Dim**, **Public**, **Protected**, **Friend**, **Protected Friend**, and **Private**.
Advanced Features of Arrays
Describes assigning the contents of one array to another, and returning an array from a function.
Arrays of Arrays
Describes creating an **Object** array and populating it with other arrays of different data types.
Collections as an Alternative to Arrays
Provides information on storing items in a collection instead of an array, a practice that can be more efficient in some scenarios.

Related Sections

Language Changes in Visual Basic
Provides an overview of the new features in Visual Basic .NET.
Object-Oriented Programming in Visual Basic
Covers object-oriented programming basics.

Arrays Overview

Arrays allow you to refer to a series of variables by the same name and to use a number, called an *index* or *subscript*, to tell them apart. This helps you create shorter and simpler code in many situations, because you can set up loops that deal efficiently with any number of elements by using the index number.

Array Dimensions

An array can have one dimension or more than one. The *dimensionality* or *rank* corresponds to the number of subscripts used to identify an individual element. You can specify up to 32 dimensions, although more than three is extremely rare.

Array Size

Every dimension of an array has a nonzero length. The elements of the array are contiguous along each dimension from subscript 0 through the highest subscript of that dimension. Because Visual Basic allocates space for an array element corresponding to each index number, you should avoid declaring any dimension of an array larger than necessary.

A *zero-length array* is an array with no elements. It is sometimes necessary to specify such an array, for example when using Windows Forms. To do this, you declare one of the array's dimensions to be –1. The array is then empty, but it still exists. Therefore, a variable pointing to the array is not equal to **Nothing**.

Arrays do not have fixed size in Visual Basic. You can change the size of an array after you have created it. The **ReDim** statement assigns a completely new array object to the specified array variable. Therefore, **ReDim** can change the length of each dimension.

Arrays as Objects

Arrays are objects in Visual Basic .NET, so every array type is an individual reference type. This has the following implications:

- An array variable holds a pointer to the data constituting the elements and the rank and length information.
- When you assign one array variable to another, only the pointer is copied.
- No two array variables are considered to be of the same data type unless they have the same rank and element data type.

Array Class

All arrays inherit from the **Array** class in the **System** namespace, and you can access the methods and properties of **System.Array** on any array. For example, the **Rank** property returns the array's rank, and the **Sort** method sorts its elements.

Array Element Type

An array declaration specifies a data type, and all its elements must be of that type. When the data type is **Object**, the individual elements can contain different kinds of data (objects, strings, numbers, and so on). You can declare an array of any of the fundamental data types, of a structure (described in Structures: Your Own Data Types), or of an object class (described in Object-Oriented Programming in Visual Basic).

You can also declare an array that contains other arrays as its elements. In this case, the constituent arrays must all have the same element data type, which you specify in the declaration. An array of arrays is called a *jagged array* because the constituent arrays do not have to have the same sizes.

> **Note** Jagged arrays are not compliant with the common language specification (CLS). This means you should not expose jagged arrays from any class you want CLS-compliant code to consume.

See Also

Array Usage | ReDim Statement | Declaring Array Variables | Advanced Features of Arrays | Arrays of Arrays | Collections as an Alternative to Arrays | Value Types and Reference Types | Parameter Arrays | Loop Structures | System Namespace | Array Class | Array.Rank Property | Array.Sort Method | Object Data Type

Array Usage

You can declare an array to work with a set of values of the same data type. An array is a single variable with many compartments to store values, while a scalar variable (that is, not an array) has only one storage compartment in which it can store only one value. You can refer to the array as a whole when you want to refer to all the values it holds, or you can refer to its individual elements one at a time.

For example, to store daily expenses for each day of the year, you can create one array with 366 elements, rather than declaring 366 variables. Each element in the array contains one value, which you access by specifying the element's index. The following example declares the array variable `CurExpense`, initializes it to have 366 elements, and then assigns an initial value of 20 to each element.

```
Sub FillArray()
   Dim CurExpense(365) As Decimal   ' Allocates (0) through (365).
   Dim I As Integer
   For I = 0 to 365
      CurExpense(I) = 20.00
   Next I
End Sub
```

You can mix data types in an array if it is declared of type **Object**. The following example stores employee information in the array variable `EmployeeData`.

```
Dim EmployeeData(3) As Object   ' Allocates (0) through (3).
EmployeeData(0) = "Ron Bendel"
EmployeeData(1) = "4242 Maple Blvd"
EmployeeData(2) = 48
EmployeeData(3) = "06-09-1953"
```

Multidimensional Arrays

In Visual Basic, you can declare arrays with up to 32 dimensions. For example, the following statement declares a two-dimensional array with 5 rows and 10 columns.

```
Dim Rectangle(4, 9) As Single   ' (0) through (4), (0) through (9).
```

The total number of elements is the product of the sizes of all the dimensions, in this case 50.

The rank of an array is held in its **Rank** property, and the length of each dimension is returned by the **GetLength** method. The lowest subscript value for a dimension is always 0, and the highest subscript value is returned by the **GetUpperBound** method for that dimension. Note that the argument you pass to **GetLength** and **GetUpperBound** (the dimension for which you want the length or the highest subscript) is 0-based.

> **Note** When you add dimensions to an array, the total storage needed by the array increases considerably, so use multidimensional and jagged arrays with care.

You can efficiently process a multidimensional array by using nested **For** loops. For example, the following statements initialize every element in `MatrixA` to a value between 0 and 99, based on its location in the array.

```
Dim I, J As Integer
Dim MaxDim0, MaxDim1 As Integer   ' Highest subscripts, not lengths.
Dim MatrixA(9, 9) As Double
MaxDim0 = MatrixA.GetUpperBound(0)   ' Maximum for first dimension.
MaxDim1 = MatrixA.GetUpperBound(1)   ' Maximum for second dimension.
For I = 0 To MaxDim0
   For J = 0 To MaxDim1
      MatrixA(I, J) = (I * 10) + J
   Next J
Next I
```

You can obtain the overall size of an array of any rank from its **Length** property. This represents the total number of elements

currently contained in the array, not the number of bytes they consume in storage. In the previous example, `MatrixA.Length` would return 100.

See Also

Object Data Type | Array.Rank Property | Array.GetLength Method | Array.GetUpperBound Method | Array.Length Property | Arrays Overview | Declaring Array Variables | Advanced Features of Arrays | Arrays of Arrays | Collections as an Alternative to Arrays

Declaring Array Variables

Array variables are declared the same way as other variables, using the **Dim** statement. You follow the variable name with one or more pairs of parentheses to indicate that it is an array rather than a *scalar* (a variable containing a single value).

To declare a single-dimensional array variable

- In your declaration, add one pair of parentheses after the variable name, as in the following example:

```
Dim MySingleArray() As Integer
```

To declare a multidimensional array variable

- In your declaration, add one pair of parentheses after the variable name and place commas inside the parentheses to separate the dimensions, as in the following example:

```
Dim My4DArray(,,,) As Short   ' Four-dimensional array.
```

To declare a jagged array variable

- In your declaration, add as many pairs of parentheses after the variable name as there are levels of nested arrays, as in the following example:

```
Dim MyJaggedArray()()() As Byte   ' Array of arrays of Byte arrays.
```

Initializing Arrays

You can initialize an array variable as part of its declaration. You can do one of the following in the declaration statement:

- Specify the initial length of one or more of the dimensions in the parentheses following the variable name, without assigning an array object to the variable.
- Assign an array object to the variable, using the **New** clause. When you use a **New** clause, you must follow it with braces (**{}**), even if they are empty.
- Assign an array object to the variable and supply initial lengths in the **New** clause.
- Assign an array object to the variable and supply initial element values in the **New** clause. You can supply both lengths and values in the same **New** clause.

If you specify dimension lengths in the parentheses following the variable name, you must use a subsequent assignment statement to assign an array object to the variable. The following sample declarations show valid and invalid syntax for a single-dimensional array variable:

```
Dim BA(2) As Byte   ' Currently contains Nothing (no object).
Dim BA(2) As Byte = New Byte()   ' INVALID (New after length specified).
Dim BA() As Byte = New Byte() {}   ' Empty Byte array object.
Dim BA() As Byte = New Byte()   ' INVALID (missing braces).
Dim BA() As Byte = New Byte(2)   ' INVALID (missing braces).
Dim BA() As Byte = New Byte() {0,1,2}   ' (0) through (2).
Dim BA() As Byte = New Byte(2) {0,1,2}   ' (0) through (2).
```

You can initialize a multidimensional array variable in a similar manner. The following sample declarations show a two-dimensional array variable being declared as an **Short** array with 2 rows and 2 columns.

```
Dim S2X2(1, 1) As Short   ' (0) through (1), (0) through (1).
Dim S2X2(1, 1) As Short = New Short(,)   ' INVALID (New after lengths).
Dim S2X2(,) As Short = New Short(,)   ' INVALID (missing braces).
Dim S2X2(,) As Short = New Short(1, 1)   ' INVALID (missing braces).
Dim S2X2(,) As Short = New Short(,) {}   ' Empty array object.
Dim S2X2(,) As Short = New Short(1, 1) {}   ' Elements have default value.
Dim S2X2(,) As Short = New Short(,) {{5, 6}, {7, 8}}   ' Four elements.
```

The first argument represents the rows; the second argument represents the columns. In the arrays in the preceding declarations, the index values for both the row and the column range from 0 through 1.

When you initialize a jagged array variable, you can specify the dimension lengths only for the top-level array. The following sample declarations show valid and invalid syntax for an array of arrays of **Byte** elements:

```
Dim JB(1)() As Byte   ' Array of two arrays of Byte elements.
Dim JB(1)(1) As Byte   ' INVALID (can only specify top-level size).
Dim JB(1)() As Byte = New Byte()()   ' INVALID (New after lengths).
Dim JB()() As Byte = {New Byte() {}, New Byte() {}}   ' Empty arrays.
Dim JB()() As Byte = {New Byte(1) {}, New Byte(1) {}}   ' Default values.
Dim JB()() As Byte = {New Byte() {5, 6}, New Byte() {7, 8}}
```

In the last of the preceding declarations, `JB` is initialized to two elements, `JB(0)` and `JB(1)`, each of which is initialized to a two-element **Byte** array, the first with element values 5 and 6 and the second with elements 7 and 8.

Array Size Limits

The length of every dimension of an array is limited to the maximum value of a **Long** data type, which is (2 ^ 64) - 1. The total size limit of an array varies, based on your operating system and how much memory is available. Using an array that exceeds the amount of RAM available on your system is slower because the data must be read from and written to disk.

> **Note** Because arrays are objects in Visual Basic .NET, it is important to distinguish between an array object and an array variable. An array object, once created, does not change its size or rank. An array variable, however, can have a succession of different arrays assigned to it during its lifetime, and these can be of different sizes and ranks.

Resizing Arrays

You can resize an array at any time by assigning a different array object to the same array variable, using either **ReDim** or a standard assignment statement. The new array object can have different dimensions, although it must retain the same rank. This helps you manage memory efficiently. For example, you can use a large array for a short time and then **ReDim** it to a smaller size. This frees up memory you no longer need.

When you **ReDim** an array, its existing values are normally lost. However, you can retain them by including the **Preserve** keyword in the **ReDim** statement. For example, the following statement allocates a new array, initializes its elements from the corresponding elements of the existing `MyArray`, and assigns the new array to `MyArray`.

```
ReDim Preserve MyArray(10, 20)
```

In a multidimensional array, you can change only the last dimension when you use **Preserve**. If you attempt to change any of the other dimensions, a run-time error occurs. If you do not know the current size of a dimension, you can use the **GetUpperBound** method, which returns the highest subscript value for the dimension you specify.

In the following example, the first line is valid, but the second line is not, because it attempts to change the first of two dimensions.

```
ReDim Preserve Matrix(Matrix.GetUpperBound(0), Matrix.GetUpperBound(1) + 10)
ReDim Preserve Matrix(Matrix.GetUpperBound(0) + 10, Matrix.GetUpperBound(1))
```

See Also

Arrays Overview | Array Usage | Advanced Features of Arrays | Arrays of Arrays | Collections as an Alternative to Arrays | Object-Oriented Programming in Visual Basic | Dim Statement | Byte Data Type | New | Short Data Type | Long Data Type | Preserve | Array.GetUpperBound Method | ReDim Statement | New

Advanced Features of Arrays

You can assign the contents of one array to another, create functions that accept and return arrays, and create properties that accept and return arrays. In many cases these techniques can improve the performance of your application.

Array Assignment

Since arrays are objects, they can be used in assignment statements like other object types. When assigning one array to another, the following rules apply:

- The rank (number of dimensions) of the destination array must be the same as that of the source array.
- Provided the ranks of the two arrays are equal, the dimensions do not need to be equal. The number of elements in a given dimension can change during assignment.
- Either both arrays must have reference type elements or both arrays must have value type elements.
- If both arrays have value type elements, the element data types must be exactly the same.
- If both arrays have reference type elements, there must be a widening conversion from the source element type to the destination element type.

The compiler reports an error if the above rules are violated, for example if a data type cannot be coerced or the ranks are unequal. You can add error handling to your code to make sure that the arrays are compatible before attempting an assignment.

Returning an Array from a Function

A function can return an array of values. For example, you might want to return a **Byte** array without having to perform conversions to and from a string. In the following example, the **Function** procedure `ArrayFunction` returns an array of bytes.

```
Dim B As Byte = CByte(54)   ' Seed value for ArrayFunction argument.
Dim I As Integer   ' Loop counter for displaying returned values.
Dim ReturnArray() As Byte   ' To be set from ArrayFunction return.
ReturnArray = ArrayFunction(B)
For I = 0 To ReturnArray.GetUpperBound(0)   ' Highest subscript.
   Debug.WriteLine("Byte " & I & ": " & ReturnArray(I))
Next I
' ...
Public Function ArrayFunction(ByVal B As Byte) As Byte()
   Dim X(2) As Byte   ' Allocates three elements.
   X(0) = B
   X(1) = B + B
   X(2) = B + CByte(200)
   Return X
End Function
```

After running the preceding example, `ReturnArray` points to a three-element array containing the element values assigned to the array `X` in `ArrayFunction`. In this example, `ReturnArray` must have the same element type as the function return because it is a value type (in this case **Byte**). If the element types of `ReturnArray` and `ArrayFunction` had been reference types, a widening conversion would have been acceptable.

See Also

Byte Data Type | Arrays Overview | Array Usage | Declaring Array Variables | Arrays of Arrays | Collections as an Alternative to Arrays | Value Types and Reference Types | Widening and Narrowing Conversions | Array Conversions

Arrays of Arrays

Sometimes the data in your application is two-dimensional but not rectangular. As an example, you might have an array of months, each element of which is an array of days. Since different months have different numbers of days, the elements do not form a rectangular two-dimensional array. In such a case, you can use a jagged array instead of a multidimensional array, as the following example shows:

```
Dim Sales()() As Double = New Double(11)() {}   ' Assign top-level array.
Dim Month As Integer   ' Counter variable for month.
Dim Days As Integer   ' Days in given month for this year.
For Month = 0 To 11
   Days = DateTime.DaysInMonth(Year(Now), Month + 1)   ' Get number of days.
   Sales(Month) = New Double(Days - 1) {}   ' Assign array for this month.
Next Month
```

The declaration of `Sales` sets the array variable to a 12-element array, each element of which is an array of **Double** elements. The **For** loop then determines how many days are in each month this year (**Year(Now)**), and sets the corresponding element of `Sales` to a **Double** array of the appropriate length.

In the preceding example, the jagged array saves seven elements (six in a leap year) compared to a two-dimensional array. In a more extreme case the memory savings could be significant.

> **Note** Jagged arrays are not compliant with the common language specification (CLS). This means you should not expose jagged arrays from any class you want CLS-compliant code to consume.

Arrays of Objects

It is also possible to create an **Object** array and populate it with other arrays of different data types. The following code creates two arrays, one containing integers and the other strings. It then declares a third array of type **Object** and populates it with the integer and string arrays.

```
Dim I As Integer   ' Counter variable.
' Declare and populate an integer array.
Dim CountersA(4) As Integer   ' Allocates five elements.
For I = 0 To 4
   CountersA(I) = I
Next I
' Declare and populate a string array.
Dim CountersB(4) As String   ' Allocates five elements.
For I = 0 To 4
   CountersB(I) = "String number " & CStr(I)
Next I
' Declare a new two-element array and populate it with the other arrays.
Dim Counters(1) As Object'   ' Elements (0) and (1).
Counters(0) = CountersA
Counters(1) = CountersB
' Display a member of each array.
MsgBox(Counters(0)(2))   ' Displays "2".
MsgBox(Counters(1)(3))   ' Displays "String number 3".
```

In the preceding example, the calls to **MsgBox** use late binding on `Counters(0)` and `Counters(1)`. `Counters(0)(2)` is a specific object instance created on `Counters(0)`, and `Counters(1)(3)` is an instance on `Counters(1)`. The Visual Basic compiler allows late binding only when **Option Strict** is **Off**.

See Also

Double Data Type | For...Next Statements | Object Data Type | MsgBox Function | Option Strict Statement | Arrays Overview | Array Usage | Declaring Array Variables | Advanced Features of Arrays | Collections as an Alternative to Arrays | DateTime Structure

Collections as an Alternative to Arrays

Although collections are most often used for working with objects, you can use a collection to work with any data type. In some circumstances, it can be more efficient to store items in a collection than in an array.

You might want to use a collection if you are working with a small, dynamic set of items. The following code fragment shows how you might use a collection to save and display a list of URL addresses.

```
' Module-level collection declaration.
Public URLHistory As New Microsoft.VisualBasic.Collection
' ...
' Code for adding a specified URL address to the collection.
Private Sub SaveURLHistory(ByVal URLAddress As String)
   URLHistory.Add(URLAddress)
End Sub
' ...
' Code for displaying the list of URL addresses in the Immediate window.
Private Sub PrintURLHistory()
   Dim URLAddress As String
   For Each URLAddress In URLHistory
      Debug.WriteLine(URLAddress)
   Next URLAddress
End Sub
```

See Also

Arrays Overview | Array Usage | Declaring Array Variables | Advanced Features of Arrays | Arrays of Arrays

Objects

When you create an application in Visual Basic .NET, you constantly work with objects. You can use objects provided by Visual Basic — such as controls, forms, and data access objects or can also control objects from other applications within your Visual Basic application. You can even create your own objects, and define additional properties and methods for them. Objects act like prefabricated building blocks for programs — they let you to write a piece of code once and reuse it over and over.

The following topics discuss objects in detail:

In This Section

What Is an Object?
Provides an overview of objects and classes.
Relation of Objects to Each Other
Discusses object hierarchies, collections, and containers.
The Basics of Working with Objects
Provides an overview of how objects are used.
Objects, Properties, Methods, and Events
Summarizes the properties, methods and events exposed by objects.

Related Sections

Creating and Using Objects
Shows how to create and use instances of classes.
Object-Oriented Programming in Visual Basic
Covers object-oriented programming basics.

What Is an Object?

An object is a combination of code and data that can be treated as a unit. An object can be a piece of an application, like a control or a form. An entire application can also be an object.

Objects let you declare variables and procedures once and then reuse them whenever needed. For example, if you want to add a spell checker to an application you could define all the variables and support functions to provide spelling check functionality to your application. However, if you create your spell checker as a class, you can reuse it in other applications by adding a reference to the compiled assembly. Better yet, you may be able to save yourself some work by using a spell checker class that someone else has already developed.

Each object in Visual Basic .NET is defined by a class. Classes describe the fields, properties, methods, and events of an object. Objects are instances of classes; you can create as many objects you need once you have created a class.

To understand the relationship between an object and its class, think of cookie cutters and cookies. The cookie cutter is the class. It defines the characteristics of each cookie — for instance, size and shape. The class is used to create objects. The objects are the cookies.

Two examples of the relationship between classes and objects in Visual Basic .NET may make this clearer:

- The controls on the Toolbox in Visual Basic .NET represent classes. When you drag and drop a control from the toolbox onto a form, you are actually creating an object —an instance of a class.
- The form you work with at design time is a class. At run time, Visual Basic .NET creates an instance of the form's class.

Objects are created as identical copies of their class. Once they exist as individual objects, however, their properties can be changed. For example, if you add three check boxes to a form, each check box button object is an instance of the **CheckBox** class. The individual **CheckBox** objects share a common set of characteristics and capabilities (properties, fields, methods, and events), defined by the class. However, each has its own name, can be separately enabled and disabled, can be placed in a different location on the form.

See Also

The Basics of Working with Objects | Relation of Objects to Each Other | Creating and Using Objects

Relation of Objects to Each Other

Objects can be related to each other in several ways. One such relationship is hierarchical. Classes that are derived from more fundamental classes are said to have a hierarchical relationship. Class hierarchies are useful when describing items that are a subtype of a more general class. Derived classes inherit members from the class they are based on, allowing you to add complexity as you progress in a class hierarchy.

Another way that objects are related is the object object-container relationship. Container objects encapsulate other objects. For example, an instance of class with a property that stores an object is a container object.

One type of object containment is represented by collections. Collections are like arrays of objects that can be enumerated. Visual Basic .NET supports a specific syntax in the form of **For Each...Next** syntax blocks that allow you to iterate the items of a collection. Additionally, collections often allow you to use a method named **Item** property to retrieve elements by their index or by associating them with a unique string. Collections are easier to use than arrays because they allow you to add or remove items without using indexes. Because of their ease of use, collections are often used to store forms and controls.

See Also

Designing an Inheritance Hierarchy | Collections in Visual Basic .NET | The Basics of Working with Objects | For Each...Next Statements (Conceptual) | For Each...Next Statements (Language Reference) | Setting and Retrieving Properties | Performing Actions with Methods | Objects from Visual Basic and Other Sources | Creating and Using Objects | Item Property

The Basics of Working with Objects

Almost everything you do in Visual Basic .NET involves objects. This topic shows you some ways that the elements of objects are used.

Member Access

You access the members of an object by specifying, in order, the name of the object, a period, and the name of the member you want to use. For example, if you have an object named Obj1 that has a property named Prop1 you could use the syntax `Obj1.Prop1` to access that property. In some cases you need to use several dots because objects can contain other objects. For example, you could use the syntax `Obj1.Prop1.X` if the property Prop1 is an object that contains a field named X.

Fields and Properties

Properties and fields represent information stored in an object. The values of fields and properties are retrieved and set with assignment statements the same way that variables are retrieved and set. The following example sets and retrieves the value of a field and a property.

```
Dim X As Integer = 8
Dim Obj1 As New Class1   ' Declares an instance of a class.
Obj1.Prop1 = X   ' Property assignment. Assigns the number 8 to Prop1.
X = Obj1.Prop1 + 1   ' Retrieval. The variable X contains the number 9.
Obj1.Field1 = X    ' Field assignment. Assigns the number 9 to Field1.
X = Obj1.Prop1 + 1   ' Retrieval. The variable X contains the number 10.
```

Methods

Methods are procedures defined within a class. Methods are used like subroutines or functions, but are accessed by way of the object to which they belong. The following code fragment calls a function and a subroutine declared in an object:

```
Dim Obj2 As New Class2   ' Declares an instance of a class
Dim Y As Integer
' Retrieves a value from a method declared as a function.
Y = Obj2.GetNumber
Obj2.StoreNumber(Y)   ' Calls a method declared as a subroutine.
```

Events

Events are notifications that cause a procedure to execute. Objects can both handle and generate events. Events are generated using the **RaiseEvent** statement and handled using either the **Handles** keyword or the **AddHandler** statement. The following example raises and handles an event declared using the **WithEvents** keyword:

```
' Declares a WithEvents variable in the declarations section.
   Dim WithEvents EClass As EventClass

   Class EventClass
      Public Event AnEvent()   ' Declares an event.

      Sub CauseEvent()
         RaiseEvent AnEvent()   'Raises an event.
      End Sub
   End Class

' Declares an event handler
   Sub EventHandler() Handles EClass.AnEvent
      MessageBox.Show("Received Event.")
   End Sub

' Test the class and the event handler.
   Sub TestEvents()
      EClass = New EventClass()   ' Creates an instance of the class.
      EClass.CauseEvent()   ' Calls a method that raises an event.
```

```
    End Sub
```

See Also

What Is an Object? | Creating and Using Objects | Setting and Retrieving Properties | Performing Actions with Methods | Events and Event Handlers | Objects from Visual Basic and Other Sources | RaiseEvent Statement | Handles | AddHandler Statement | WithEvents

Objects, Properties, Methods, and Events

Objects, properties, methods, and events are the basic units of object-oriented programming. An object is an element of an application, representing an *instance* of a class. Properties, methods, and events are the building blocks of objects, and constitute their *members*.

Objects

An object represents an instance of a class such as **Form**, **Control**, or **Component**. In Visual Basic code, you must *instantiate* an object before you can apply one of the object's methods or change the value of one of its properties. Instantiation is the process by which an instance of a class is created and assigned to an object variable. An example is shown below:

```
Dim x As New MyClass()
```

In this example, the variable `x` is assigned to refer to a new instance of the class `MyClass`.

Properties

A property is an attribute of an object that defines one of the object's characteristics, such as size, color, or screen location, or an aspect of its behavior, such as whether it is enabled or visible. To change the characteristics of an object, you change the values of its corresponding properties.

To set the value of a property, affix the reference to an object with a period, the property name, an equal sign (=), and the new property value. For example, the following procedure changes the caption of a Visual Basic Windows Form by setting the **Text** property:

```
Public Sub ChangeName(newTitle)
   myForm.Text = newTitle
End Sub
```

You can't set some properties. The Help topic for each property indicates whether you can set that property (read/write), only read the property (read-only), or only write the property (write-only).

You can retrieve information about an object by returning the value of one of its properties. The following procedure uses a message box to display the title that appears at the top of the currently active form.

```
Public Sub GetFormName()
   Dim formName As String
   formName = myForm.Text
   MsgBox(formName)
End Sub
```

Methods

A method is an action that an object can perform. For example, **Add** is a method of the **ComboBox** object, because it adds a new entry to a combo box.

The following example demonstrates the **Start** method of a **Timer** component:

```
' Instantiates a Timer object.
Dim myTimer As New System.Windows.Forms.Timer()
' Invokes the Start method of myTimer.
MyTimer.Start
```

Events

An event is an action recognized by an object, such as clicking the mouse or pressing a key, and for which you can write code to respond. Events can occur as a result of a user action or program code, or they can be triggered by the system. You can also develop your own custom events to be raised by your objects and handled by other objects. For further information, see Events and Delegates.

See Also

Objects | Setting and Retrieving Properties | Performing Actions with Methods | Events and Delegates

Strings

The **String** data type represents a series of characters (each representing in turn an instance of the **Char** data type). An instance of a string can be assigned to a literal value that represents a series of characters. For example:

```
Dim MyString As String
MyString = "This is an example of the String data type"
```

A **String** variable can also accept any expression that evaluates to a string. Examples are shown below:

```
Dim OneString As String
Dim TwoString As String
OneString = "one, two, three, four, five"
TwoString = OneString.Substring(5,3)  ' Evaluates to "two".
OneString = "1"
TwoString = OneString & "1"  ' Evalates to "11".
```

Any literal that is assigned to a **String** variable must be enclosed by quotation marks (""). Naturally, this makes creating a literal that contains quotation marks somewhat problematic. For example, the following code returns a compiler error:

```
Dim myString As String
myString = "He said, "Look at this example!""  ' This causes an error.
```

This code returns a compiler error because the compiler terminates the string after the second quotation mark, leaving the rest of the string hanging. To have a quotation mark represented in a string literal, you must represent it as two quotation marks. The following example demonstrates the correct way to include a quotation mark in a string:

```
Dim myString As String
' The value of myString is: He said, "Look at this example!"
myString = "He said, ""Look at this example!"""
```

In the above example, the two quotation marks preceding the word 'Look' exist in the string as a single quotation mark. The three quotation marks at the end of the line represent a quotation mark in the string and the string termination character.

A string can be thought of as a series of **Char** instances, and the **String** type has built-in functions that allow you perform many manipulations on a string that resemble the manipulations allowed by arrays. You may refer to a specific character in a string through the **Chars** property, which provides a way to access a character by the position in which it appears in the string. For example:

```
Dim myString As String = "ABCDE"
Dim myChar As Char
myChar = myString.Chars(3)  ' The value of myChar is "C".
```

In the above example, the **Chars** property of the string returns the third character in the string, which is "C," and assigns it to `myChar`. You can also get the length of a particular string through the **Length** property. If greater array-type manipulation of the string is necessary, you can convert it to an array of **Char** instances using the **ToCharArray** function of the string. For example:

```
Dim myString As String = "abcdefghijklmnop"
Dim myArray As Char() = myString.ToCharArray
```

The variable `myArray` now contains an array of **Char** instances, each representing a character from `myString`.

The Immutability of Strings

A string is *immutable*, which means its value cannot be changed once it has been created. How, then, does the following example make any sense?

```
Dim myString As String = "This string is immutable"
myString = "Or is it?"
```

In the above example, it appears that a string is created, given a value, and then has that value changed. This appears to contradict the opening statement that strings are immutable. What is actually happening is that in the first line, an instance of **String** is created and given the value "This string is immutable". In the second line of the example, that *instance* is destroyed and a new instance is created and given the value "Or is it?". The only thing that doesn't change is the name of the variable.

Unlike other intrinsic data types, **String** is a reference type. When a variable of reference type is passed as an argument to a function or subroutine, a reference to the memory address where the data is stored is passed instead of the actual value of the string. So in the previous example, the name of the variable remains the same, but it points to a new and different instance of the **String** class, which holds the new value.

See Also

String Manipulation | String Class | Char Data Type

String Manipulation

The **String** class of the .NET framework provides many built-in methods to facilitate the comparison and manipulation of strings. It is now a trivial matter to get data about a string, or to create new strings by manipulating current strings. The Visual Basic .NET language also has inherent methods that duplicate many of these functionalities.

Types of String Manipulation Methods

In this section you will read about several different ways to analyze and manipulate your strings. Some of the methods are a part of the Visual Basic language, and others are inherent in the **String** class.

Visual Basic .NET methods are used as inherent functions of the language. They may be used without qualification in your code. The following example shows typical use of a Visual Basic .NET string-manipulation command:

```
Dim aString As String = "SomeString"
Dim bString As String
bString = Mid(aString, 3, 3)
```

In this example, the **Mid** function performs a direct operation on `aString` and assigns the value to `bString`.

You can also manipulate strings with the methods of the **String** class. There are two types of methods in **String**: *shared* methods and *instance* methods.

A shared method is a method that stems from the **String** class itself and does not require an instance of that class to work. These methods can be qualified with the name of the class (**String**) rather than with an instance of the **String** class. For example:

```
Dim aString As String
bString = String.Copy("A literal string")
```

In the preceding example, the **String.Copy** method is a static method, which acts upon an expression it is given and assigns the resulting value to `bString`.

Instance methods, by contrast, stem from a particular instance of **String** and must be qualified with the instance name. For example:

```
Dim aString As String = "A String"
Dim bString As String
bString = aString.SubString(2,6) ' bString = "String"
```

In this example, the **SubString** method is a method of the instance of **String** (that is, `aString`). It performs an operation on `aString` and assigns that value to `bString`.

Nothing and Strings

The Visual Basic runtime and the .NET Framework evaluate **Nothing** differently when it comes to strings. Consider the following example:

```
Dim MyString As String = "This is my string"
Dim stringLength As Integer
' Explicitly set the string to Nothing.
MyString = Nothing
' stringLength = 0
stringLength = Len(MyString)
' This line, however, causes an exception to be thrown.
stringLength = MyString.Length
```

The Visual Basic .NET runtime evaluates **Nothing** as an empty string; that is, "". The .NET Framework, however, does not, and will throw an exception whenever an attempt is made to perform a string operation on **Nothing**.

Comparing Strings

You can compare two strings by using the **String.Compare** method. This is a static, overloaded method of the base string class.

In its most common form, this method can be used to directly compare two strings based on their alphabetical sort order. This is similar to the Visual Basic StrComp Function function. The following example illustrates how this method is used:

```
Dim myString As String = "Alphabetical"
Dim secondString As String = "Order"
Dim result As Integer
result = String.Compare (myString, secondString)
```

This method returns an integer that indicates the relationship between the two compared strings based on the sorting order. A positive value for the result indicates that the first string is greater than the second string. A negative result indicates the first string is smaller, and zero indicates equality between the strings. Any string, including an empty string, evaluates to greater than a null reference.

Additional overloads of the **String.Compare** method allow you to indicate whether or not to take case or culture formatting into account, and to compare substrings within the supplied strings. For more information on how to compare strings, see String.Compare Method. Related methods include String.CompareOrdinal Method and String.CompareTo Method.

Searching for Strings Within Your Strings

There are times when it is useful to have data about the characters in your string and the positions of those characters within your string. A string can be thought of as an array of characters (**Char** instances); you can retrieve a particular character by referencing the index of that character through the **Chars** property. For example:

```
Dim myString As String = "ABCDE"
Dim myChar As Char
myChar = myString.Chars(3) ' myChar = "D"
```

You can use the **String.IndexOf** method to return the index where a particular character is encountered, as in the following example:

```
Dim myString As String = "ABCDE"
Dim myInteger As Integer
myInteger = myString.IndexOf("D")  ' myInteger = 3
```

In the previous example, the **IndexOf** method of `myString` was used to return the index corresponding to the first instance of the character "C" in the string. **IndexOf** is an overloaded method, and the other overloads provide methods to search for any of a set of characters, or to search for a string within your string, among others. The Visual Basic .NET command **InStr** also allows you to perform similar functions. For more information of these methods, see String.IndexOf Method and InStr Function. You can also use the String.LastIndexOf Method to search for the last occurrence of a character in your string.

Creating New Strings from Old

When using strings, you may want to modify your strings and create new ones. You may want to do something as simple as convert the entire string to uppercase, or trim off trailing spaces; or you may want to do something more complex, such as extracting a substring from your string. The **System.String** class provides a wide range of options for modifying, manipulating, and making new strings out of your old ones.

To combine multiple strings, you can use the concatenation operators (**&** or **+**). You can also use the String.Concat Method to concatenate a series of strings or strings contained in objects. An example of the **String.Concat** method follows:

```
Dim aString As String = "A"
Dim bString As String = "B"
Dim cString As String = "C"
Dim dString As String = "D"
Dim myString As String
' myString = "ABCD"
myString = String.Concat(aString, bString, cString, dString)
```

You can convert your strings to all uppercase or all lowercase strings using either the Visual Basic .NET functions UCase Function and LCase Function or the String.ToUpper Method and String.ToLower Method methods. An example is shown below:

```
Dim myString As String = "UpPeR oR LoWeR cAsE"
```

```
Dim newString As String
' newString = "UPPER OR LOWER CASE"
newString = UCase(myString)
' newString = "upper or lower case"
newString = LCase(myString)
' newString = "UPPER OR LOWER CASE"
newString = myString.ToUpper
' newString = "upper or lower case"
newString = myString.ToLower
```

The **String.Format** method and the Visual Basic .NET **Format** command can generate a new string by applying formatting to a given string. For information on these commands, see Format Function or String.Format Method.

You may at times need to remove trailing or leading spaces from your string. For instance, you might be parsing a string that had spaces inserted for the purposes of alignment. You can remove these spaces using the String.Trim Method function, or the Visual Basic .NET **Trim** function. An example is shown:

```
Dim spaceString As String = _
"        This string will have the spaces removed        "
Dim oneString As String
Dim twoString As String
' This removes all trailing and leading spaces.
oneString = spaceString.Trim
' This also removes all trailing and leading spaces.
twoString = Trim(spaceString)
```

If you only want to remove trailing spaces, you can use the String.TrimEnd Method or the **RTrim** function, and for leading spaces you can use the String.TrimStart Method or the **LTrim** function. For more details, see LTrim, RTrim, and Trim Functions functions.

The **String.Trim** functions and related functions also allow you to remove instances of a specific character from the ends of your string. The following example trims all leading and trailing instances of the "#" character:

```
Dim myString As String = "#####Remove those!######"
Dim oneString As String
OneString = myString.Trim("#")
```

You can also add leading or trailing characters by using the String.PadLeft Method or the String.PadRight Method.

If you have excess characters within the body of your string, you can excise them by using the String.Remove Method, or you can replace them with another character using the String.Replace Method. For example:

```
Dim aString As String = "This is My Str@o@o@ing"
Dim myString As String
Dim anotherString As String
' myString = "This is My String"
myString = aString.Remove(14, 5)
' anotherString = "This is Another String"
anotherString = myString.Replace("My", "Another")
```

You can use the **String.Replace** method to replace either individual characters or strings of characters. The Visual Basic .NET Mid Statement can also be used to replace an interior string with another string.

You can also use the String.Insert Method to insert a string within another string, as in the following example:

```
Dim aString As String = "This is My Stng"
Dim myString As String
' Results in a value of "This is My String".
myString = aString.Insert(13, "ri")
```

The first parameter that the **String.Insert** method takes is the index of the character the string is to be inserted after, and the second parameter is the string to be inserted.

You can concatenate an array of strings together with a separator string by using the String.Join Method. Here is an example:

```
Dim shoppingItem(2) As String
Dim shoppingList As String
shoppingItem(0) = "Milk"
shoppingItem(1) = "Eggs"
shoppingItem(2) = "Bread"
shoppingList = String.Join(",", shoppingItem)
```

The value of `shoppingList` after running this code is "Milk,Eggs,Bread". Note that if your array has empty members, the method still adds a separator string between all the empty instances in your array.

You can also create an array of strings from a single string by using the String.Split Method. The following example demonstrates the reverse of the previous example: it takes a shopping list and turns it into an array of shopping items. The separator in this case is an instance of the **Char** data type; thus it is appended with the literal type character **c**.

```
Dim shoppingList As String = "Milk,Eggs,Bread"
Dim shoppingItem(2) As String
shoppingItem = shoppingList.Split(","c)
```

The Visual Basic .NET Mid Function can be used to generate substrings of your string. The following example shows this functions in use:

```
Dim aString As String = "Left Center Right"
Dim rString, lString, mString As String
' rString = "Right"
rString = Mid(aString, 13)
' lString = "Left"
lString = Mid(aString, 1, 4)
' mString = "Center"
mString = Mid(aString, 6,6)
```

Substrings of your string can also be generated using the String.Substring Method. This method takes two arguments: the character index where the substring is to start, and the length of the substring. The **String.Substring** method operates much like the **Mid** function. An example is shown below:

```
Dim aString As String = "Left Center Right"
Dim subString As String
' subString = "Center"
subString = aString.SubString(5,6)
```

There is one very important difference between the **String.SubString** method and the **Mid** function. The **Mid** function takes an argument that indicates the character position for the substring to start, starting with position 1. The **String.SubString** method takes an index of the character in the string at which the substring is to start, starting with position 0. Thus, if you have a string "ABCDE", the individual characters are numbered 1,2,3,4,5 for use with the **Mid** function, but 0,1,2,3,4 for use with the **System.String** function.

See Also

Strings | String Class | Mid Function | Mid Statement | Nothing | StrComp Function | InStr Function | UCase Function | LCase Function | Format Function | LTrim, RTrim, and Trim Functions

Constants and Enumerations

Constants are a way to use meaningful names in place of a value that does not change. Constants store values that, as the name implies, remain constant throughout the execution of an application. Using constants allows you to provide meaningful names instead of numbers, making your code more readable.

Enumerations provide a convenient way to work with sets of related constants, and to associate constant values with names. For example, you can declare an enumeration for a set of integer constants associated with the days of the week, and then use the names of the days rather than their integer values in your code.

In This Section

Constants Overview
Describes constants and their uses.
Declaring Constants
Explains how to use the **Const** statement to declare a constant and set its value, thus assigning a meaningful name to the value.
User-Defined Constants
Describes how to create your own constants, including information on scoping and how to avoid circular references.
Constant and Literal Data Types
Provides information on how the Visual Basic compiler initializes constants when **Option Explicit** is turned off.
Enumerations Overview
Describes how to work with sets of related constants and associate constant values with names.
Declaring Enumerations
Explains how to use the **Enum** statement to create an enumeration, and discusses initialization of constant values.
Enumerations and Name Qualification
Discusses how to use the **Imports** statement to get rid of the need to qualify enumeration member names.
Intrinsic Constants and Enumerations
Discusses the constants and enumerations that Visual Basic .NET provides.

Related Sections

Const Statement
Provides syntax and usage rules for the **Const** statement, which is used to declare constants.
Enum Statement
Provides syntax and usage rules for the **Enum** statement, which is used to create enumerations.
Option Explicit Statement
Provides syntax and usage rules for the **Option Explicit** statement, which is used at module level to force explicit declaration of all variables in that module.
Option Strict Statement
Provides syntax and usage rules for the **Option Strict** statement, which restricts implicit data type conversions to widening conversions only.

Constants Overview

Code often contains constant values that reappear. It can also depend on certain numbers that are difficult to remember or have no obvious meaning.

In these cases, you can greatly improve the readability of your code and make it easier to maintain by using constants. A constant is a meaningful name that takes the place of a number or string that does not change. Constants store values that, as the name implies, remain constant throughout the execution of an application.

You declare a constant with the **Const** statement, using the same guidelines you would for creating a variable name. If **Option Strict** is **On**, you must explicitly declare the constant type.

A constant's scope is the same as that of a variable declared in the same location. To create a constant that exists within the scope of a particular procedure, declare it inside that procedure. To create a constant that is available throughout an application, declare it using the **Public** keyword in the declarations section of the class.

> **Note** Although constants somewhat resemble variables, you cannot modify them or assign new values to them as you can to variables.

The constants you use in your code can be intrinsic to the object model for controls or components you work with, or they can be user-defined (that is, those you create yourself).

See Also

User-Defined Constants | Declaring Constants | Constant and Literal Data Types | Enumerations Overview | Declaring Enumerations | Enumerations and Name Qualification | Intrinsic Constants and Enumerations | Choice of Variable Scope

Declaring Constants

You use the **Const** statement to declare a constant and set its value. By declaring a constant, you assign a meaningful name to a value. Once a constant is declared, it cannot be modified or assigned a new value.

You declare a constant within a procedure or in the declarations section of a module, class, or structure. Class or structure-level constants are **Private** by default, but may also be declared as **Public**, **Friend**, **Protected**, or **Protected Friend** for the appropriate level of code access.

The constant must have a valid symbolic name (the rules are the same as those for creating variable names) and an expression composed of numeric or string constants and operators (but no function calls).

To declare a constant

- Write a declaration that includes an access specifier, the **Const** keyword, and an expression, as in the following examples:

```
Public Const DaysInYear = 365
Private Const WorkDays = 250
```

When **Option Strict** is **On**, you must declare a constant explicitly by specifying a data type (**Boolean**, **Byte**, **Char**, **DateTime**, **Decimal**, **Double**, **Integer**, **Long**, **Short**, **Single**, or **String**). When **Option Strict** is **Off**, the constant is assigned a data type by the compiler. See Constant and Literal Data Types for more information.

To declare a constant with Option Strict On

- With **Option Strict On**, write a declaration that includes the **As** keyword and an explicit data type, as in the following examples:

```
Option Strict On
Public Const MyInteger As Integer = 42
Private Const DaysInWeek As Short = 7
Protected Friend Const Funday As String = "Sunday"
```

You can declare multiple constants on a single line, although your code is more readable if you declare only a single constant per line. If you declare multiple constants on a single line, they must all have the same access level (**Public**, **Private**, **Friend**, **Protected**, or **Protected Friend**).

To declare multiple constants on a single line

- Separate the declarations with a comma and a space, as in the following example:

```
Public Const Four As Integer = 4, Five As Integer = 5, Six As Integer = 44
```

See Also

Const Statement | Data Types | Constant and Literal Data Types | Constants and Enumerations | Enumerations Overview | Constants Overview | Declaring Enumerations | Enumerations and Name Qualification | Option Strict Statement | Intrinsic Constants and Enumerations | Choice of Variable Scope

User-Defined Constants

You can use constants that are intrinsic to the object model for controls or components you work with, or you can create your own. Constants you create yourself are described as being *user-defined*.

You declare a constant with the **Const** statement, using the same guidelines you would for creating a variable name. If **Option Strict** is **On**, you must explicitly declare the constant type.

Const Statement Usage

A **Const** statement can represent a mathematical or date/time quantity:

```
Const conPi = 3.14159265358979
Public Const conMaxPlanets As Integer = 9
Const conReleaseDate = #1/1/95#
```

It also can define **String** constants:

```
Public Const conVersion = "07.10.A"
Const conCodeName = "Enigma"
```

The expression on the right side of the equal sign (**=**) is often a number or literal string, but it also can be an expression that results in a number or string (although that expression cannot contain calls to functions). You can even define constants in terms of previously defined constants:

```
Const conPi2 = conPi * 2
```

Scope of User-Defined Constants

A **Const** statement's scope is the same as that of a variable declared in the same location. You can specify scope in any of the following ways:

- To create a constant that exists only within a procedure, declare it within that procedure.
- To create a constant available to all procedures within a class, but not to any code outside that module, declare it in the declarations section of the class.
- To create a constant that is available to all members of an assembly, but not to outside clients of the assembly, declare it using the **Friend** keyword in the declarations section of the class.
- To create a constant available throughout the application, declare it using the **Public** keyword in the declarations section the class.

For more information, see Declaring Constants and Choice of Variable Scope.

Avoiding Circular References

Because constants can be defined in terms of other constants, it is possible to inadvertently create a *cycle*, or circular reference, between two or more constants. A cycle occurs when you have two or more public constants, each of which is defined in terms of the other, as in the following example:

```
Public Const conA = conB * 2
Public Const conB = conA / 2
```

If a cycle occurs, Visual Basic generates a compiler error.

See Also

Const Statement | String Class | Constant and Literal Data Types | Declaring Constants | Option Strict Statement | Constants Overview | Enumerations Overview | Declaring Enumerations | Enumerations and Name Qualification | Intrinsic Constants and Enumerations | Choice of Variable Scope

Constant and Literal Data Types

When **Option Strict** is **On**, you must declare all constants explicitly with a data type. In the following example, the data type of `MyByte` is explicitly declared as data type **Byte**:

```
Option Strict On
Public Const MyByte As Byte = 2
```

Explicitly declaring data types is unambiguous, and typed code is easier to read and maintain than untyped code. When **Option Strict** is **Off**, however, the compiler uses the type of the expression used to initialize the constant. A numeric integer literal is cast by default to the **Integer** data type. The default data type for floating point numbers is **Double**, and the keywords **True** and **False** specify a **Boolean** constant.

Literals and Type Coercion

In some cases, you might want to force a literal to a particular data type; for example, when assigning a particularly large integral literal value to a variable of type **Decimal**. The following example produces an error:

```
Dim myDecimal as Decimal
myDecimal = 10000000000000000000   ' This causes a compiler error.
```

The error results from the representation of the literal. The **Decimal** data type can hold a value this large, but the literal is implicitly represented as a **Long**, which cannot.

You can coerce a literal to a particular data type in two ways: by appending a type character to it, or by placing it within enclosing characters. A type character or enclosing characters must immediately precede and/or follow the literal, with no intervening space or characters of any kind.

To make the previous example work, you can append the D type character to the literal, which causes it to be represented as a **Decimal**:

```
MyDecimal = 10000000000000000000D
```

The following example demonstrates correct usage of type characters and enclosing characters:

```
Option Strict Off
Public Const DefaultInteger = 100   ' Default is Integer.
Public Const DefaultDouble = 54.3345612   ' Default is Double.
Public Const MyCharacter = "a"C   ' Forces constant to be a Char type.
Public Const MyDate = #01/15/01#   ' Demonstrates DateTime constants.
Public Const MyTime = #1:15:59 AM#
Public Const MyLong = 45L   ' Forces data type to be a Long.
Public Const MySingle = 45.55!   ' Forces data type to be a Single.
```

The following table shows the enclosing characters and type characters available in Visual Basic.

Data type	Enclosing character	Appended type characte r
Boolean	(none)	(none)
Byte	(none)	(none)
Char	"	C
Date	#	(none)
Decimal	(none)	D or @
Double	(none)	R or #
Integer	(none)	I or %
Long	(none)	L or &
Short	(none)	S
Single	(none)	F or !
String	"	(none)

See Also

User-Defined Constants | Declaring Constants | Constants Overview | Option Strict Statement | Data Types | Option Explicit Statement | Enumerations Overview | Declaring Enumerations | Enumerations and Name Qualification | Intrinsic Constants and Enumerations

Enumerations Overview

Enumerations provide a convenient way to work with sets of related constants and to associate constant values with names. For example, you can declare an enumeration for a set of integer constants associated with the days of the week, and then use the names of the days rather than their integer values in your code.

You create an enumeration in the declarations section of a class or module, using the **Enum** statement.

An **Enum** type has a name, an underlying type, and a set of fields, each representing a constant. The name must be a valid Visual Basic .NET qualifier. The underlying type must be one of the integer types — **Byte**, **Short**, **Long** or **Integer**. **Integer** is the default. Enumerations are always strongly typed and are not interchangeable with integer number types.

You can assign the same value to multiple fields. When you do so, the compiler marks one of the **Enum** values as primary for purposes of string conversion and reflection. You can use an assignment statement to explicitly assign values to constants in an enumeration. All integer values, including negative numbers, are permissible.

For more information on using the **Enum** statement, see Declaring Enumerations.

See Also

Constants Overview | User-Defined Constants | Declaring Constants | Constant and Literal Data Types | Declaring Enumerations | Enumerations and Name Qualification | Intrinsic Constants and Enumerations

Declaring Enumerations

You create an enumeration with the **Enum** statement in the declarations section of a class or module. You cannot declare an enumeration within a method. To specify the appropriate level of code access, use **Private**, **Protected**, **Friend**, or **Public**.

To declare an enumeration

- Write a declaration that includes an access level, the **Enum** keyword, and a valid name, as in the following examples:

```
Private Enum MyEnum
Public Enum MyEnum
Protected Enum MyEnum
Friend Enum MyEnum
Protected Friend Enum MyEnum
```

By default, the first constant in an enumeration is initialized to 0, and subsequent constants are initialized to a value of one more than the previous constant. For example, the following enumeration, `Days`, contains a constant named `Sunday` with the value 0, a constant named `Monday` with the value 1, a constant named `Tuesday` with the value of 2, and so on:

```
Public Enum Days
   Sunday
   Monday
   Tuesday
   Wednesday
   Thursday
   Friday
   Saturday
End Enum
```

You can explicitly assign values to constants in an enumeration by using an assignment statement. You can assign any integer value, including negative numbers. For example, you may want constants with values less than zero to represent error conditions.

In the following enumeration, the constant `Invalid` is explicitly assigned the value –1, and the constant `Sunday` is assigned the value 0. Because it is the first constant in the enumeration, `Saturday` is also initialized to the value 0. The value of `Monday` is 1 (one more than the value of `Sunday`); the value of `Tuesday` is 2, and so on:

```
Public Enum WorkDays
   Saturday
   Sunday = 0
   Monday
   Tuesday
   Wednesday
   Thursday
   Friday
   Invalid = -1
End Enum
```

By default, enumerations are of the **Integer** type. You can explicitly declare them as **Integer**, **Short**, **Long**, or **Byte**.

To declare an enumeration explicitly

- Write a declaration using the following syntax:

```
Public Enum MyEnum As Byte
   Zero
   One
   Two
End Enum
```

Enumerations cannot have floating-point values. If an enumeration is assigned a floating-point value with **Option Strict On**, a compiler error results. If **Option Strict** is **Off**, the value is automatically converted to the **Enum** type.

For information on names, and how to use the **Imports** statement to make name qualification unnecessary, see Enumerations and Name Qualification.

See Also

Enumerations Overview | Enumerations and Name Qualification | Intrinsic Constants and Enumerations | Constants Overview | User-Defined Constants | Declaring Constants | Constant and Literal Data Types

Enumerations and Name Qualification

Normally, when referring to a member of your enumeration, you must qualify the member name with the enumeration name. For example, to refer to the `Sunday` member of your `Days` enumeration, you would use the following syntax:

```
X = Days.Sunday
```

You can avoid using fully qualified names by adding an **Imports** statement to the namespace declarations section of your code, as in the following example:

```
Imports WindowsApplication1.Form1.Days
```

An **Imports** statement imports namespace names from referenced projects and assemblies, and from within the same project as the module in which the statement appears. Once this statement is added, you can refer to your enumeration members without qualification, as in the following example:

```
X = Sunday
```

By organizing sets of related constants in enumerations, you can use the same constant names in different contexts. For example, you can use the same names for the weekday constants in the `Days` and `WorkDays` enumerations. If you use the **Imports** statement with your enumerations, you must be careful to avoid ambiguous references. Consider the following example:

```
Imports WindowsApplication1.Form1.Days
Imports WindowsApplication1.Form1.WorkDays
Public Sub New()
   ' Insert code to implement constructor.
   ' Ambiguous reference. Does it refer to Days.Monday or WorkDays.Monday?
X = Monday
End Sub
```

Assuming that `Monday` is a member of both the `Days` enumeration and the `Workdays` enumeration, this code generates a compiler error. To avoid ambiguous references when referring to an individual constant, qualify the constant name with its enumeration. The following code refers to the `Saturday` constants in the `Days` and `WorkDays` enumerations.

```
X = Days.Saturday
Y = WorkDays.Saturday
```

See Also

Enumerations Overview | Declaring Enumerations | Constants Overview | User-Defined Constants | Declaring Constants | Constant and Literal Data Types | Intrinsic Constants and Enumerations | Enum Statement | Data Types | Imports Statement

Intrinsic Constants and Enumerations

Visual Basic .NET provides a number of intrinsic constants and enumerations to facilitate your programming.

Constants store values that remain constant throughout the execution of an application. They are meaningful names that take the place of a number or string and make code more readable. Enumerations offer an easy way to work with sets of related constants. An enumeration, or Enum, is a symbolic name for a set of values.

In the following enumeration, `SchoolDays`, the constant `NotValid` is assigned the value of -1, and the constant `Sunday` is assigned the value of 0. Because it is the first constant, `Saturday` is also initialized to a value of 0. The value of `Monday` is 1 (one more than the value of `Sunday`, the immediately preceding value); the value of `Tuesday` is 2 (one more than `Monday`, the immediately preceding value).

```
Public Enum SchoolDays
   Saturday
   Sunday = 0
   Monday
   Tuesday
   Wednesday
   Thursday
   Friday
   NotValid = -1
End Enum
```

You can create your own constants and enumerations, as discussed in User-Defined Constants, or you can use the intrinsic constants and enumerations Visual Basic .NET provides, which are listed in the Object Browser. The following table lists the intrinsic enumerations available in Visual Basic .NET.

Enumeration	**Used with**
AppWinStyle	Shell commands
CallType	**CallByName** function
CompareMethod	Comparison functions
DateFormat	DateValue function
FileAttribute	**Dir**, **GetAttr** and **SetAttr** functions
FirstDayOfWeek	Date related functions
FirstWeekOfYear	Date related functions
MsgBoxResult	**MsgBox** function
MsgBoxStyle	**MsgBox** function
Tristate	Number-formatting functions
VbStrConv	**StrConv** function

See Also

Constants and Enumerations | Constants Overview | Declaring Constants | Constant and Literal Data Types | Enumerations Overview | User-Defined Constants | Enum Statement | Object Browser

Operators

An operator is a code unit that performs an operation on one or more value-returning code elements. Such an operation can be an arithmetic operation such as addition or multiplication; a concatenation operation that combines two strings into a new string; a comparison operation that determines which of two values is greater; a logical operation evaluating whether two expressions are both true. Operators combined with value-returning code elements such as literals or variables form expressions, or in the case of the assignment operator, statements.

The following is an example of a statement that uses the simple assignment operator (**=**):

```
x = 45
```

In the preceding example, the assignment operator performs the assignment operation: it takes the value on the right side of the operator and assigns it to the variable on the left. Several operators can perform actions in a single expression or statement, as demonstrated below:

```
x = 45 + y * z ^ 2
```

In this example, the operations in the expression on the right side of the assignment operator are performed, and the resulting value is assigned to the variable `x`, on the left. There is no limit to the number of operators that can be combined into an expression, but an understanding of Operator Precedence in Visual Basic is necessary to ensure that you get the results you expect. For more information, see Efficient Combination of Operators.

In This Section

Arithmetic Operators
: Discusses operators used to perform many familiar arithmetic operations that involve calculation of numeric values represented by literals, variables, other expressions, function and property calls, and constants.

Comparison Operators
: Discusses operators that compare two expressions and return a **Boolean** value representing the result of the comparison.

Concatenation Operators
: Provides information on the **+** and **&** operators, which join multiple strings into a single string.

Logical Operators
: Provides information on operators that compare **Boolean** expressions and return a **Boolean** result.

Efficient Combination of Operators
: Discusses how to create complex numeric, **Boolean**, and parenthetical expressions, assuming knowledge of operator precedence.

Related Sections

Expressions
: Shows how to combine elements with operators to yield new values.

Statements
: Provides an overview of the two basic units of instruction in Visual Basic: declaration statements and executable statements.

Operators
: Provides links to Language Reference topics covering all Visual Basic operators, listed by category.

Arithmetic Operators

Arithmetic operators are used to perform many of the familiar arithmetic operations that involve the calculation of numeric values represented by literals, variables, other expressions, function and property calls, and constants.

Arithmetic Operations

You can add two values in an expression together with the **+** operator, or subtract one from another with the **-** operator, as shown in the following examples:

```
Dim x As Integer
x = 67 + 34
x = 32 - 12
```

Negation also uses the **-** operator, but with somewhat different syntax, as shown below:

```
Dim x As Integer = 65
Dim y As Integer
y = -x
```

Multiplication and division use the ***** and **/** operators, respectively, as in these examples:

```
Dim y As Double
y = 45 * 55.23
y = 32 / 23
```

Exponentiation uses the **^** operator, as shown below:

```
Dim z As Double
z = 23 ^ 3    ' z equals the cube of 23.
```

Integer division is carried out using the **** operator. Integer division returns the number of times one integer can evenly divide into another. Only integral types (**Byte**, **Short**, **Integer**, and **Long**) can be used with this operator. All other types must be converted to an integral type first. The following is an example of integer division:

```
Dim k As Integer
k = 23 \ 5      ' k = 4
```

Modulus arithmetic is performed using the **Mod** operator. This operator returns the remainder after the divisor is divided into the dividend an integral number of times. If both divisor and dividend are integral types, the returned value is integral. If divisor and dividend are floating-point types, the returned value is a floating point variable. Two examples illustrate this behavior:

```
Dim x As integer = 100
Dim y As Integer = 6
Dim z As Integer
z = x Mod y   ' z equals 4.
Dim a As Double = 100.3
Dim b As Double = 4.13
Dim c As Double
c = a Mod b   ' c equals 1.18.
```

Division by Zero

Division by zero has different results depending on the data types involved. In integral divisions (**Short**, **Integer**, **Long**), a **System.DivideByZeroException** is thrown. In division operations on the **Decimal** or **Single** data type, Visual Basic also throws a **System.DivideByZeroException** exception. In floating-point divisions with the **Double** variable type, however, the result is the class member representing **NaN**, **PositiveInfinity**, or **NegativeInfinity**, depending on the dividend. The following table summarizes the various results of attempting to divide by zero.

Dividend data type	Divisor data type	Dividend value	Result
Double	**Double**	0	**System.Double.NaN** (not a number)
Double	**Double**	> 0	**System.Double.PositiveInfinity**
Double	**Double**	< 0	**System.Double.NegativeInfinity**

When you catch a **System.DivideByZeroException** exception, you can use its members to help you handle it. The **HelpLink** property has a link to the appropriate help file, the **Message** property holds the message text for the exception, and the **StackTrace** property contains the trace, which should indicate where the error occurred. The **GetBaseException** method returns the original exception that caused this **DivideByZeroException**, as well as any other exceptions linked to it.

Bitwise Operations

Bitwise operations evaluating two values in binary (base 2) form, compare the bits at corresponding positions, and then assign values based on the comparison, as in the demonstration below of bitwise **And**:

```
Dim x As Integer
x = 3 And 5   ' x = 1
```

In this example, the value of `x` is `1`. Here's why:

- First, the values are converted to binary form, as shown below:

  3 in binary form = 011

  5 in binary form = 101

- Next, the binary representations are compared, one binary position at a time. If both bits at a given position are 1, then a 1 is placed in that position in the result. If either bit is 0, then a 0 is placed in that position in the result. In our example this works out as follows:

  011 3 in binary form

  101 5 in binary form

  001 The result, in binary form

  Converting back to decimal, 001 is the binary representation of 1, so x = 1.

The bitwise **Or** operation is similar, except that a 1 is assigned to the result bit if either of the compared bits is 1. **Xor** assigns a 1 to the result bit if only one of, but not both of, the compared bits is 1. **Not** takes a single operand and inverts all the bits, including the sign bit, and assigns that value to the result. This means that for positive numbers, **Not** always returns a negative. Likewise, for negative numbers, **Not** always returns a positive or zero.

> **Note** Bitwise operations can be performed on integral types only. Floating point values must be converted to integral types before bitwise operation can proceed.

Bit Shift Operations

A bit shift operation performs an arithmetic shift on a bit pattern. The pattern is contained in the operand on the left, while the operand on the right specifies the number of positions to shift the pattern. The data type of the pattern operand must be **Byte**, **Short**, **Integer**, or **Long**. The data type of the shift amount operand must be **Integer** or must widen to **Integer**.

Arithmetic shifts are not circular, which means the bits shifted off one end of the result are not reintroduced at the other end. The bit positions vacated by a shift are set as follows:

- Zero for an arithmetic left shift
- Zero for an arithmetic right shift of a positive number
- Zero for an arithmetic right shift of any pattern with data type **Byte**
- One for an arithmetic right shift of a negative number (**Short**, **Integer**, or **Long**)

In the following example, an **Integer** value is shifted both left and right:

```
Dim Pattern As Integer = 12   ' Low-order bits are 0000 1100
Dim LResult, RResult As Integer
LResult = Pattern << 3   ' Left shift of 3 bits produces value of 96
RResult = Pattern >> 2   ' Right shift of 2 bits produces value of 3
```

Arithmetic shifts never generate overflow exceptions.

Type Safety

Operands should normally be of the same type. For example, if you are adding an **Integer** variable, you should add it to another **Integer** variable, and the variable that the value is assigned to should be of type **Integer** as well.

One way to ensure good type-safe coding practice is to use **Option Strict**. If **Option Strict** is **On,** Visual Basic .NET automatically performs type-safe conversions. Thus, if you try to add an **Integer** variable to a **Double** variable and assign the value to a **Double** variable, the operation proceeds normally. Unsafe, or narrowing conversions, on the other hand, cause a compiler error. Thus if you try to add an **Integer** variable to a **Double** variable and assign the value to an **Integer** variable, a compiler error results, because a **Double** variable cannot be implicitly converted to type **Integer**.

If **Option Strict** is **Off**, however, these implicit narrowing conversions are allowed to take place, but can result in the unexpected loss of data or precision. For this reason, it is recommended that you use **Option Strict On** when writing production code. For additional information about conversions, see Widening and Narrowing Conversions, and for information about **Option Strict**, see Option Strict Statement.

See Also

DivideByZeroException Class | Widening and Narrowing Conversions | Option Strict Statement | Arithmetic Operators | Bit Shift Operators

Comparison Operators

Comparison operators compare two expressions and return a **Boolean** value that represents the result of the comparison. There are operators for comparing numeric values, operators for comparing strings, and operators for comparing objects. All three types of operators are discussed below.

Comparing Numeric Values

Numeric values are compared using six numeric comparison operators. The following table lists the operators and gives an example of the condition tested by the operator.

Operator	Condition tested	Examples

= (Equality)	Is the value represented by the first expression equal to the value represented by the second?	`23 = 33  ' False`
		`23 = 23  ' True`
		`23 = 12 ' False`

<> (Inequality)	Is the value represented by the first expression not equal to the value represented by the second?	`23 <> 33  ' True`
		`23 <> 23  ' False`
		`23 <> 12 ' True`

< (Less than)	Is the value represented by the first expression less than the value represented by the second?	`23 < 33  ' True`
		`23 < 23  ' False`
		`23 < 12 ' False`

> (Greater than)	Is the value represented by the first expression greater than the value represented by the second?	`23 > 33  ' False`
		`23 > 23  ' False`
		`23 > 12 ' True`

<= (Less than or equal to)	Is the value represented by the first expression less than or equal to the value represented by the second?	`23 <= 33  ' True`
		`23 <= 23  ' True`
		`23 <= 12 ' False`

>= (Greater than or equal to)	Is the value represented by the first expression greater than or equal to the value represented by the second?	`23 >= 33  ' False`
		`23 >= 23  ' True`
		`23 >= 12 ' True`

Comparing Strings

Strings can be compared using the **Like** operator as well as using the numeric comparison operators. The **Like** operator allows you to specify a pattern; the string is then compared against the pattern, and if it matches, the result is **True**. Otherwise, the result is **False**. The numeric operators allow you to compare **String** values based on their sort order, as in this example:

```
"73" < "9"    ' This result is True.
```

The result is **True** because the first character in the first string sorts before the first character in the second string. If the first characters were equal, the comparison would continue to the next character in both strings, and so on. Equality of strings can also be tested using the equality operator, as in this example:

```
"734" = "734"      ' The result is True.
```

In this example, the two strings are directly compared, and, because they are equal, **True** is returned.

If one string is a prefix of another, such as "aa" and "aaa", the longer string sorts after the shorter. Thus the following expression is true:

```
"aaa" > "aa"
```

The sort order will be based on either a binary comparison or a textual comparison depending on the Option Compare setting. For more information see Option Compare Statement.

Comparing Objects

You can use the **Is** operator to determine if two object variables refer to the same instance of an object. For example:

```
Dim x As MyClass
Dim y As New MyClass()
x = y
If x Is y Then
   ' Insert code to continue program execution.
```

In this example, `x Is y` evaluates to **True**, because both variables refer to the same instance. Contrast this result with the following example:

```
Dim x As New MyClass()
Dim y As New MyClass()
If x Is y then
   ' Insert code to continue program execution.
```

In this example, `x Is y` evaluates to **False**, because although the variables are of the same type, they refer to different instances of that type.

You can test whether an object is of a particular type with the **TypeOf...Is** operator. . The syntax is

```
    TypeOf <object expression> Is <TypeName>
```

When TypeName specifies an interface type then the TypeOf ... Is operator returns true if the object implements the interface type. When the TypeName is a class type then the operator returns true if the object is an instance of the specified class or is an instance of a class that derives from the specified class. For example:

```
Dim x As Button
x = New Button()
If TypeOf x Is Control Then
   ' Insert code to continue program execution.
```

Since the type of `x` is `Button`, and `Button` inherits from `Control`, the `TypeOf x Is Control` expression evaluates to **True**.

See Also

Value Comparisons | Comparison Operators | Operators

Concatenation Operators

Concatenation operators join multiple strings into a single string. There are two concatenation operators, **+** and **&**; both carry out the basic concatenation operation, as shown below:

```
Dim x As String
x = "Con" & "caten" & "ation"   ' x equals "Concatenation".
x = "Con" + "caten" + "ation"   ' x equals "Concatenation".
```

These operators can also concatenate **String** variables, as in the following example:

```
Dim x As String = "abc"
Dim y As String = "def"
Dim z As String
z = x & y   ' z equals "abcdef".
z = x + y   ' z equals "abcdef".
```

If **Option Strict** is **On**, the **+** operator explicitly requires strings for concatenation operations. It also carries out arithmetic operations with numeric operands when **Option Strict** is **On**. If **Option Strict** is **Off**, however, the **+** operator supports a wide variety of implicit conversions for concatenation operations. For further information on these conversions, see + Operator.

The **&** operator is defined only for **String** operands, and it always widens its operands to **String**, regardless of the setting of **Option Strict**. The **&** operator is recommended for string concatenation because it is defined exclusively for strings and reduces your chances of generating an unintended conversion.

See Also

& Operator | + Operator | Option Strict Statement | String Manipulation

Logical Operators

The logical operators compare **Boolean** expressions and return a **Boolean** result. The **And**, **Or**, **AndAlso**, **OrElse**, and **Xor** operators take two operands, and the **Not** operator takes a single operand.

The **Not** operator performs logical negation on a **Boolean** expression. Put simply, it yields the opposite of the expression it evaluates. If the expression evaluates to **True**, **Not** yields **False**; if the expression evaluates to **False**, **Not** yields **True**. An example is shown below:

```
Dim x As Boolean
x = Not 23 > 12   ' x equals False.
x = Not 23 > 67   ' x equals True.
```

The **And** operator performs logical conjunction on two **Boolean** expressions. That is, if both expressions evaluate to **True**, then the **And** operator returns **True**. If either or both expressions evaluate to **False**, then **And** returns **False**.

The **Or** operator performs logical disjunction on two **Boolean** expressions. If either expression evaluates to **True**, **Or** returns **True**. If neither expression evaluates to **True**, **Or** returns **False**.

Xor performs logical exclusion on two expressions. If either expression evaluates to **True**, but not both, **Xor** returns **True**. If both expressions evaluate to **True** or both expressions evaluate to **False**, **Xor** returns **False**.

Examples of the **And**, **Or**, and **Xor** operators are shown below:

```
Dim x As Boolean
x = 23 > 12 And 12 >4   ' x = True
x = 12 > 23 And 12 > 4   ' x = False
x = 23 > 12 Or 4 > 12`   ' x = True
x = 23 > 45 Or 4 > 12   ' x = False
x = 23 > 45 Xor 12 > 4   ' x = True
x = 23 > 12 Xor 12 > 4   ' x = False
x = 12 > 23 Xor 4 > 12   ' x = False
```

> **Note** In addition to being logical operators, **Not**, **Or**, **And**, and **Xor** also perform bitwise arithmetic when used on numeric values. Information on this functionality can be found at Arithmetic Operators.

The **AndAlso** operator is very similar to the **And** operator, in that it also performs logical conjunction on two **Boolean** expressions. The key difference between **AndAlso** and **And** is that **AndAlso** exhibits short-circuiting behavior. If the first expression in an **AndAlso** expression evaluates to **False**, then the second expression is not evaluated and **False** is returned for the **AndAlso** expression.

Similarly, the **OrElse** operator performs short-circuiting logical disjunction on two **Boolean** expressions. If the first expression in an **OrElse** expression evaluates to **True**, then the second expression is not evaluated and **True** is returned for the **OrElse** expression. Below are some examples that illustrate the difference between **And**, **Or**, and their counterparts:

```
12 > 45 And MyFunction(4) ' MyFunction() is called.
12 > 45 AndAlso MyFunction(4)   ' MyFunction() is not called.
45 > 12 Or MyFunction(4)   ' MyFunction is called.
45 > 12 OrElse MyFunction(4)   ' MyFunction is not called
```

In the first example, `MyFunction()` is called, even though `12 > 45` returns **False**, because **And** does not short circuit. In the second example, `MyFunction` is not called, because `12 > 45` returns **False**, so **AndAlso** short circuits the second expression. In the third example, `MyFunction` is called, even though `45 > 12` returns **True**, because **Or** does not short circuit. In the fourth example, `MyFunction` is not called because `45 > 12` returns **True**, so **OrElse** short circuits the second expression.

> **Note** Although **And** and **Or** support bitwise operations with numeric values, **AndAlso** and **OrElse** do not.

See Also

Logical/Bitwise Operators | Arithmetic Operators | Boolean Expressions | Efficient Combination of Operators

Efficient Combination of Operators

Complex expressions can contain many different operators. Consider the following statement:

```
x = (45 * (y + z)) ^ (2 / 85) * 5 + z
```

Creating complex expressions such as these requires a thorough understanding of the rules of operator precedence. This topic, which will help you construct efficient expressions, assumes that you are familiar with the rules of operator precedence. For more information, see Operator Precedence in Visual Basic.

Numeric Expressions

In numeric expressions such as the one in the above example, operators follow a strict precedence pattern. First, any variables or properties are read, and any value-returning function calls are executed.

Once these values are retrieved, the expression is evaluated in accordance with the rules of operator precedence. Exponentiation (**^**) occurs first, then negation (**-**). These are followed by multiplication (*****) and division (**/**), which have equal operator precedence and therefore occur in textual order from left to right. Integer division (****) occurs next, followed by modulus arithmetic (**Mod**). Addition (**+**) and subtraction (**-**) take place next; Logical/Bitwise Operators evaluate last. When operators of equal precedence occur in an expression, evaluation is in the order of operator occurrence, from left to right.

Boolean Expressions

Boolean expressions can be even more complex than numeric expressions, since they can combine arithmetic operators, comparison operators, and logical operators. The following is an example of an expression that contains all three kinds:

```
x + y > z And z + y < w
```

In this example, arithmetic operations are performed by the `+` operator, comparison operations are performed by the `>` and `<` operators, and a logical operation is performed by the `And` operator. When evaluating an expression with different kinds of operators, standard operator precedence usually determines the order of operation. For details, see Operator Precedence in Visual Basic.

Parenthetical Expressions

Often you want the execution of operations to proceed in a different order from that determined by operator precedence. Consider this example:

```
x = z * y + 4
```

In this example, `z` is multiplied by `y`, and then the result is added to `4`. But what if you want to add `y` and `4` before multiplying by `z`? In such a case you can override normal operator precedence by using parentheses. By enclosing an expression in parentheses, you force that expression to evaluate first, regardless of operator precedence.

So, to force the above expression to evaluate addition first, you would rewrite the example as follows:

```
x = z * (y + 4)
```

The value of `y` is now added to `4`, and that sum is multiplied by `z`.

You can nest expressions in multiple levels of parentheses to enforce precedence further. The expressions most deeply nested in parentheses are granted primary precedence, followed by less deeply nested expressions, and so on. Consider the following example:

```
x = (z * 4) ^ (y * (z + 2))
```

Here, `z + 2` has primary precedence, then the other parenthetical expressions. Exponentiation, which normally has higher precedence than addition or multiplication, evaluates last in this example because the other expressions are enclosed in parentheses.

See Also

Arithmetic Operators | Comparison Operators| Logical Operators | Logical/Bitwise Operators | Boolean Expressions | Value Comparisons | Calculating Numeric Values | Operator Precedence in Visual Basic

Expressions

An expression can be thought of as a series of value-representing elements, separated by operators, that yields a new value. Examples of such value-representing elements are literals, variables, other expressions, and function calls. The operators act upon the value-representing elements by performing calculations, comparisons, or other operations. The end result of the expression represents a value, which can be either numeric, **Boolean**, or **String**.

The following are examples of expressions:

```
5 + 4   ' Evaluates to 9.
15 * System.Math.Sqrt(9) + x    ' Evaluates to 45 plus the value of x.
"Concat" & "ena" + "tion"   ' Evaluates to "Concatenation".
763 < 23   ' Evaluates to False.
x Is y   ' Value depends on the objects represented by x and y.
```

In This Section

Calculating Numeric Values
: Describes how to use numeric expressions, which contain literals and variables representing numeric values, as well as operators that act upon those values.

Value Comparisons
: Demonstrates how to compare the values of numeric variables.

Boolean Expressions
: Discusses expressions that evaluate to the **Boolean** values **True** or **False**. Also covers short-circuiting operators and parenthetical expressions.

Related Sections

Assignment Operators
: Provides links to Language Reference topics covering assignment operators such as **=**, ***=**, and **&=**.

Operators
: Provides links to Language Reference topics covering all Visual Basic operators, listed by category.

Calculating Numeric Values

Numeric values can be calculated through the use of numeric expressions. A numeric expression is an expression that contains literals and variables representing numeric values, and operators that act upon those values. The following is an example of a numeric expression:

```
4 * (67 + x) ' For the purposes of this example, x = 2.
```

This expression evaluates to 276, which is the value of 67 plus 2, multiplied by 4.

If the numeric expression contains more than one operator, the order in which they are evaluated is determined by the rules of operator precedence. To override the rules of operator precedence, you enclose expressions in parentheses, as in the above example; such expressions are evaluated first. For more information, see Operator Precedence in Visual Basic, and Efficient Combination of Operators

You can use an assignment statement to assign the value represented by this expression to another variable, as in this example:

```
Dim x As Integer = 2
Dim y As Integer
y = 4 * (67 + x)
```

Here, the value of the expression on the right side of the operator is assigned to the variable `y` on the left side of the operator, so `y` evaluates to 276. For more information on assignment, see Assignment Statements.

See Also

Expressions | Value Comparisons | Assignment Statements | Operator Precedence in Visual Basic | Arithmetic Operators | Efficient Combination of Operators

Value Comparisons

Comparison operators can be used to construct expressions that compare the values of numeric variables. These expressions return a **Boolean** value based on whether the comparison is true or false. Examples of such an expression would be:

```
45 > 26
26 > 45
```

The first expression evaluates to **True**, because 45 is greater than 26. The second example evaluates to **False**, because 26 is not greater than 45.

You can compare any two numeric expressions in this fashion. The expressions you compare can themselves be complex expressions, as in this example:

```
x / 45 * (y +17) >= System.Math.Sqrt(z) / (p - (x * 16))
```

This complex expression includes literals, variables, and function calls. The expressions on both sides of the comparison operator are evaluated, and the resulting values are then compared using the **>=** comparison operator. If the value of the expression on the left side is greater than or equal to the value of the expression on the right, the entire expression evaluates to **True**; otherwise, it evaluates to **False**.

Expressions that compare values are most commonly used in **If...Then** statements, as in the following example:

```
If x > 50 Then
   ' Insert code to be executed if x is greater than 50.
Else
   ' Insert code to be executed if x is less than or equal to 50.
End If
```

The **=** sign is a comparison operator as well as an assignment operator. When used as a comparison operator, it evaluates whether the value on the left is equal to the value on the right, as shown in the following example:

```
If x = 50 Then
   ' Insert code to continue program.
End If
```

You can also use a comparison expression anywhere that a **Boolean** value is needed such as in an **If**, **While**, **Loop**, **ElseIf** statement or when assigning to or passing a value to a Boolean variable. In the following example, the value returned by the comparison expression is assigned to a **Boolean** variable:

```
Dim x As Boolean
x = 50 < 30   ' x is assigned False.
```

See Also

Boolean Expressions | Operators | Comparison Operators | Expressions | Calculating Numeric Values | Operator Precedence in Visual Basic

Boolean Expressions

A **Boolean** expression is an expression that evaluates to a **Boolean** value. **Boolean** expressions can take several forms. The simplest is the direct comparison of the value of a **Boolean** variable to a **Boolean** literal, as shown in this example:

```
If x = True Then   ' Compares x to the Boolean value True.
   ' Insert code to execute if x = True.
Else
   ' Insert code to execute if x = False.
End If
```

Note that the assignment statement `x = True` looks the same as the expression in the example, but it performs a different function and is used differently. In the above example, the expression `x = True` represents a value, and thus the = sign is interpreted as a comparison operator. In a stand-alone statement, it would be interpreted as an assignment operator and assign the value on the right to the variable on the left. The following example illustrates this:

```
If x = True then   ' Compare the value of x to Boolean value True.
   y = False      ' Assign the Boolean value True to variable y.
End If
```

For further information, see Value Comparisons and Assignment Statements.

Comparison operators, such as =, **<**, **>**, **<>**, **<=**, and **>=**, produce **Boolean** expressions by comparing the expression on the left side of the operator to the expression on the right side of the operator and evaluating the result as true or false. For instance:

```
42 < 81   ' Evaluates to True.
```

For more information on this kind of expression, see Value Comparisons.

Comparison expressions can be combined using logical operators to produce more complex **Boolean** expressions. The following example demonstrates the use of comparison operators in conjunction with a logical operator:

```
x > y And x < 1000
```

In the example, the value of the overall expression depends on the values of the expressions on each side of the **And** operator. If both expressions are true, then the overall expression evaluates as **True**. If either expression is false, then the entire expression evaluates to **False**.

Short-Circuiting Operators

The logical operators **AndAlso** and **OrElse** exhibit behavior known as *short-circuiting*. A short-circuiting operator evaluates the left expression first. If the left expression falsifies (in the case of **AndAlso**) or validates (in the case of **OrElse**) the entire expression, then program execution proceeds without evaluating the right expression. Take the following example:

```
If 45 < 12 AndAlso MyFunction(3) = 81 Then
   ' Add code to continue execution.
```

In this example, the operator evaluates the left expression, `45 < 12`. Since this expression evaluates to **False**, the entire logical expression evaluates to **False**. Program execution thus begins execution of the code within the If block without evaluating the right expression, `MyFunction(3)`. In this example, `MyFunction()` is never called because the left expression falsifies the entire expression.

Similarly, if the left expression in a logical expression using **OrElse** evaluates to **True**, execution proceeds to the next line of code without evaluating the second expression because the first expression has already enabled the entire expression.

By contrast, both sides of the logical operator are evaluated when the logical operators **And** and **Or** are used. Consider the following expression for example:

```
If 45 < 12 And MyFunction(3) = 81 Then
   ' Add code to continue execution.
```

In this example, a call is made to `MyFunction()`, even though the first expression evaluates to **False**.

Parenthetical Expressions

You can use parentheses to control the evaluation of **Boolean** expressions. Expressions enclosed by parentheses evaluate first. For multiple levels of nesting, precedence is granted to the most deeply-nested expressions. Within parentheses, evaluation proceeds according to the rules of operator precedence. For more information, see Operator Precedence in Visual Basic and Efficient Combination of Operators.

See Also

Logical Operators | Value Comparisons | Assignment Statements | Comparison Operators | Efficient Combination of Operators | Operator Precedence in Visual Basic

Statements

A *statement* in Visual Basic is a complete instruction that can contain keywords, operators, variables, constants, and expressions. All statements fall into one of two categories: *declaration statements*, which name a variable, constant, or procedure and can also specify a data type; or *executable statements*, which initiate actions.

In This Section

Statements Overview
: Provides a high-level look at statements, including how to have multiple statements on one line, continue a statement over multiple lines, add comments, and check compilation errors.

Assignment Statements
: Describes statements that carry out assignment operations, which assign the value on the right side of the operator to the variable on the left side.

Declaration Statements
: Describes statements that name and define procedures, variables, arrays, and constants.

Executable Statements
: Describes statements that initiate action by executing a method. Executable statements can loop or branch through blocks of code, and often contain mathematical or conditional operators.

Related Sections

Assignment Operators
: Provides links to Language Reference topics covering assignment operators such as **=**, ***=**, and **&=**.

Expressions
: Shows how to combine elements with operators to yield new values.

Operators
: Describes the code units that manipulate value-returning elements, and how to use them efficiently.

Statements Overview

A statement in Visual Basic is a complete instruction. It can contain keywords, operators, variables, constants, and expressions. Each statement belongs to one of the following two categories:

- Declaration statements, which name a variable, constant, or procedure and can also specify a data type. For more information, see Declaration Statements.
- Executable statements, which initiate actions. These statements can execute a method or function, and they can loop or branch through blocks of code. Executable statements include assignment statements, which assign a value or expression to a variable or constant. For more information, see Executable Statements and Assignment Statements.
- Declaration statements, which name a variable, constant, or procedure and can also specify a data type. For more information, see Declaration Statements.
- Executable statements, which initiate actions. These statements can execute a method or function, and they can loop or branch through blocks of code. Executable statements include assignment statements, which assign a value or expression to a variable or constant. For more information, see Executable Statements and Assignment Statements

Having Multiple Statements on One Line

You can have multiple statements on a single line separated by the colon (:) character. For example:

```
Dim MyString As String = "Hello World" : MsgBox(MyString)
```

Though occasionally convenient, this form of syntax makes your code hard to read and maintain. Thus, it is recommended that you keep one statement to a line.

Continuing a Statement over Multiple Lines

A statement usually fits on one line, but when it doesn't, you can continue a long statement onto the next line using a line-continuation character, which consists of an underscore character (_) followed by a carriage return. In the following example, the **MsgBox** executable statement is continued over two lines:

```
Public Sub DemoBox()   'This procedure displays a message.
   Dim myVar As String
   myVar = "John"
   MsgBox("Hello " & myVar & _
      ". How are you?")
End Sub
```

Adding Comments

Source code is not always self-explanatory, even to the programmer who wrote it. To help document their code, therefore, most programmers make liberal use of embedded comments. Comments in code can explain a procedure or a particular instruction to anyone reading or working with it later. Visual Basic ignores comments, however, when it runs your procedures.

Comment lines begin with an apostrophe (') or **REM** followed by a space, and can be added anywhere in code. To append a comment to a statement, insert an apostrophe or **REM** after the statement, followed by the comment. Such a comment appears in the preceding example. Comments can also go on their own separate line.

Checking Compilation Errors

If, after you type a line of code, the line is displayed with a wavy blue underline (an error message may display as well), there is a syntax error in the statement. You must find out what's wrong with your statement (by looking in the task list, or hovering over the error and reading the help text, for example) and correct it. Until you have fixed all syntax errors in your code, your program will fail to compile correctly.

See Also

Assignment Statements | Declaration Statements | Executable Statements | Labeling Statements

Assignment Statements

Assignment statements carry out assignment operations. Simple assignment operations consist of taking the value on the right side of the operator and assigning it to the variable on the left, as in this example:

```
x = 42
```

Here, the variable `x` is assigned the literal value 42.

The value on the right side of the operator can be any literal, variable, expression, or function call that returns a value, as in this example:

```
x = y + z + MyFunction(3)
```

Here, the value represented by `y` is added to the value represented by `z`, which is added to the value returned by the call to `MyFunction(3)`. The total value of this expression is then assigned to the variable `x`.

The assignment operator can also assign values to **String** variables and **String** expressions, as in this example:

```
Dim x As String
x = "String variable assignment"
x = "Con" & "cat" & "enation" ' x equals "Concatenation".
```

The values of **Boolean** variables can be assigned as well, using either a **Boolean** literal or a **Boolean** expression as the right argument:

```
Dim x As Boolean
x = True
x = 45 > 1003 ' x equals False.
x = 45 > 1003 Or 45 > 17 ' x equals True.
```

The argument on the left side of the operator can be any valid variable or property. The following example demonstrates setting the value of the text property of a text box:

```
MyTextBox.Text = "This " & "is a " & "String"
   ' MyTextBox.Text = "This is a String".
```

Other assignment statements first perform an operation on an argument before assigning it to another argument. The following example illustrates one of these operators, **+=**, which increments the value of the variable on the left side of the operator by the value of the expression on the right:

```
x += 1
```

In this example, 1 is added to the value of `x`, and that new value is then assigned to `x`. It is a shorthand equivalent of the following statement:

```
x = x + 1
```

A variety of compound assignment operations can be performed using operators of this type. If the value of the variable on the left side of the operator has not been set, it is treated as zero. For a list of these operators and more information about them, see Assignment Operators in the Visual Basic Language Reference.

Lastly, the concatenation assignment operator is useful for adding a string to the end of already existing strings, as illustrated in this example:

```
Dim x As String = "My "
x &= "String"    ' x equals "My String".
```

If the variable on the left side of the concatenation assignment operator has not been assigned a value, it is treated as an empty string ("").

Widening and Narrowing Conversions

When assigning values, you must always assign a value to a variable of the appropriate type. Some numeric types can be converted to other numeric types. If **Option Strict** is **On**, all of the literals and variables in a numeric expression must either be the same data type (for example, **Integer**), or be types that are implicitly convertible to the widest (most precise) type. Visual Basic .NET implicitly converts Byte, Short, and Integer types as Long when they are assigned to a Long variable. This is called a widening conversion. Numeric type conversions in the following direction are widening: **Byte**, **Short**, **Integer**, **Long**, **Decimal**, **Single**, **Double**. An example of widening conversion is shown below:

```
Dim x As Double
Dim y As Integer = 6
Dim z As Integer = 23
x = y + z
```

Even though `y + z` returns a value that is type **Integer**, it can be implicitly converted to type **Double** and assigned to `x` because there is no possible loss of information in a conversion in that direction.

Conversions in the other direction are called narrowing conversions. These are conversions that can result in an overflow error at runtime. An example is shown below:

```
Dim x As Integer
Dim y As Double = 34.445
Dim z As Double = 22.12
x = y + z
' Since x is an Integer, it cannot accept the information on the right
' side of the decimal point. Conversion will lose information.
```

If **Option Strict** is **On**, the above example raises a compiler error. If **Option Strict** is **Off**, however, the conversion is performed implicitly, even though this implicit conversion may cause an error at runtime. For this reason, you should always use **Option Strict On**. For further information about widening and narrowing conversions, see Widening and Narrowing Conversions.

Though a widening conversion will never cause an overflow error, one can sometimes cause a loss of precision. For example, conversion of type **Decimal** to type **Single** is a widening conversion. But the **Decimal** type holds more significant digits than **Single**, so this conversion may cause the least significant digits to be lost. For this reason, all conversions should be handled carefully.

See Also

Statements Overview | Declaration Statements | Executable Statements | Assignment Operators | Widening and Narrowing Conversions

Declaration Statements

You use declaration statements to name and define procedures, variables, arrays, and constants. When you declare a procedure, variable, or constant, you also define its scope, depending on where you place the declaration and what keywords you use to declare it.

The following example contains three declarations.

```
Public Sub ApplyFormat()
   Const limit As Integer = 33
   Dim myWidget As Widget
   ' Insert code to implement the method.
End Sub
```

First, the **Public Sub** statement (with matching **End Sub** statement) declares a procedure named `ApplyFormat`. All the statements enclosed by the **Public Sub** and **End Sub** statements are executed whenever the `ApplyFormat` procedure is called or run.

Next, the **Const** statement declares the constant `limit`, specifying the **Integer** data type and a value of 33.

Finally, the **Dim** statement declares the variable `myWidget`. The data type is an object, in this case, a `Widget` object. You can declare a variable to be any object that is exposed in the application you are using. **Dim** statements are one type of statement used to declare variables. Other keywords used in declarations are **ReDim, Static, Public, Private, Protected,** and **Friend**.

A declaration statement reserves the memory needed to create a variable, but does not explicitly create it. You can explicitly create a variable and assign it a value at the same time by using the following syntax:

```
Dim x As Integer = 45   ' x is created and assigned the value 45.
```

If your variable is an object variable, you can explicitly create an instance of its class when you declare it by using the **New** keyword, as illustrated below:

```
Dim x As New System.Windows.Forms.Form()
```

See Also

Statements Overview | Assignment Statements | Executable Statements | Declaring Constants

Executable Statements

An executable statement initiates action by executing a method. It can loop or branch through blocks of code. Executable statements often contain mathematical or conditional operators.

The following example uses an **If...Then...Else** statement to execute different blocks of code based on the value of a variable. Within each block of code, a **For...Next** statement causes a loop to be executed a specified number of times.

```
Public Sub StartWidget(aWidget As Widget, clockwise As Boolean, _
   revolutions As Integer)
   Dim counter As Integer
   If clockwise = True Then
      For counter = 1 to revolutions
         aWidget.SpinClockwise
      Next counter
   Else
      For counter = 1 to revolutions
         aWidget.SpinCounterClockwise
      Next counter
   End If
End Sub
```

The **If...Then...Else** statement in the example checks the value of the variable `clockwise`. If the value is **True**, the `SpinClockwise` method of `aWidget` is invoked. If the value is **False**, then the `SpinCounterClockwise` method of `aWidget` is invoked. **If...Then...Else** statements end with **End If**.

The **For...Next** statements within each block cause the appropriate method to be invoked a number of times equal to the `revolutions` variable.

See Also

Statements Overview | Assignment Statements | Declaration Statements

Procedures

A *procedure* is a block of Visual Basic statements enclosed by a declaration statement and an **End** statement. All Visual Basic code is written within procedures.

In This Section

Procedures Overview
Provides an overview of the types of procedures, and explains the benefits of using them to structure your code.
Sub Procedures
Describes how to declare and call **Sub** procedures.
Function Procedures
Describes how to declare and call **Function** procedures.
Property Procedures
Describes how to declare and call **Property** procedures.
Procedure Arguments
Provides information on defining arguments and specifying them in calls.
Recursive Procedures
Provides information on procedures that call themselves.
Procedure Overloading
Describes how to define multiple versions of a procedure.

Related Sections

Object-Oriented Programming in Visual Basic
Covers the basics of object-oriented programming.

Procedures Overview

A *procedure* is a block of Visual Basic statements enclosed by a declaration statement and an **End** statement. All Visual Basic code is written within procedures.

A procedure is invoked from some other place in the code. When it is finished executing, it returns control to the code that invoked it; this code is known as the *calling code*. The calling code is a statement, or an expression within a statement, that specifies the procedure by name and transfers control to it.

Types of Procedures

Visual Basic uses several types of procedures:

- **Sub** procedures perform actions but do not return a value to the calling code.
- Event-handling procedures are **Sub** procedures that execute in response to an event triggered by user action or by an occurrence in a program.
- **Function** procedures return a value to the calling code.
- **Property** procedures return and assign values of properties on objects or modules.

Procedures and Structured Code

Every line of code in your application must be inside some procedure, such as `Main`, `Calculate`, or `Button1_Click`. If you subdivide large procedures into smaller ones, your application is more readable.

Procedures are useful for performing repeated or shared tasks, such as frequently used calculations, text and control manipulation, and database operations. You can call a procedure from many different places in your code, so you can use procedures as building blocks for your application.

Structuring your code with procedures gives you the following benefits:

- Procedures allow you to break your programs into discrete logical units. You can debug separate units more easily than you can an entire program without procedures.
- You can use procedures developed for one program in other programs, often with little or no modification.

See Also

Procedures | Sub Procedures | Function Procedures | Property Procedures | Procedure Arguments | Recursive Procedures | Procedure Overloading | Sub Statement

Sub Procedures

A **Sub** procedure is a series of Visual Basic statements enclosed by the **Sub** and **End Sub** statements. Each time the procedure is called, its statements are executed, starting with the first executable statement after the **Sub** statement and ending with the first **End Sub**, **Exit Sub**, or **Return** statement encountered.

A **Sub** procedure performs actions but does not return a value. It can take arguments, such as constants, variables, or expressions, that are passed to it by the calling code.

The syntax for declaring a **Sub** procedure is as follows:

```
[accessibility] Sub subname[(argumentlist)]
   ' Statements of the Sub procedure go here.
End Sub
```

The accessibility can be Public, Protected, Friend, **Protected Friend**, or Private.

You can define **Sub** procedures in modules, classes, and structures. They are **Public** by default, which means you can call them from anywhere in your application.

Argument Declaration

You declare each argument for a procedure the same way you declare a variable, specifying the argument name and data type. You can also specify the passing mechanism, and whether the argument is optional.

The syntax for each argument in the argument list is as follows:

```
[Optional] [ByVal|ByRef] [ParamArray] argumentname As datatype
```

If the argument is optional, you must also supply a default value in its declaration, as follows:

```
Optional [ByVal|ByRef] argumentname As datatype = defaultvalue
```

Calling Syntax

You invoke a **Sub** procedure explicitly with a stand-alone calling statement. You cannot call it by using its name within an expression. The calling statement must provide values for all arguments that are not optional, and it must enclose the argument list in parentheses. If no arguments are supplied, you can optionally omit the parentheses. The use of the **Call** keyword is optional also.

The syntax for a call to a **Sub** procedure is as follows:

```
[Call] subname[(argumentlist)]
```

The following **Sub** procedure tells the computer operator which task the application is about to perform, and also displays a time stamp. Instead of duplicating this code at the beginning of every task, the application simply calls `TellOperator` from various places. Each call passes a string in the `Task` argument that identifies the task being started.

```
Sub TellOperator(ByVal Task As String)
   Dim Stamp As Date      ' Stamp is local to TellOperator.
   Stamp = TimeOfDay()    ' Get current time for time stamp.
   ' Use MessageBox class of System.Windows.Forms namespace.
   MessageBox.Show("Starting " & Task & " at " & CStr(Stamp))
End Sub
```

A typical call to `TellOperator` is as follows:

```
Call TellOperator("file update")
```

See Also

Event-Handling Procedures

An event is an action or an occurrence — such as a mouse click or a credit limit exceeded — that is recognized by some object, and for which you can write code to respond. Events can occur as a result of a user action or a program calculation, or they can be triggered by the system. An event-handling procedure, or *event handler*, is the code you write to respond to a particular event.

Although an event handler is a **Sub** procedure, you usually do not call it in the same manner as other **Sub** procedures. When an object recognizes that an event has occurred, it invokes the event handler using the name corresponding to the event. This is known as *raising* the event, which the object does by executing the **RaiseEvent** statement.

For a discussion of writing event handlers, see Writing Event Handlers.

See Also

Procedures | Sub Procedures | Procedure Arguments | Events and Delegates | Events and Event Handlers | RaiseEvent Statement

Function Procedures

A **Function** procedure is a series of Visual Basic statements enclosed by the **Function** and **End Function** statements. Each time the procedure is called, its statements are executed, starting with the first executable statement after the **Function** statement and ending with the first **End Function**, **Exit Function**, or **Return** statement encountered.

A **Function** procedure is similar to a **Sub** procedure, but it also returns a value to the calling program. A **Function** procedure can take arguments passed to it by the calling code, such as constants, variables, or expressions.

The syntax for declaring a **Function** procedure is as follows:

```
[accessibility] Function functionname[(argumentlist)] As datatype
   ' Statements of the Function procedure.
End Function
```

The accessibility can be Public, Protected, Friend, **Protected Friend**, or Private.

You can define **Function** procedures in modules, classes, and structures. They are **Public** by default, which means you can call them from anywhere in your application.

You declare each argument the same as you do for a **Sub** procedure.

Return Values

The value a **Function** procedure sends back to the calling program is called its *return value*. The function returns the value in one of two ways:

- It assigns a value to its own function name in one or more statements of the procedure. Control is not returned to the calling program until an **Exit Function** or **End Function** statement is executed, as in the following example:

```
Function functionname[(argumentlist)] As datatype
   ' ...
   functionname = expression   'Control remains within the function.
   ' ...
End Function
```

- It uses the **Return** statement to specify the return value, and returns control immediately to the calling program, as in the following example:

```
Function functionname[(argumentlist)] As datatype
   ' ...
   Return expression   'Control is returned immediately.
   ' ...
End Function
```

The advantage of assigning the return value to the function name is that control does not return from the function until the program encounters an **Exit Function** or **End Function** statement. This allows you to assign a preliminary value and adjust it later if necessary.

If the function returns an array data type, you cannot access the individual elements of the array within the function. If you attempt to do this, the compiler interprets it as a call to the function. The following example generates compiler errors:

```
Function AllOnes(ByVal N As Integer) As Integer()
   Dim I As Integer    ' Singleton Integer, not Integer array.
   For I = 1 To N - 1    ' Assume N has a valid value.
      AllOnes(I) = 1    ' Compiler error: call with wrong data type.
   Next I
   Return AllOnes()    ' Compiler error: call with no argument.
End Function
```

Every **Function** procedure has a data type, just as every variable does. This data type is specified by the **As** clause in the **Function** statement, and it determines the type of the return value. The following sample declarations illustrate this:

```
Function Yesterday As Date
   ' ...
End Function
Function FindSqrt(ByVal Number As Single) As Single
   ' ...
End Function
```

Calling Syntax

You call a **Function** procedure by including its name and arguments either on the right side of an assignment statement or in an expression. The following sample calls illustrate this:

```
lvalue = functionname[(argumentlist)]
If ((functionname[(argumentlist)] / 3) <= expression) Then ...
```

The following **Function** procedure calculates the longest side, or hypotenuse, of a right triangle, given the values for the other two sides:

```
Function Hypotenuse (ByVal Side1 As Single, ByVal Side2 As Single) As Single
   Return Math.Sqrt((Side1 ^ 2) + (Side2 ^ 2))
End Function
```

Typical calls to `Hypotenuse` are as follows:

```
Dim TestLength, TestHypotenuse, X, Y, Area As Single
TestHypotenuse = Hypotenuse(TestLength, 10.7)
' Assume TriangleArea is a defined Function procedure — the
' returned value from Hypotenuse is used as an argument to TriangleArea.
Area = TriangleArea(X, Y, Hypotenuse(X, Y))
```

The common language runtime provides many functions, located in the **Microsoft.VisualBasic** namespace. These include such common functions as **Beep**, **MsgBox**, and **StrComp**. You call these functions the same way you call your own **Function** procedures.

When you call a **Function** procedure, you do not have to use its return value. If you do not, all the actions of the function are performed, but the return value is ignored. **MsgBox** is often called in this manner.

See Also

Function Statement | Procedures | Sub Procedures | Property Procedures | Procedure Arguments

Property Procedures

A **Property** procedure is a series of Visual Basic statements that manipulate a custom property on a module, class, or structure. Property procedures are also known as *property accessors*.

A property differs from a public variable or a field in the following respects:

- You implement a property with executable code (its **Property** procedures), rather than with a single declaration statement.
- **Property** procedures are executed when the property value is set or retrieved. This allows a class to perform custom actions when client code accesses the property.
- A property does not have any storage location associated with its declaration. Although its **Property** procedures often define local variables and constants, these are usually not available to the code accessing the property. Therefore, while you can include a variable or a field as a member of a structure or array, you cannot do this with a property.
- You can define a property as read-only, write-only, or read/write. The default is read/write.

Visual Basic provides the following **Property** procedures:

- **Get** procedures return the value of a property; they are called when the property is accessed in an expression.
- **Set** procedures set a property to a value, including an object reference; they are called when a value is assigned to the property.

You usually define **Property** procedures in pairs, using the **Get** and **Set** keywords, but you can define either procedure alone if the property is read-only (**Get**) or write-only (**Set**).

Property Declaration

A property itself is defined by a block of code enclosed within the **Property** and **End Property** statements. Inside this block, each **Property** procedure appears as an internal block enclosed within a declaration statement (**Get** or **Set**) and an **End** statement. The syntax for declaring a property and its procedures is as follows:

```
[Default] [accessibility] Property propertyname[(argumentlist)] As datatype
Get
   ' ...
   Return expression   ' expression is returned as property's value.
End Get
Set [(ByVal newvalue As datatype)]
   ' ...
   lvalue = newvalue   ' newvalue is the new value to be assigned.
End Set
End Property
```

The accessibility can be Public, Protected, Friend, **Protected Friend**, or Private.

You can define properties in classes, structures, and modules. Properties are **Public** by default, which means you can call them from anywhere in your application.

Argument Declaration

You declare each argument the same as for a **Function** or **Sub** procedure, except that the passing mechanism must be by value.

The syntax for each argument in the argument list is as follows:

```
[Optional] ByVal [ParamArray] argumentname As datatype
```

If the argument is optional, you must also supply a default value in its declaration, as follows:

```
Optional ByVal argumentname As datatype = defaultvalue
```

In a **Get** procedure, the return value is supplied to the calling expression as the value of the property.

In a **Set** procedure, the new property value is passed through the argument of the **Set** statement. If you declare an explicit argument, you must declare it as the same data type as the property. If you do not declare an argument, the compiler uses the

implicit argument ***Value*** to represent the new value to be assigned to the property.

Calling Syntax

A **Property** procedure is invoked implicitly by code that makes reference to the property. The code uses the name of the property the same way it would use the name of a variable, except that it must provide values for all arguments that are not optional, and it must enclose the argument list in parentheses. If no arguments are supplied, you can optionally omit the parentheses.

The syntax for an implicit call to a **Set Property** procedure is as follows:

```
propertyname[(argumentlist)] = expression
```

The following sample calls show allowable calling syntax for a **Get Property** procedure:

```
lvalue = propertyname[(argumentlist)]
Do While (propertyname[(argumentlist)] > expression)
```

> **Note** A property's data type and access are defined in the **Property** statement, not in the **Property** procedures. A property can have only one data type and one accessibility. For example, you cannot define a property to set a **Decimal** value, but get a **Double** value. Similarly, you cannot define a **Private Set** and a **Public Get** on the same property. To achieve this functionality, you can define a **Public ReadOnly** property and a **Private** setting method separate from the property.

See Also

Procedure Arguments | Property Statement

Procedure Arguments

In most cases, a procedure needs some information about the circumstances in which it has just been called. A procedure that performs repeated or shared tasks uses different information for each call. This information consists of variables, constants, and expressions that are passed to the procedure when it is called. Each value passed to a procedure is called an *argument*.

In This Section

Argument Data Type Declaration
Shows how to define data types of arguments.
Argument Passing ByVal and ByRef
Describes how to pass arguments **ByVal** and **ByRef**.
Argument Passing by Position and by Name
Describes how to pass arguments by position and by name.
Optional Arguments
Describes arguments that do not have to be supplied when a procedure is called.
Parameter Arrays
Discusses how to pass an indefinite number of arguments in a call.

Related Sections

Sub Procedures
Describes how to declare and call **Sub** procedures.
Function Procedures
Describes how to declare and call **Function** procedures.
Property Procedures
Describes how to declare and call **Property** procedures.

Argument Data Type Declaration

You declare a data type for an argument by using the **As** clause in its declaration. For example, the following function accepts a string and an integer:

```
Function Appointment(ByVal Day As String, ByVal Hour As Integer) As String
   ' Code to return any appointment for the given day and time.
End Function
```

If the type checking switch (Option Strict Statement) is off, the **As** clause is optional, except that if any one argument uses it, all arguments must use it. If type checking is on, the **As** clause is required for all procedure arguments.

If you intend to pass different data types to a particular argument, such as **Byte** to **Integer** or late bound classes to **Object**, you must have **Option Strict Off**.

See Also

Procedures | Procedure Arguments | Argument Passing ByVal and ByRef | Argument Passing by Position and by Name | Optional Arguments | Parameter Arrays | Type Checking | Function Statement | Data Types | Visual Basic Language and Run-Time Reference

Argument Passing ByVal and ByRef

In Visual Basic, you can pass an argument to a procedure *by value* or *by reference* by specifying the **ByVal** or **ByRef** keywords, respectively. Passing an argument by value means the procedure cannot modify the contents of the variable element in the calling code underlying the argument. Passing by reference allows the procedure to modify the contents in the same way that the calling code itself can.

Passing arguments by value and by reference is not the same distinction as the classification of data types into value types and reference types. However, the two categories do interact.

Variable and Nonvariable Arguments

The programming element underlying an argument can be either a variable element, capable of having its value changed, or a nonvariable element. The following table lists variable and nonvariable elements.

Variable elements	Nonvariable elements
Declared variables, including object variables	Constants
Fields (of classes)	Literals
Array elements	Enumerations
Structure elements	Expressions

Nonvariable arguments are never modified in the calling code, even if they are passed by reference. The called procedure might modify its copy of such an argument, but that modification does not affect the underlying element in the calling code.

In This Section

Argument Passing ByVal
Describes passing arguments by value, which means the procedure cannot modify the variable itself.
Argument Passing ByRef
Describes passing arguments by reference, which means the procedure can modify the variable itself.
Argument Passing Mechanism
Summarizes the interaction between the element's data type and the passing mechanism.

Related Sections

Procedures
Provides links to topics about Visual Basic procedures, including information on the different procedure types and how they are called.

Argument Passing ByVal

If you pass a variable argument by value using the **ByVal** keyword, the procedure cannot modify the variable itself. However, if the argument is a reference type, you can modify the members of the object to which it points, even though you cannot replace the object itself. In particular, you can modify the members of the object. For example, if the argument is an array variable, you cannot assign a new array to it, but you can change one or more of its elements. The changed elements are reflected in the underlying array variable in the calling code.

The following example shows two subroutines that take an array variable by value and operate on its elements. `Increase` simply adds one to each element. `Replace` assigns a new array to the argument array `A()` and then adds one to each element. However, the reassignment does not affect the underlying array variable in the calling code.

```
Public Sub Increase(ByVal A() As Long)
   Dim J As Integer
   For J = 0 To UBound(A)
      A(J) = A(J) + 1
   Next J
End Sub
   ' ...
Public Sub Replace(ByVal A() As Long)
   Dim J As Integer
   Dim K() As Long = {100, 200, 300}
   A = K
   For J = 0 To UBound(A)
      A(J) = A(J) + 1
   Next J
End Sub
 ' ...
Dim N() As Long = {10, 20, 30, 40}
Increase(N)
MsgBox("After Increase(N): " & _
             CStr(N(0)) & CStr(N(1)) & CStr(N(2)) & CStr(N(3)))
' The element values of N() are now 11, 21, 31, and 41. Because N is a
' reference type, Replace could change its members.
Replace(N)
MsgBox("After Replace(N): " & _
             CStr(N(0)) & CStr(N(1)) & CStr(N(2)) & CStr(N(3)))
' The element values of N() are still 11, 21, 31, and 41. Because N was
' passed ByVal, Replace could not modify the variable N by assigning a
' new array to it. When Replace created the new array instance K and
' assigned it to the local variable A, it lost the reference to N()
' passed in by the calling code. When it changed the members of A(),
' only the local array K() was affected.
```

See Also

Argument Passing ByRef | Argument Passing Mechanism | Argument Passing by Position and by Name | ByVal | ByRef

Argument Passing ByRef

If you pass a variable argument by reference using the **ByRef** keyword, the procedure can modify the variable itself. In particular, if the argument is an object variable, you can assign a new object to it. This reassignment also affects the object variable in the calling program.

In the example provided in Argument Passing ByVal, if the `Replace` subroutine had instead taken the argument array by reference, the new array `K()` would have been assigned to the array variable `N` in the calling code. The results for `N()` would then have been as follows:

```
MsgBox("After Replace(N) taking A() argument ByRef: " & _
             CStr(N(0)) & CStr(N(1)) & CStr(N(2)) & CStr(N(3)))
' The element values of N() are now 101, 201, and 301. Because N was
' passed ByRef, Replace could modify the variable N and assign a new
' array to it. Because N is a reference type, Replace could also change
' its members.
```

See Also

Argument Passing Mechanism | Argument Passing by Position and by Name | ByVal | ByRef

Argument Passing Mechanism

When a nonvariable element is passed as an argument, the procedure can never modify it in the calling code, whether it is passed **ByVal** or **ByRef**. For a variable element, the following table summarizes the interaction between the element's data type and the passing mechanism.

Element type	Passed ByVal	Passed ByRef
Value type (contains only a value)	The procedure cannot change the variable or any of its members.	The procedure can change the variable and its members.
Reference type (contains a pointer to a class or structure instance)	The procedure cannot change the variable but can change members of the instance to which it points.	The procedure can change the variable and members of the instance to which it points.

Determination of the Passing Mechanism

The procedure declaration determines the passing mechanism. The calling code cannot override a **ByVal** mechanism, but if an argument is declared with **ByRef**, the calling code can force the mechanism to pass by value by enclosing the argument name in parentheses in the call. The following example illustrates this:

```
Sub DoSomething(ByRef InString As String)
   ' Code that makes changes to InString.
End Sub
' ...
Dim Str As String = "Cannot be replaced if passed ByVal"
Call DoSomething((Str))   ' Pass Str ByVal even though declared ByRef.
' The parentheses around Str protect it from change.
```

The default in Visual Basic is to pass arguments by value. You can make your code easier to read by using the **ByVal** keyword, as in the following example:

```
Sub PostAccounts(ByVal AcctNum As Integer)
   ' Code to post accounts.
End Sub
```

When you pass a variable by reference, you must use the **ByRef** keyword to specify this mechanism. The preceding example can be extended with a second argument as follows:

```
Sub PostAccount(ByVal AcctNum As Integer, ByRef RunningTotal As Single)
   ' Code to post account for AcctNum and change value of RunningTotal.
End Sub
```

It is good programming practice to include either the **ByVal** or **ByRef** keyword with every declared argument.

Choice of the Passing Mechanism

In choosing between the two passing mechanisms, the most important criterion is the exposure of calling variables to change. The advantage of passing an argument **ByRef** is that the procedure can return a value to the calling code through that argument. The advantage of passing an argument **ByVal** is that it protects a variable from being changed by the procedure.

Although the passing mechanism can also affect the performance of your code, the difference is usually insignificant. One exception to this is a value type passed **ByVal**. In this case, Visual Basic copies the entire data contents of the argument. Therefore, for a large value type such as a structure, it is more efficient to pass it **ByRef**.

For reference types, only the four-byte pointer to the data is copied. Therefore, you can pass arguments of type **String** or **Object** **ByVal** without harming performance.

See Also

Argument Passing ByVal | Argument Passing ByRef | Argument Passing by Position and by Name | Procedures | Procedure Arguments | Argument Data Type Declaration | Optional Arguments | Parameter Arrays | Value Types and Reference Types | ByVal | ByRef

Argument Passing by Position and by Name

When you call a **Sub** or **Function** procedure, you can pass arguments *by position* — that is, in the order in which they appear in the procedure's definition — or you can pass them *by name*, without regard to position.

When you pass an argument by name, you specify the argument's declared name followed by a colon and an equal sign (**:=**), followed by the argument value. You can supply named arguments in any order.

For example, the following **Sub** procedure takes three arguments:

```
Sub StudentInfo(ByVal Name As String, Optional ByVal Age As Short = 0, _
                        Optional ByVal Birth As Date = #1/1/2000#)
   Debug.WriteLine("Name = " & Name & "; age = " & CStr(Age) & _
                   "; birth date = " & CStr(Birth))
End Sub
```

When you call this procedure, you can supply the arguments by position, by name, or by using a mixture of both.

Passing Arguments by Position

You can call the procedure `StudentInfo` with its arguments passed by position and delimited by commas, as shown in the following example:

```
StudentInfo("Mary", 19, #9/21/1981#)
```

If you omit an optional argument in a positional argument list, you must hold its place with a comma. The following example calls `StudentInfo` without the `Age` argument:

```
StudentInfo("Mary", , #9/21/1981#)
```

Passing Arguments by Name

Alternatively, you can call `StudentInfo` with the arguments passed by name, also delimited by commas, as shown in the following example:

```
StudentInfo(Age:=19, Birth:=#9/21/1981#, Name:="Mary")
```

Mixing Arguments by Position and by Name

You can supply arguments both by position and by name in a single procedure call, as shown in the following example:

```
StudentInfo("Mary", Birth:=#9/21/1981#)
```

In the preceding example, no extra comma is necessary to hold the place of the omitted `Age` argument, since `Birth` is passed by name.

If you supply arguments both by position and by name, the positional arguments must all come first. Once you supply an argument by name, the remaining arguments must all be by name.

Supplying Optional Arguments by Name

Passing arguments by name is especially useful when you call a procedure that has more than one optional argument. If you supply arguments by name, you do not have to use consecutive commas to denote missing positional arguments. Passing arguments by name also makes it easier to keep track of which arguments you are passing and which ones you are omitting.

Restrictions on Supplying Arguments by Name

You cannot pass arguments by name to avoid entering required arguments. You can omit only the optional arguments.

You cannot pass a parameter array by name. This is because when you call the procedure, you supply an indefinite number of

comma-separated arguments for the parameter array, and the compiler cannot associate more than one argument with a single name.

See Also

Procedures | Procedure Arguments | Argument Data Type Declaration | Argument Passing ByVal and ByRef | Optional Arguments | Parameter Arrays | ByVal | ByRef | Optional | ParamArray

Optional Arguments

You can specify that a procedure argument is optional and does not have to be supplied when the procedure is called. *Optional arguments* are indicated by the **Optional** keyword in the procedure definition. The following rules apply:

- Every optional argument in the procedure definition must specify a default value.
- The default value for an optional argument must be a constant expression.
- Every argument following an optional argument in the procedure definition must also be optional.

The following syntax shows a procedure declaration with an optional argument:

```
    Sub subname(ByVal arg1 As type1, Optional ByVal arg2 As type2 = default)
```

Calling Procedures with Optional Arguments

When you call a procedure with an optional argument, you can choose whether to supply the argument. If you do not, the procedure uses the default value declared for that argument.

When you omit one or more optional arguments in the argument list, you use successive commas to mark their positions. The following example call supplies the first and fourth arguments but not the second or third:

```
    Call subname(arg1, , , arg4)   ' Leaves out arg2 and arg3.
```

Determining Whether an Optional Argument Is Present

A procedure cannot detect at run time whether a given argument has been omitted or the calling code has explicitly supplied the default value. If you need to make this distinction, you can set an unlikely value as the default. The following procedure includes the optional argument `Office`, and tests for its default value, `QJZ`, to see if it has been omitted:

```
Sub Notify(ByVal Company As String, Optional ByVal Office As String = "QJZ")
   If Office = "QJZ" Then
      Debug.WriteLine("Office not supplied -- notifying Headquarters")
      Office = "Headquarters"
   End If
   ' Code to notify headquarters or specified office.
End Sub
```

If the optional argument is a reference type such as a **String**, you can use **Nothing** as the default value, provided this is not an expected value for the argument.

Another way to define a procedure with optional arguments is to use overloading. If you have one optional argument, you can define two overloaded versions of the procedure, one with the argument and one without it. This approach becomes more complicated as the number of optional arguments increases. However, its advantage is that you can be absolutely sure whether the calling program supplied each optional argument.

See Also

Procedures | Procedure Arguments | Argument Data Type Declaration | Argument Passing ByVal and ByRef | Argument Passing by Position and by Name | Parameter Arrays | Procedure Overloading | Considerations in Overloading Procedures | Optional

Parameter Arrays

Usually, you cannot call a procedure with more arguments than the procedure declaration specifies. When you need an indefinite number of arguments, you can declare a *parameter array*, which allows a procedure to accept an array of values for an argument. You do not have to know the number of elements in the parameter array when you define the procedure. The array size is determined individually by each call to the procedure.

You use the **ParamArray** keyword to denote a parameter array. The following rules apply:

- A procedure can have only one parameter array, and it must be the last argument in the procedure definition.
- The parameter array must be passed by value. It is good programming practice to explicitly include the **ByVal** keyword in the procedure definition.
- The code within the procedure must treat the parameter array as a one-dimensional array, each element of which is the same data type as the **ParamArray** data type.
- The parameter array is automatically optional. Its default value is an empty one-dimensional array of the parameter array's element type.
- All arguments preceding the parameter array must be required. The parameter array must be the only optional argument.

When you call a procedure with a parameter array argument, you can pass any of the following for the parameter array:

- Nothing — that is, you can omit the **ParamArray** argument. In this case, an empty array is passed to the procedure. You can also pass the **Nothing** keyword, with the same effect.
- A list of an indefinite number of arguments, separated by commas. The data type of each argument must be implicitly convertible to the **ParamArray** element type.
- An array with the same element type as the parameter array.

The following example shows how you can define a procedure with a parameter array:

```
Sub StudentScores(ByVal Name As String, ByVal ParamArray Scores() As String)
   Dim I As Integer
   Debug.WriteLine("Scores for " & Name & ":")
   ' Use UBound function to determine largest subscript of array.
   For I = 0 To UBound(Scores)
      Debug.WriteLine("Score " & I & ": " & Scores(I))
   Next I
End Sub
```

The following examples show typical calls to `StudentScores`:

```
StudentScores("Anne", "10", "26", "32", "15", "22", "24", "16")
StudentScores("Mary", "High", "Low", "Average", "High")
Dim JohnScores() As String = {"35", "Absent", "21", "30"}
StudentScores("John", JohnScores)
```

See Also

Procedures | Procedure Arguments | Argument Data Type Declaration | Argument Passing ByVal and ByRef | Argument Passing by Position and by Name | Optional Arguments | Type Checking | Arrays | ParamArray | ByVal | Optional | UBound Function

Recursive Procedures

A *recursive* procedure is one that calls itself. For example, the following procedure uses recursion to calculate the factorial of its original argument:

```
Function Factorial(ByVal N As Integer) As Integer
   If N <= 1 Then      ' Reached end of recursive calls.
      Return 1         ' N = 0 or 1, so climb back out of calls.
   Else                ' N > 1, so call Factorial again.
      Return Factorial(N - 1) * N
   End If
End Function
```

> **Note** When a **Function** procedure calls itself recursively, the procedure name must be followed by parentheses, even if there is no argument list. Otherwise, the function name is taken as representing the return value of the function.

Programs have a limited amount of space for variables. Each time a procedure calls itself, more of that space is used. If this process continues indefinitely, it eventually causes a stack space error. The cause may be less obvious if two procedures call each other indefinitely, or if a condition that limits the recursion is never met.

You should test your recursive procedure to make sure it does not call itself indefinitely, or so many times that you run out of memory. You can often substitute loops for recursion.

See Also

Loop Structures

Procedure Overloading

Overloading a procedure means defining it in multiple versions, using the same name but different argument lists. The purpose of overloading is to define several closely related versions of a procedure without having to differentiate them by name. You do this by varying the argument list.

In This Section

Rules of Procedure Overloading
Discusses the requirements of procedure overloading, including the ways in which overloaded versions must differ from each other (in number, order, or data types of arguments).

Considerations in Overloading Procedures
Discusses restrictions on overloading optional arguments, implicit overloads for **ParamArray** arguments, and typeless programming as an alternative to overloading.

Overload Resolution
Describes the steps the Visual Basic compiler takes when, encountering a call to a procedure that is defined in several overloaded versions, it decides which overload to call.

Related Sections

Procedures
Provides links to topics about Visual Basic procedures, including information on the different procedure types and how they are called.

Rules of Procedure Overloading

When you overload a procedure, the following rules apply:

- Each overloaded version must use the same procedure name.
- Each overloaded version must differ from all other overloaded versions in at least one of the following respects:
    - Number of arguments
    - Order of the arguments
    - Data types of the arguments

The compiler uses the procedure name and the argument list to check that the call correctly matches the definition. These items are collectively called the *signature* of the procedure.

You cannot overload a procedure without varying the argument list. In particular, you cannot overload a procedure by varying only one or more of the following items:

- Procedure modifier keywords, such as **Public**, **Shared**, and **Static**
- Argument names
- Argument modifier keywords, such as **ByRef** and **Optional**
- The data type of the return value

The items in the preceding list are not part of the signature.

> **Note** If you intend to pass a late-bound object variable to an overloaded procedure, you must declare the appropriate argument as **Object**.

Multiple Versions of a Procedure

Suppose you are writing a **Sub** procedure to post a transaction against a customer's balance, and you want to be able to refer to the customer either by name or by account number. To accommodate this, you can define two different **Sub** procedures, as in the following example:

```
Sub PostName(ByVal CustName As String, ByVal Amount As Single)
Sub PostAcct(ByVal CustAcct As Integer, ByVal Amount As Single)
```

An alternative is to overload a single procedure name. You can use the **Overloads** keyword to define a version of the procedure for each argument list, as follows:

```
Overloads Sub Post(ByVal CustName As String, ByVal Amount As Single)
   ' Code to access customer record by customer name.
End Sub
Overloads Sub Post(ByVal CustAcct As Integer, ByVal Amount As Single)
   ' Code to access customer record by account number.
End Sub
```

If you also wanted to accept a transaction amount in either **Decimal** or **Single**, you could further overload `Post` to allow for this variation. If you did this to each of the overloads in the preceding example, you would have four **Sub** procedures, all with the same name but with four different argument lists.

The advantage of overloading a procedure is in the flexibility of the call. To use the `Post` procedure declared in the preceding example, the calling code can obtain the customer identification as either a **String** or an **Integer**, and then call the same procedure in either case. The following example illustrates this:

```
Imports MSVB = Microsoft.VisualBasic   ' Namespace for InputBox.
Dim Customer As String       ' Customer's name.
Dim AccountNum As Integer    ' Customer's account number.
Dim Amount As Single
' ...
Customer = MSVB.Interaction.InputBox("Enter customer name or number")
Amount = MSVB.Interaction.InputBox("Enter transaction amount")
Try
AccountNum = CInt(Customer)   ' See if an integer was entered.
```

```
Post(AccountNum, Amount)   ' Post transaction against account number.
Catch
Post(Customer, Amount)      ' Post transaction against customer name.
End Try
```

See Also

Considerations in Overloading Procedures | Overload Resolution | Overloads

Considerations in Overloading Procedures

The following rules apply to overloading procedures:

- You must differentiate any two versions of a procedure by their argument lists.
- You cannot use anything other than the argument list to differentiate versions. For example, you cannot have two overloads that differ only in their return types.
- However, versions that are properly differentiated by their argument lists can have different return types or different argument modifiers.
- You can overload a **Function** procedure with a **Sub** procedure, and vice versa, provided they have different argument lists.
- You can overload a method or a property on a class you are defining. You do this the same way you overload procedures.
- However, you cannot define a method and a property with the same name in the same class. In other words, you cannot overload a method with a property, or vice versa.

Restrictions on Overloading Optional Arguments

A procedure with an optional argument is equivalent to two overloaded procedures, one with the optional argument and one without it. Because of this, you cannot overload such a procedure with an argument list corresponding to either of these. The following declarations illustrate this:

```
Overloads Sub Q(ByVal B As Byte, Optional ByVal J As Long = 6)
' The preceding definition is equivalent to the following two:
'                   Overloads Sub Q(ByVal B As Byte)
'                   Overloads Sub Q(ByVal B As Byte, ByVal J As Long)
' Therefore, the following overload is not valid:
Overloads Sub Q(ByVal C As Byte, ByVal K As Long)   ' Signature in use.
' The following overload uses a different argument list and is valid:
Overloads Sub Q(ByVal B As Byte, ByVal J As Long, ByVal S As Single)
```

For a procedure with more than one optional argument, there is a set of invalid overloads, arrived at by logic similar to that in the preceding example.

Implicit Overloads for a ParamArray Argument

The compiler considers a procedure with a **ParamArray** argument to have an infinite number of overloads, differing from each other in what is passed to the parameter array, as follows:

- One overload passing a one-dimensional array of the **ParamArray** element type
- One overload passing no argument to the **ParamArray**
- For every positive integer, one overload passing that number of arguments, each of the **ParamArray** element type

The following declarations illustrate these implicit overloads:

```
Sub P(ByVal D As Date, ByVal ParamArray C() As Char)
' The preceding definition is equivalent to the following overloads:
'    Overloads Sub P(ByVal D As Date, ByVal C() As Char)
'    Overloads Sub P(ByVal D As Date)
'    Overloads Sub P(ByVal D As Date, ByVal C As Char)
'    Overloads Sub P(ByVal D As Date, ByVal C1 As Char, ByVal C2 As Char)
' And so on, with an additional Char argument in each successive overload.
```

Typeless Programming as an Alternative to Overloading

If you want to allow different data types to be passed to an argument, an alternative approach is typeless programming; that is, turning **Option Explicit** off and not declaring the argument's data type. However, this approach has the following disadvantages compared to overloading:

- Typeless programming produces less efficient execution code.
- The procedure must test for every data type it anticipates being passed.
- The compiler cannot signal an error if the calling code passes a data type that the procedure does not support.

See Also

Optional Arguments | Parameter Arrays | Procedure Overloading | Overload Resolution | Overloads | Option Explicit Statement | /optionexplicit | Typeless Programming

Overload Resolution

When the Visual Basic compiler encounters a call to a procedure that is defined in several overloaded versions, the compiler must decide which of the overloads to call. It does this by performing the following steps:

1. It eliminates any overload that defines a different number of arguments than are supplied in the call.
2. It eliminates any overload for which the data types of the calling arguments cannot be converted to the types defined in the overload.
3. It eliminates any overload that requires a narrowing conversion from the calling argument types to the defined argument types. This is true whether the type checking switch (Option Strict Statement) is on or off.
4. It considers the remaining overloads in pairs. For each pair, it compares the data types of the defined arguments. If the types in one of the overloads all widen to the corresponding types in the other, the compiler eliminates the latter. That is, it retains the overload that calls for the least amount of widening.
5. It continues considering overloads in pairs until only one overload remains, and it resolves the call to that overload. If the compiler cannot reduce the overloads to a single candidate, it generates an error.

The following example illustrates this overload resolution process.

```
Overloads Sub Z(ByVal X As Byte, ByVal Y As Double)      ' Overload 1.
Overloads Sub Z(ByVal X As Short, ByVal Y As Single)     ' Overload 2.
Overloads Sub Z(ByVal X As Integer, ByVal Y As Single)   ' Overload 3.
' ...
Dim R, S As Short
Call Z(R, S)   ' Calls overload 2 (less widening than overload 3).
' ...
Dim B As Byte, S As Short
Call Z(B, S)   ' Generates compiler error (two candidates).
```

In the first call, `Overload 1` is eliminated because the type of the first calling argument (**Short**) narrows to the corresponding defined type (**Byte**). `Overload 3` is eliminated because each argument type in `Overload 2` (**Short** and **Single**) widens to the corresponding type in `Overload 3` (**Integer** and **Single**). `Overload 2` requires less widening, so the compiler uses it for the call.

In the second call, none of the overloads can be eliminated because of narrowing. `Overload 3` is eliminated for the same reason as in the first call, because `Overload 2` can be called with less widening of the argument types. However, the compiler cannot resolve between `Overload 1` and `Overload 2`. Each has one defined argument type that widens to the corresponding type in the other (**Byte** to **Short**, but **Single** to **Double**). The compiler therefore generates an overload resolution error.

Overloaded Optional and ParamArray Arguments

If two overloads have identical signatures except that the last argument is declared **Optional** in one and **ParamArray** in the other, the compiler resolves a call as follows:

If the call supplies the last argument as	The compiler resolves the call to the overload declaring the last argument as
No value (argument omitted)	**Optional**
A single value	**Optional**
Two or more values in a comma-separated list	**ParamArray**
An array of any length (including an empty array)	**ParamArray**

See Also

Procedure Overloading | Considerations in Overloading Procedures

Control Flow

Left unregulated, a program proceeds through its statements from beginning to end. Some very simple programs can be written with only this unidirectional flow. However, most of the power and utility of any programming language comes from the ability to change execution order with control statements and loops.

Control-flow statements allow you to regulate the flow of your program's execution. Using control statements, you can write Visual Basic code that makes decisions and repeats actions. Another useful statement, **With...End With**, lets you execute a series of statements without having to requalify an object.

In This Section

Decision Structures
Describes control-flow statements used for branching.
Loop Structures
Discusses control-flow statements used to repeat processes.
With...End With Statements
Explains how to execute several statements on the same object.
Nested Control Statements
Covers control-flow statements inside other control-flow statements.
Exit Statement
Summarizes how to transfer control directly out of a control-flow statement.

Related Sections

Control Statement Changes in Visual Basic
Lists changes to the behavior and syntax of control-flow statements from previous versions of Visual Basic.
Control Flow Summary
Provides links to language reference topics on this subject.

Decision Structures

Visual Basic allows you to test conditions and perform different operations depending on the results of that test. You can test for a condition being true or false, for various values of an expression, or for various exceptions generated when you execute a series of statements. The decision statements supported by Visual Basic include:

- **If...Then**
- **If...Then...Else**
- **Select...Case**
- **Try...Catch...Finally**

In This Section

If...Then...Else Statements
Discusses testing one or more **Boolean** conditions.
Select...Case Statements
Covers testing for different values of the same expression.
Try...Catch...Finally Statements
Explains testing for different exceptions generated by your code.

Related Sections

Loop Structures
Provides an overview of control-flow statements used to repeat processes.
With...End With Statements
Explains how to execute several statements on the same object.
Nested Control Statements
Describes control-flow statements inside other control-flow statements.
Exit Statement
Discusses how to transfer control directly out of a control-flow statement.

If...Then...Else Statements

You can use **If...Then...Else** statements to execute a specific statement or block of statements depending on the **Boolean** value of a condition. The condition usually results from a comparison of two values, but it can be any expression that evaluates to a **Boolean** value (**True** or **False**). This includes values of other data types, such as numeric types, that have been converted to **Boolean**.

Executing Statements if a Condition Is True

To execute only one statement when a condition is **True**, you can use the single-line syntax of **If...Then...Else**, omitting the **Else** and **End If** statements, as in the following example:

```
Sub FixDate()
   Dim MyDate As Date = #2/13/1973#
   If MyDate < Now Then MyDate = Now
End Sub
```

To execute more than one line of code when the condition is **True**, you use multiple-line syntax, which includes the **End If** statement. If you have no code to run when the condition is **False**, you omit the **Else** statement, as the following example shows:

```
Dim AlertLabel As New System.Windows.Forms.Label
   ' ...
Sub AlertUser(ByVal Value As Long)
   If Value = 0 Then
     AlertLabel.ForeColor = Color.Red
     AlertLabel.Font = New Font(AlertLabel.Font, _
                                FontStyle.Bold Or FontStyle.Italic)
   End If
End Sub
```

Executing Some Statements if a Condition Is True and Others if It Is False

You can use **If...Then...Else** with the **Else** statement to define two blocks of executable statements. One block is executed if the condition is **True**, the other if it is **False**.

```
Dim AlertLabel As New System.Windows.Forms.Label
   ' ...
Sub AlertUser(ByVal Value As Long)
   If Value = 0 Then
     AlertLabel.ForeColor = Color.Red
     AlertLabel.Font = New Font(AlertLabel.Font, _
                                FontStyle.Bold Or FontStyle.Italic)
   Else
     AlertLabel.Forecolor = Color.Black
     AlertLabel.Font = New Font(AlertLabel.Font, _
                                FontStyle.Regular)
   End If
End Sub
```

Testing Additional Conditions if the First Condition Is False

You can add one or more **ElseIf** statements to **If...Then...Else** to test additional conditions if the first condition is **False**. For example, the following **Function** procedure computes a payroll bonus based on performance rating. The statement following the **Else** statement runs only if the conditions in all of the **If** and **ElseIf** statements are **False**.

```
Function Bonus(ByVal Performance As Integer, ByVal Salary As Decimal) _
              As Decimal
   If Performance = 1 Then
     Return Salary * 0.1
   ElseIf Performance = 2 Then
     Return Salary * 0.09
```

```
    ElseIf Performance = 3 Then
      Return Salary * 0.07
    Else
      Return 0
    End If
  End Function
```

Visual Basic tests the conditions in the order they appear in the **If...Then...Else** statements tests until it encounters a **True** condition or an **Else** statement, at which point it executes the corresponding statement block. Execution then branches to the end of the **If...Then...Else** block.

You can have any number of **ElseIf** statements, or none at all. You can include or omit an **Else** statement regardless of whether you have **ElseIf** statements.

If...Then...Else statements can be nested to as many levels as you need. However, for readability, you might want to use **Select...Case** statements rather than multiple levels of nested **If...Then...Else** statements or many **ElseIf** statements.

See Also

Loop Structures | Nested Control Statements | If...Then...Else Statements (Language Reference) | Select...Case Statements | Try...Catch...Finally Statements | Boolean Data Type | ElseIf

Select...Case Statements

When you are comparing the same expression to several different values, you can use the **Select...Case** statements as an alternative to the **If...Then...Else** statements. Whereas the **If** and **ElseIf** statements can evaluate a different expression in each statement, the **Select** statement evaluates a single expression only once and uses it for every comparison.

Visual Basic compares the value of the expression to the values in the **Case** statements in the order they appear in the **Select...Case** block. If it finds a match or a **Case Else** statement, it executes the corresponding statement block. In any case, it then executes the code following the **End Select** statement.

You can have any number of **Case** statements, and you can include or omit a **Case Else** statement.

In the following example, **Select...Case** is used to evaluate the performance rating that is passed to the **Function** procedure. Note that each **Case** statement can contain more than one value, a range of values, or a combination of values and comparison operators. When a **Case** statement contains more than one value, the **Case** block is executed if any of the values match the value of the **Select** statement expression.

```
Function Bonus(ByVal Performance As Integer, ByVal Salary As Decimal) _
      As Decimal
   Select Performance
     Case 1   ' Performance is 1.
        Return Salary * 0.1
     Case 2, 3   ' Performance is 2 or 3.
        Return Salary * 0.09
     Case 5 To 7   ' Performance is 5, 6, or 7.
        Return Salary * 0.07
     Case 4, 8 To 10   ' Performance is 4, 8, 9, or 10.
        Return Salary * 0.05
     Case Is < 15   ' Performance is 11, 12, 13, or 14.
        Return 100
     Case Else   ' Performance is < 1 or > 14.
        Return 0
   End Select
End Function
```

See Also

Loop Structures | Nested Control Statements | Select...Case Statements (Language Reference) | If...Then...Else Statements | Try...Catch...Finally Statements | Exit Statement

Try...Catch...Finally Statements

You can use the **Try...Catch...Finally** statements for structured exception handling. This allows you to execute a particular block of statements if a specified exception occurs while your code is running. When this happens, the code is said to *throw* the exception, and you *catch* it with the appropriate **Catch** statement.

Execution enters the **Try...Catch...Finally** block with the statement block following the **Try** statement. If Visual Basic completes this block without an exception being generated, it looks for the optional **Finally** statement at the end. If you have supplied the **Finally** statement, Visual Basic executes the statement block following it. In any case, control is then transferred to the statement following the **End Try** statement.

If an exception occurs, Visual Basic examines the **Catch** statements in the order they appear within **Try...Catch...Finally**. If it finds a **Catch** statement that handles the generated exception, it executes the corresponding statement block. When it has finished executing the **Catch** block, it executes the **Finally** block if it is present. Execution then proceeds to the statement following the **End Try** statement.

A **Catch** statement handles an exception that is of the same type as the type declared in the **Catch** statement, or of a type derived from it. If you supply a **Catch** statement that does not specify an exception type, it handles any exception derived from the **Exception** class. If such a statement is the last **Catch** statement, it can catch any exception that you did not handle in the preceding **Catch** blocks.

A **Catch** statement, whether or not it specifies an exception type, can optionally include the **When** clause. **When** is followed by an expression that must be convertible to **Boolean**. The exception is handled only if it is of the right type and the **When** expression evaluates to **True**, as in the following example:

```
Catch Exc As System.NotSupportedException When SupportCheck >= 1
```

If you supply the optional **Finally** statement, the corresponding statement block is always the last code to be executed just before control leaves **Try...Catch...Finally**. This is true even if an unhandled exception occurs, or if you execute an **Exit Try** statement.

You can have any number of **Catch** statements, and you can include or omit a **Finally** statement. However, you must have at least one **Catch** statement or a **Finally** statement.

You cannot branch out of a **Try** block, into a **Catch** block, or into or out of a **Finally** block. You can branch from a **Catch** block into the **Try** block associated with that **Catch**. For example, if one **Try...Catch...Finally** construction is nested within another, a **Catch** block can branch into the **Try** block at its own nesting level, but not into any other **Try** block.

In the following example, **Try...Catch...Finally** attempts to calculate the date and time exactly 100 years from the value provided in the **Object** variable `GivenDate`:

```
Dim GivenDate As Object   ' Should contain date/time information.
Dim NextCentury As Date
Try
   NextCentury = DateAdd("yyyy", 100, GivenDate)
   Catch ThisExcep As System.ArgumentException
   ' At least one argument has an invalid value.
   Catch ThisExcep As ArgumentOutOfRangeException
   ' The result is later than December 31, 9999.
   Catch ThisExcep As InvalidCastException
   ' GivenDate cannot be interpreted As a date/time.
   Catch
   ' An unforeseen exception has occurred.
   Finally
   ' This block is always executed before leaving.
End Try
```

The exceptions that you can anticipate from the **DateAdd** function can be handled in the first three **Catch** blocks, and you can deal with any unexpected exception in the last **Catch** block. No matter what happens, the **Finally** block is always the last code to be executed before leaving **Try...Catch...Finally**.

If `ThisExcep` does not appear in a declaration statement such as **Dim**, the **Catch** statement with the **As** clause serves as a declaration of the exception variable.

See Also

Loop Structures | Nested Control Statements | Try...Catch...Finally Statements (Language Reference) | Throw Statement | If...Then...Else Statements | Select...Case Statements | Exit Statement | Structured Exception Handling | DateAdd Function | System Namespace | Exception Class | ArgumentException Class | ArgumentOutOfRangeException Class | InvalidCastException Class

Loop Structures

Loop structures allow you to execute one or more lines of code repetitively. You can repeat the statements until a condition is true, until a condition is false, a specified number of times, or once for each object in a collection. The loop structures supported by Visual Basic include:

- **While**
- **Do...Loop**
- **For...Next**
- **For Each...Next**

In This Section

While Statement
Describes how to execute a block of statements while a condition is true.
Do...Loop Statements
Describes how to execute a block of statements until a condition is satisfied.
For...Next Statements
Describes how to execute a block of statements a specified number of times.
For Each...Next Statements
Describes how to execute a block of statements for each element in a collection.
Faster For...Next Loops
Explains how to optimize performance of loops.

Related Sections

Decision Structures
Explains control-flow statements used for branching.
With...End With Statements
Describes how to execute several statements on the same object.
Nested Control Statements
Explains control-flow statements inside other control-flow statements.
Exit Statement
Shows how to transfer control directly out of a control-flow statement.

While Statement

You can use the **While** statement to execute a block of statements an indefinite number of times depending on the **Boolean** value of a condition. The statements are repeated as long as the condition is **True**. The condition usually results from a comparison of two values, but it can be any expression that evaluates to a **Boolean** value (**True** or **False**). This includes values of other data types, such as numeric types, that have been converted to **Boolean**.

The **While** statement always checks the condition before it begins the loop. Looping continues while the condition remains **True**.

In the following example, the `CheckWhile` procedure tests the condition before it enters the loop. If `Number` had been initialized to 6 instead of 10, the statements inside the loop would never run.

```
Sub CheckWhile()
   Dim Counter As Integer = 0
   Dim Number As Integer = 10
   While Number > 6
      Number = Number - 1
      Counter = Counter + 1
   End While
   MsgBox("The loop ran " & Counter & " times.")   ' Runs 4 times.
End Sub
```

You can transfer out of a **While** statement by using the **Exit While** statement. One use of this is to test for a condition that could cause an *endless loop*, which is a loop that could run an extremely large or even infinite number of times. If the condition is **True**, use **Exit While** to escape. If the condition is **False**, the loop continues running.

In the following example, `Number` is assigned a value that could cause the loop to execute more than 2 ^ 31 times. The **If** statement checks for this and exits if it exists, preventing endless looping.

```
Sub ExitWhileExample()
   Dim Counter As Integer = 0
   Dim Number As Integer = 8
   While Number <> 10
      If Number < 0 Then Exit While
      Number = Number - 1
      Counter = Counter + 1
   End While
   MsgBox("The loop ran " & Counter & " times.")   ' Runs 9 times.
End Sub
```

Note To stop an endless loop, press ESC or CTRL+BREAK.

See Also

Decision Structures | Nested Control Statements | While...End While Statements | Do...Loop Statements | For...Next Statements | For Each...Next Statements | Exit Statement | Boolean Data Type

Do...Loop Statements

You can use **Do...Loop** statements to execute a block of statements an indefinite number of times depending on the **Boolean** value of a condition. The statements can be repeated either while the condition is **True** or until it becomes **True**. The condition usually results from a comparison of two values, but it can be any expression that evaluates to a **Boolean** value (**True** or **False**). This includes values of other data types, such as numeric types, that have been converted to **Boolean**.

Repeating Statements While a Condition Is True

There are two ways to use the **While** keyword to check a condition in a **Do** loop. You can check the condition before you enter the loop, or you can check it after the loop has run at least once. Looping continues while the condition remains **True**.

In the following example, the `CheckWhileFirst` procedure tests the condition before it enters the loop. If `Number` had been initialized to 6 instead of 10, the statements inside the loop would never run. In the `CheckWhileLast` procedure, the statements inside the loop run once before testing the condition, which is **False** on the first test.

```
Sub CheckWhileFirst()
   Dim Counter As Integer = 0
   Dim Number As Integer = 10
   Do While Number > 6
      Number = Number - 1
      Counter = Counter + 1
   Loop
   MsgBox("The loop ran " & Counter & " times.")   ' Runs 4 times.
End Sub
   ' ...
Sub CheckWhileLast()
   Dim Counter As Integer = 0
   Dim Number As Integer = 5
   Do
      Number = Number - 1
      Counter = Counter + 1
   Loop While Number > 6
   MsgBox("The loop ran " & Counter & " times.")   ' Runs 1 time.
End Sub
```

Repeating Statements Until a Condition Becomes True

There are two ways to use the **Until** keyword to check a condition in a **Do** loop. You can check the condition before you enter the loop, or you can check it after the loop has run at least once. Looping continues while the condition remains **False**.

In the following example, the `CheckUntilFirst` procedure tests the condition before it enters the loop. If `Number` had been initialized to 15 instead of 20, the statements inside the loop would never run. In the `CheckUntilLast` procedure, the statements inside the loop run once before testing the condition, which is **False** on the first test.

```
Sub CheckUntilFirst()
   Dim Counter As Integer = 0
   Dim Number As Integer = 20
   Do Until Number = 15
      Number = Number - 1
      Counter = Counter + 1
   Loop
   MsgBox("The loop ran " & Counter & " times.")   ' Runs 5 times.
End Sub
   ' ...
Sub CheckUntilLast()
   Dim Counter As Integer = 0
   Dim Number As Integer = 20
   Do
      Number = Number - 1
      Counter = Counter + 1
   Loop Until Number = 15
   MsgBox("The loop ran " & Counter & " times.")   ' Runs 5 times.
End Sub
```

Exiting a Do Loop from Inside the Loop

You can transfer out of a **Do** loop by using the **Exit Do** statement. One use of this is to test for a condition that could cause an *endless loop*, which is a loop that could run an extremely large or even infinite number of times. If the condition is **True**, use **Exit Do** to escape. If the condition is **False**, the loop continues running.

In the following example, `Number` is assigned a value that could cause the loop to execute more than 2 ^ 31 times. The **If** statement checks for this and exits if it exists, preventing endless looping.

```
Sub ExitDoExample()
   Dim Counter As Integer = 0
   Dim Number As Integer = 8
   Do Until Number = 10
      If Number <= 0 Then Exit Do
      Number = Number - 1
      Counter = Counter + 1
   Loop
   MsgBox("The loop ran " & Counter & " times.")   ' Runs 8 times.
End Sub
```

Note To stop an endless loop, press ESC or CTRL+BREAK.

Replacing the Continue Statement

Visual Basic .NET does not support the **Continue** statement of previous versions of Visual Basic. However, you can achieve the same functionality by putting a statement label on the **Loop** statement and branching to it from the middle of the loop:

```
Dim LoopCounter As Integer = 0
   Do While LoopCounter < 100
      LoopCounter += 1
      Dim SkipToNextIteration As Boolean   ' Local to this loop.
      ' Processing, which might change value of SkipToNextIteration.
      If SkipToNextIteration = True Then GoTo EndOfLoop
      ' More processing if SkipToNextIteration was still False.
EndOfLoop: Loop ' Acts like Continue.
```

See Also

Decision Structures | Nested Control Statements | Do...Loop Statements (Language Reference) | While Statement | For...Next Statements | For Each...Next Statements | Exit Statement | Labeling Statements | Boolean Data Type | Until

For...Next Statements

Do loops work well when you do not know in advance how many times you need to execute the statements in the loop. However, when you expect to execute the loop a specific number of times, a **For...Next** loop is a better choice. Unlike a **Do** loop, a **For** loop uses a variable called a *counter* that increases or decreases in value during each repetition of the loop. The syntax is as follows:

```
For counter [ As datatype ] = start To end [ Step step ]
   ' Statement block to be executed for each value of counter.
Next [ counter ]
```

The variable *counter* must be of a numeric data type that supports the greater than or equal (**>=**), less than or equal (**<=**), and addition (**+**) operators; usually it is **Integer**. The iteration values *start, end,* and *step* are expressions that must evaluate to data types that widen to the type of *counter*. Usually they are also **Integer**. The optional *step* can be positive or negative. If it is omitted, it is taken to be 1.

If *counter* is not declared outside the loop, you can use the **As** clause to declare it as part of the **For** statement.

When execution of the **For...Next** loop begins, Visual Basic evaluates *start, end,* and *step*. It then assigns *start* to *counter*. Before it executes the statement block, it compares *counter* to *end*. If *counter* is already past the end value, the **For** loop terminates and control passes to the statement following the **Next** statement. Otherwise the statement block is executed.

Each time Visual Basic encounters the **Next** statement, it increments the counter by *step* and returns to the **For** statement. Again it compares the counter to *end*, and again it either executes the block or terminates the loop depending on the result. This process continues until the counter passes *end* or an **Exit For** statement is executed.

The loop does not terminate until *counter* has passed the end value. If *counter* is equal to *end*, the loop continues. The comparison that determines whether to execute the block is *counter* **<=** *end* if *step* is positive and *counter* **>=** *end* if *step* is negative.

You can optionally specify *counter* in the **Next** statement. This improves the readability of your program. You must specify the same variable as the one that appears in the **For** statement.

The following procedure presets all the elements of an array to 128. The **For** statement specifies the counter variable `I` and its *start* and *end* values. The **Next** statement increments the counter by 1, because *step* is not supplied.

```
Sub Preset(ByRef A() As Integer)
   Dim I As Integer
   For I = 0 To UBound(A)
      A(I) = 128
   Next I
End Sub
```

The iteration values *start*, *end*, and *step* are evaluated only once, before the loop begins. If your statement block changes *end* or *step*, these changes do not affect the iteration of the loop. In the preceding example, the **UBound** function is called only when the **For** statement is first executed. If the statement block had subsequently changed the length of the array `A`, the **For** loop would still use the original length as the end value, because it does not call **UBound** again.

Incrementing and Decrementing the Counter Variable

Using the **Step** keyword, you can increment or decrement the counter by the value you specify. In the following example, the counter variable `J` is incremented by 2 each time the loop repeats. When the loop is finished, `Total` is the sum of 2, 4, 6, 8, and 10.

```
Sub TwosTotal()
   Dim J, Total As Integer
   For J = 2 To 10 Step 2
      Total = Total + J
   Next J
   MsgBox("The total is " & Total)
End Sub
```

To decrease the counter variable, use a negative **Step** value. When you do this, you should specify an end value that is less than the start value. In the following example, the counter variable `N` is decreased by 2 each time the loop repeats. When the loop is finished, `Total` is the sum of 16, 14, 12, 10, 8, 6, and 4.

```
Sub NewTotal()
   Dim N, Total As Integer
   For N = 16 To 4 Step -2
      Total = Total + N
   Next N
   MsgBox("The total is " & Total)
End Sub
```

Exiting a For...Next Loop Before the Counter Passes Its End Value

You can exit a **For...Next** loop before the counter passes its end value by using the **Exit For** statement. For example, you might want to exit a loop if you detect a condition that makes it unnecessary or impossible to continue iterating, such as an erroneous value or a termination request. Also, if you catch an exception in a **Try...Catch...Finally**, you can use **Exit For** at the end of the **Finally** block.

See Also

Decision Structures | Nested Control Statements | For...Next Statements (Language Reference) | Do...Loop Statements | While Statement | For Each...Next Statements | Exit Statement | Data Types | Integer Data Type | UBound Function | Step

For Each...Next Statements

The **For Each...Next** loop is similar to the **For...Next** loop, but it executes the statement block for each element in a collection, instead of a specified number of times. The syntax is as follows:

```
For Each elementvariable [ As datatype ] In collection
   ' Statement block to be executed for each value of elementvariable.
Next [ elementvariable ]
```

The elements of *collection* can be of any data type. The data type of *elementvariable* must be such that each element of the collection can be converted to it.

If *elementvariable* is not declared outside the loop, you can use the **As** clause to declare it as part of the **For Each** statement.

For each iteration of the loop, Visual Basic sets the variable *elementvariable* to one of the elements in the collection and executes the statement block. When all the elements in the collection have been assigned to *elementvariable*, the **For Each** loop terminates and control passes to the statement following the **Next** statement.

You can optionally specify the element variable in the **Next** statement. This improves the readability of your program. You must specify the same variable as the one that appears in the **For Each** statement.

The following procedure accepts a form and sets the background color of every control in that form to light blue:

```
Sub LightBlueBackground(ByVal ThisForm As System.Windows.Forms.Form)
   Dim ThisControl As System.Windows.Forms.Control
   For Each ThisControl In ThisForm.Controls
      ThisControl.BackColor = System.Drawing.Color.LightBlue
   Next ThisControl
End Sub
```

The collection must be an object that implements the **IEnumerable** interface of the **System.Collections** namespace. **IEnumerable** defines the **GetEnumerator** method, which returns an enumerator object for the collection. The enumerator object exposes the **Current** property and the **MoveNext** method. Visual Basic uses these to traverse the collection. The data type of the elements returned by **Current** must be convertible to the data type of *elementvariable*.

Modifying Elements

The **Current** property of the enumerator object is **ReadOnly**, and it returns a local copy of each collection element. This means that you cannot modify the elements themselves in a **For Each...Next** loop. Any modification you make affects only the local copy from **Current** and is not reflected back into the collection. However, if an element is a reference type, you can modify the members of the instance to which it points. This is why, in the previous example, you can modify the `BackColor` member of each `ThisControl` element, although you cannot modify `ThisControl` itself.

Accessing Arrays

Because **System.Array** implements the **IEnumerable** interface, all arrays expose the **GetEnumerator** method. This means you can iterate through an array with a **For Each...Next** loop. However, you can only read the array elements, not change them. The following procedure is intended to find the sum of the elements of an array and reset each element to zero:

```
Function SumAndReset(ByRef A() As Integer) As Integer
   Dim Elt As Integer
   Dim Sum As Integer = 0
   For Each Elt In A
      Sum += Elt
      Elt = 0
   Next Elt
   Return Sum
End Function
```

The summation works as intended because it relies only on reading the elements. However, the resetting to zero does not work as intended, because only the local copy is reset to zero, while the original element is left unchanged in the collection.

Altering the Collection

The enumerator object does not allow you to alter the collection by adding, deleting, replacing, or reordering any elements. If you alter the collection after you have initiated a **For Each...Next** loop, the enumerator object becomes invalid, and the next attempt to access an element results in an **InvalidOperationException**.

Traversing the Collection

When you execute a **For Each...Next** loop, traversal of the collection is under the control of the enumerator object returned by the **GetEnumerator** method. The order of traversal is not determined by Visual Basic, but rather by the **MoveNext** method of the enumerator object. This means that you might not be able to predict which element of the collection is the first to be returned in *elementvariable*, or which is the next to be returned after a given element.

If your code depends on traversing a collection in a particular order, a **For Each...Next** loop is not the best choice unless you know the characteristics of the enumerator object the collection exposes. You might achieve more reliable results using a different loop, such as **For...Next** or **Do...Loop**.

Exiting a For Each...Next Loop Before It Is Finished with the Collection

You can exit a **For Each...Next** loop before it has traversed the collection by using the **Exit For** statement. For example, you might want to exit a loop if you detect a condition that makes it unnecessary or impossible to continue iterating, such as an erroneous value or a termination request. Also, if you catch an exception in a **Try...Catch...Finally**, you can use **Exit For** at the end of the **Finally** block.

See Also

Decision Structures | Nested Control Statements | For Each...Next Statements (Language Reference) | Do...Loop Statements | While Statement | For...Next Statements | Exit Statement | Try...Catch...Finally Statements | Form Class | Control Class | Color Structure | IEnumerable Interface | IEnumerable.GetEnumerator Method | IEnumerator.Current Property | Array Class | Array.GetEnumerator Method

Faster For...Next Loops

Integer variables are slightly faster to update than either **Short** or **Long** ones. This difference is noticeable only if you perform many thousands of operations. However, the difference between **Integer** and **Single** or **Decimal** is more readily noticeable. The following example shows some possible variations:

```
Dim Fastest As Integer   ' First case -- Integer is fastest.
For Fastest = 0 to 100000
   ' Statements to execute for each value of Fastest.
Next Fastest
   ' ...
Dim NotAsFast As Long   ' Second case -- Long is not as fast.
For NotAsFast = 0 to 100000
   ' Statements to execute for each value of NotAsFast.
Next NotAsFast
   ' ...
Dim MuchSlower As Decimal   ' Third case -- Decimal is slower.
For MuchSlower = 0 to 100000
   ' Statements to execute for each value of MuchSlower.
Next MuchSlower
```

The first case above takes slightly less time to run than the second case. However, an error occurs if `Fastest` exceeds 2,147,483,647. The second or third cases fix this, because both **Long** and **Decimal** types accept a wider range of integers, but they run more slowly. For data types that use 4 or more bytes, smaller types usually take less time to manipulate than larger ones.

See Also

Decision Structures | Nested Control Statements | For...Next Statements | For Each...Next Statements | Exit Statement | Integer Data Type | Short Data Type | Long Data Type | Single Data Type | Decimal Data Type

With...End With Statements

In Visual Basic you usually must specify an object in every statement that calls one of its methods or accesses one of its properties. However, if you have a series of statements that all operate on the same object, you can use a **With...End With** statement to specify the object once for all of the statements. Doing this can make your procedures run faster and help you avoid repetitive typing.

The following example sets the foreground color and font style of a label depending on the value of a procedure argument.

```
Dim AlertLabel As New System.Windows.Forms.Label
   ' ...
Sub AlertUser(ByVal Value As Long)
   With AlertLabel
      If Value = 0 Then
         .ForeColor = Color.Red
         .Font = New Font(.Font, FontStyle.Bold Or FontStyle.Italic)
      Else
         .Forecolor = Color.Black
         .Font = New Font(.Font, FontStyle.Regular)
      End If
   End With
End Sub
```

You can nest **With...End With** statements for greater efficiency. The following example inserts a formula into cell A1, and then formats the font.

```
Sub MyInput()
   With Workbooks("Book1").Worksheets("Sheet1").Cells(1, 1)
      .Formula = "=SQRT(50)"
      With .Font
         .Name = "Arial"
         .Bold = True
          .Size = 8
      End With
   End With
End Sub
```

See Also

Decision Structures | Loop Structures | With...End With Statements (Language Reference) | Nested Control Statements

Nested Control Statements

You can place control statements inside other control statements, for example an **If...Then...Else** block within a **For...Next** loop. A control statement placed inside another control statement is said to be *nested*.

Control statements in Visual Basic can be nested to as many levels as you want. It is common practice to make nested decision structures and loops more readable by indenting the body of each one.

In the following example, the procedure adds together the positive elements of each row of the matrix:

```
Public Sub SumRows(ByVal A(,) As Double, ByRef R() As Double)
   Dim I, J As Integer
   For I = 0 To UBound(A, 1)
   R(I) = 0
      For J = 0 To UBound(A, 2)
       If A(I, J) > 0 Then
         R(I) = R(I) + A(I, J)
       End If
      Next J
   Next I
End Sub
```

Notice that the first **Next** closes the inner **For** loop and the last **Next** closes the outer **For** loop. Likewise, in nested **If** statements, the **End If** statements automatically apply to the nearest prior **If** statement. Nested **Do** loops work in a similar fashion, with the innermost **Loop** statement matching the innermost **Do** statement.

See Also

Decision Structures | Loop Structures | With...End With Statements | Exit Statement

Exit Statement

The **Exit** statement allows you to exit directly from any decision structure, loop, or procedure. It immediately transfers execution to the statement following the last control statement. The syntax for the **Exit** statement specifies which type of control statement you are transferring out of. The following versions of the **Exit** statement are possible:

- **Exit Select**
- **Exit Try**
- **Exit Do**
- **Exit While**
- **Exit For**

You can also exit directly from a **Function**, **Sub**, or **Property** procedure; the syntax is similar to that of **Exit For** and **Exit Do**:

- **Exit Sub**
- **Exit Function**
- **Exit Property**

Exit Sub, **Exit Function**, and **Exit Property** can appear as many times as needed, anywhere within the body of the procedure. They can even appear inside a control statement such as **If...Then...Else**. These statements are useful when a procedure has done everything it needs to do and can return immediately.

Exiting from Nested Control Statements

If an **Exit** statement is encountered within nested control statements, execution continues with the statement following the end of the innermost control statement of the kind specified in the **Exit** statement. In the following example, the **Exit For** statement is located in the inner **For** loop, so it passes control to the statement following that loop and continues with the outer **For** loop.

```
Public Sub InvertElements(ByRef A(,) As Double)
   Dim I, J As Integer
   For I = 0 To UBound(A, 1)
      For J = 0 To UBound(A, 2)
       If A(I, J) = 0 Then
         Exit For     ' Cannot complete this row.
       Else
         A(I, J) = 1 / A(I, J)
       End If
      ' Control comes here if the Exit For is executed.
      Next J
   Next I
End Sub
```

Multiple Exit Statements

The **Exit** statement in any of its forms can appear as many times as needed inside the appropriate loop or statement. The following example shows this inside a **Do** loop:

```
Dim x, y, z As Double
Do Until y = -1
   If x < 0 Then Exit Do
   x = Math.Sqrt(x)
   If y > 0 Then Exit Do
   y = y + 3
   If z = 0 Then Exit Do
   z = x / z
Loop
```

The **Exit Do** statement works with all versions of **Do** loop syntax (with **While** or **Until**), and **Exit For** works with all versions of **For** loop syntax (with or without **Each**).

See Also

Decision Structures | Loop Structures | With...End With Statements | Nested Control Statements |
Exit Statement (Language Reference) | Sub Procedures | Function Procedures | Property Procedures

Object-Oriented Programming in Visual Basic

Objects are central to Visual Basic programming. Forms and controls are objects. Databases are objects. If you've used Visual Basic for a while, or if you've worked through the examples in this Help file, then you've already programmed with objects, but there's a lot more to objects than what you've seen so far.

In the following topics, you'll see how easy it is to create your own objects from the classes you define, and to use objects to simplify your coding and increase code reuse.

In This Section

Introduction to Objects in Visual Basic
Provides an introduction to the terms and concepts used in object-oriented programming.
Early and Late Binding
Describes binding, which is performed by the compiler when an object is assigned to an object variable, and the differences between early-bound and late-bound objects.
Shared Members
Describes members that can be used without instances.
Creating and Using Objects
Shows how to create and use instances of classes.
Managing Groups of Objects
Provides techniques for working with arrays and collections of objects.
Obtaining Class Information at Run Time
Shows how to determine the class that an object belongs to.
Understanding Classes
Covers object creation and lifetime issues in a step-by-step procedure.
Events and Delegates
Shows how to declare and use events and delegates.
Interfaces in Visual Basic .NET
Describes what interfaces are and how you can use them in your applications.
Inheritance
Describes how to define classes that serve as the basis for other classes.

Related Sections

Objects

Describes what objects are and how they are used.

Programming with Components

Demonstrates how you can use Visual Basic to control objects provided by other applications.

Introduction to Objects in Visual Basic

Almost everything you do in Visual Basic is associated with objects. If you are new to object-oriented programming, the following terms and concepts will help you get started.

Classes and Objects

The words "class" and "object" are used so much in object-oriented programming that it is easy to get the terms mixed up. Generally speaking, a *class* is an abstract representation of something, whereas an *object* is a usable example of the thing the class represents. The one exception to this rule is shared class members, which are usable in both instances of a class and object variables declared as the type of the class.

Fields, Properties, Methods, and Events

Classes are made of fields, properties, methods, and events. Fields and properties represent information that an object contains. Fields are like variables in that they can be read or set directly. For example, if you have an object named Car you could store its color in a field named Color.

Properties are retrieved and set like fields, but are implemented using **Property Get** and **Property Set** procedures, which provide more control on how values are set or returned. The layer of indirection between the value being stored and the procedures that use this value helps isolate your data and allows you to validate values before they are assigned or retrieved.

Methods represent actions that an object can perform. For example, a Car object could have StartEngine, Drive, and Stop methods. You define methods by adding procedures, either **Sub** routines or functions, to your class.

Events are notifications an object receives from, or transmits to, other objects or applications. Events allow objects to perform actions whenever a specific occurrence takes place. An example of an event for the Car class would be a Check_Engine event. Because Microsoft Windows is an event-driven operating system, events can come from other objects, applications, or user input such as mouse clicks or key presses.

Encapsulation, Inheritance, and Polymorphism

Fields, properties, methods, and events are only one half of the object-oriented programming equation. True object-orient programming requires objects to support three qualities: encapsulation, inheritance, and polymorphism.

Encapsulation means that a group of related properties, methods, and other members are treated as a single unit or object. Objects can control how properties are changed and methods are executed. For example, an object can validate values before allowing property changes. Encapsulation also makes it easier to change your implementation at a latter date by letting you hide implementation details of your objects, a practice called *data hiding*.

Inheritance describes the ability to create new classes based on an existing class. The new class inherits all the properties and methods and events of the base class, and can be customized with additional properties and methods. For example, you can create a new class named Truck based on the Car class. The Truck class inherits the Color property from the Car class and can have additional properties such as FourWheelDrive.

Polymorphism means that you can have multiple classes that can be used interchangeably, even though each class implements the same properties or methods in different ways. Polymorphism is essential to object-oriented programming because it allows you to use items with the same names, no matter what type of object is in use at the moment. For example, given a base class of Car, polymorphism enables the programmer to define different StartEngine methods for any number of derived classes. The StartEngine method of a derived class named DieselCar may be completely different from the method with the same name in the base class. Other procedures or methods can use the StartEngine method of the derived classes in exactly the same way, no matter what type of Car object is being used at the time.

Overloading, Overriding, and Shadowing

Overloading, overriding, and shadowing are similar concepts that can be easy to confuse. Although all three techniques allow you to create members with the same name, there are some important differences.

- Overloaded members are used to provide different versions of a property or method that have the same name, but that accept different number of parameters, or parameters with different data types.
- Overridden properties and methods are used to replace an inherited property or method that is not appropriate in a derived class. Overridden members must accept the same data type and number of arguments. Derived classes inherit overridden members.
- Shadowed members are used to locally replace a member that has broader scope. Any type can shadow any other type. For

example, you can declare a property that shadows an inherited method with the same name. Shadowed members cannot be inherited.

See Also

Objects | Creating and Using Objects | Classes: Blueprints for Objects | Inheritance | Overloaded Properties and Methods | Overriding Properties and Methods | Shadowing

Early and Late Binding

The Visual Basic compiler performs a process called *binding* when an object is assigned to an object variable. An object is *early bound* when it is assigned to a variable declared to be of a specific object type. Early bound objects allow the compiler to allocate memory and perform other optimizations before an application executes. For example, the following code fragment declares a variable to be of type **FileStream**:

```
' Add Imports statements to the top of your file.
Imports System.IO
'...
'  Create a variable to hold a new object.
   Dim FS As FileStream
  ' Assign a new object to the variable.
   FS = New FileStream("C:\tmp.txt", FileMode.Open)
```

Because **FileStream** is a specific object type, the instance assigned to `FS` is early bound.

By contrast, an object is *late bound* when it is assigned to a variable declared to be of type **Object**. Objects of this type can hold references to any object, but lack many of the advantages of early-bound objects. For example, the following code fragment declares an object variable to hold an object returned by the **CreateObject** function:

```
' To use this example, you must have Microsoft Excel installed on your computer.
Option Strict Off ' Option Strict Off allows late binding.
...
Sub TestLateBinding()
   Dim xlApp As Object
   Dim xlBook As Object
   Dim xlSheet As Object
   xlApp = CreateObject("Excel.Application")
   'Late bind an instance of an Excel workbook.
   xlBook = xlApp.Workbooks.Add
   'Late bind an instance of an Excel worksheet.
   xlSheet = xlBook.Worksheets(1)
   xlSheet.Activate()
   xlSheet.Application.Visible = True ' Show the application.
   ' Place some text in the second row of the sheet.
   xlSheet.Cells(2, 2) = "This is column B row 2"
End Sub
```

You should use early-bound objects whenever possible, because they allow the compiler to make important optimizations that yield more efficient applications. Early-bound objects are significantly faster than late-bound objects and make your code easier to read and maintain by stating exactly what kind of objects are being used. Another advantage to early binding is that it enables useful features such as automatic code completion and Dynamic Help because the Visual Studio .NET integrated development environment (IDE) can determine exactly what type of object you are working with as you edit the code. Early binding reduces the number and severity of run-time errors because it allows the compiler to report errors when a program is compiled.

> **Note** Late binding can only be used to access type members that are declared as **Public**. Accessing members declared as **Friend** or **Protected Friend** results in a runtime error.

See Also

Object Lifetime: How Objects Are Created and Destroyed | Object Data Type | CreateObject Function

Shared Members

Shared members are properties, procedures, and fields that are shared by all instances of a class. Some programming languages refer to such items as *static members*.

Shared fields and properties are useful when you have information that is part of a class, but is not specific to any one instance of a class. Normal fields and properties exist independently for each instance of a class. Changing the value of a field or property associated with any one instance does not affect the value of fields or properties of other instances of the class. On the other hand, when you change the value of a shared field and property associated with an instance of a class, you change the value associated with all instances of the class. In this way, shared fields and properties behave like global variables that can be accessed only from instances of a class. Without static fields and properties, you would need to use module-level variables to achieve the same effect. However, module-level variables can make your classes difficult to understand and maintain. Furthermore, using module-level variables in this way violates the concept of encapsulation that classes represent.

Shared procedures are class methods that are not associated with a specific instance of a class. For example, the **Cos** method defined within the **Math** class is a shared method. You can call a shared procedure by calling it either as a method of an object, or directly from the class. In this way, shared procedures represent an exception to the rule that you must create an instance of a class before you can use it.

Shared procedures are not implicitly passed instances of the class. For this reason, no unqualified references to non-shared data members are allowed in shared methods.

Shared Members Example

The following example creates an instance field, a shared field, and a shared method to demonstrate how shared members operate in code:

```
Public Class ShareClass
   Public InstanceValue As String
   Public Shared SharedValue As String
   Public Shared Sub ShareMethod()
      MsgBox("This is a shared method.")
   End Sub
End Class
Sub TestShared()
   Dim Shared1 As New ShareClass()   ' Create an instance of the class.
   Dim Shared2 As New ShareClass()   ' Create an instance of the class.
   Shared1.SharedValue = "Share Value 1"    ' Set the value of a shared field.
   Shared2.SharedValue = "Share Value 2"   ' Overwrite the first value.
   MsgBox("The value of the shared field in the first instance" & _
      "is: " & Shared1.SharedValue)
   MsgBox("The value of the shared field in the second instance" & _
      "is: " & Shared2.SharedValue)
   ' Call a method on the class without creating an instance.
   ShareClass.ShareMethod()
End Sub
```

When you execute the `TestShared` procedure, two instances of the class are created, and the shared field `SharedValue` is modified for both instances. When the shared field in the second instance of the class is modified, the value assigned to the shared field in the first instance of the class is overwritten because both instances refer to the same field.

For the sake of clarity, this example uses object variables to access shared members, but it is better programming practice to access shared members directly using the class name; for example, `ShareClass.SharedValue = "Share Value 2"`. Using class syntax for shared members reduces confusion, because it makes it clear when you are using shared members and when you are using instance members.

See Also

Class Properties, Fields, and Methods | Structures: Your Own Data Types | Procedures | Structures and Classes | Dim Statement | Math.Cos Method

Creating and Using Objects

Objects are combinations of code and data that can be treated as single units. Nearly everything you do in Visual Basic .NET involves objects, from the controls you use to the forms on which you place them. This section discusses where objects originate and how to use them.

In This Section

Objects from Visual Basic and Other Sources
Discusses how to use objects from Microsoft Word, Microsoft Excel, and other applications.
Setting and Retrieving Properties
Discusses how to change and obtain property values.
Performing Actions with Methods
Describes what methods are and how to use them.
Performing Multiple Actions on an Object
Covers using the **With...End With** statement to simplify property access.
Treating Forms as Objects
Shows how to use forms in the same way as ordinary objects.
Using the New Keyword
Shows how to create instances of forms, classes defined in class modules, and collections.
Resource Management
Discusses how to save memory and system resources by releasing references to objects.
Passing Objects to Procedures
Describes how to pass objects as arguments to procedures.

Related Sections

Objects
Covers object-oriented programming fundamentals.
Managing Groups of Objects
Lists techniques for working with arrays and collections of objects.
Components and Assemblies
Describes how to create and use components.

Objects from Visual Basic and Other Sources

The objects you use in Visual Basic .NET come from both internal and external sources. Examples of internal objects include intrinsic objects and classes in your project; examples of external objects include assemblies and COM objects.

Internal Objects

Intrinsic (or *built-in*) objects are objects that Visual Basic .NET natively provides. These include the primitive scalar types such as **Integer** and **Double** as well as the **Array** and **String** types. You do not need to create references to internal objects before using them in your project.

Other internal objects are instances of classes in your current project. You can use these classes wherever necessary within your project, and you can make them available for use by other applications when you create an assembly.

External Objects

External objects are those that come from other projects or assemblies that are not available to your project by default. You must create project references to external objects before using them in your project.

Assemblies are the most common source of objects for Visual Basic .NET applications. The .NET Framework includes assemblies that contain commonly used objects. Some objects in the .NET Framework are treated as intrinsic objects, but most assemblies must be explicitly imported using the **Imports** statement before they can be used. Any language compliant with the Common Language Specification (CLS), such as Visual Basic .NET or Visual C#, can create and use assemblies. For details, see Assemblies.

In the past, COM components were a traditional source of objects for Visual Basic programmers, but today .NET assemblies are a better choice for new objects. You can still use existing COM components in your applications; however, COM objects must be accessed via .NET interoperability classes. Accessing a COM library requires the use of an interoperability assembly containing interoperability classes for each COM class defined within the COM library.

In addition to accessing native .NET classes and COM classes, it is also possible to call functions defined in dynamic-link libraries (DLL's). Visual Basic .NET allows you to call functions in DLL's after they have been declared with a **Declare** statement. Visual Basic .NET adds functionality to the **Declare** statement by allowing you use the **DllImport** attribute to specify defaults for things such as **CallingConvention**, **ExactSpelling,** and **SetLastError**. Parameters for **Declare** statements can be annotated with the **MarshalAs** attribute, which supports conversion of parameters in ways that were not possible in earlier versions of Visual Basic.

See Also

The Basics of Working with Objects | Relation of Objects to Each Other |
Walkthrough: Implementing Inheritance with COM Objects | Setting and Retrieving Properties | Creating and Using Objects |
Declare Statement | Imports Statement | DllImportAttribute Class | Integer Data Type | Double Data Type | String Data Type |
MarshalAsAttribute Class

Setting and Retrieving Properties

When working with forms and controls in Visual Basic .NET, you can set their properties programmatically at run time, or you can set them in design mode using the **Properties** window. The properties of most other objects, such as objects from assemblies or objects you create, can only be set programmatically.

Properties you can set and read are called *read-write* properties. Properties you can read but not modify are called *read-only* properties. Properties you can write but not read are called *write-only* properties.

You set the value of a property when you want to change the appearance or behavior of an object. For example, you change the **Text** property of a text box control to modify the contents of the text box.

You get the value of a property when you want to find the state of an object before your code performs additional actions, such as assigning the value to another object. For example, you can return the **Text** property of a text box control to determine the contents of the text box before running code that might change the value.

To set property values

- Use the following syntax:

```
object.property = expression
```

The following statements provide examples of how to set properties:

```
TextBox1.Top = 200   ' Sets the Top property to 200 twips.
TextBox1.Visible = True   ' Displays the text box.
TextBox1.Text = "hello"   ' Displays 'hello' in the text box.
```

Note You can also set a property by passing it to **ByRef** parameters, in which case the property is modified by the result returned by the **ByRef** parameter.

To get property values

- Use the following syntax:

```
variable = object.property
```

You can also get a property value as part of a more complex expression, without assigning the property to a variable. The following code changes the **Top** property of a radio button control:

```
Protected Sub RadioButton1_CheckedChanged(ByVal sender As Object, _
      ByVal e As System.EventArgs)
   ' [statements]
   RadioButton1.Top += 20
   ' [statements]
End Sub
```

See Also

The Basics of Working with Objects | Performing Actions with Methods | Objects from Visual Basic and Other Sources | Creating and Using Objects

Performing Actions with Methods

Methods are procedures associated with objects. Unlike fields and properties, which represent information an object can store, methods represent actions that an object can perform. Methods can affect the values of properties. For example, to use a radio analogy, you can employ a `SetVolume` method to change the value of a `Volume` property. Similarly, in Visual Basic .NET, the items of list boxes have a **List** property, which you can change with the **Clear** and **Add** methods.

When you use a method in code, the way you write the statement depends on how many arguments the method requires and whether it returns a value. Generally, you use methods just like you use subroutines or function calls. More specifically, you invoke methods in the same way as module procedures, except that you can qualify methods with an expression specifying the object instance whose method is to be called. When unqualified, the instance is implicitly the **Me** variable.

To use a method that does not require arguments

- Use the following syntax:

```
object.method()
```

In the following example, the **Refresh** method repaints the picture box:

```
PictureBox1.Refresh()   ' Forces a repaint of the control
```

Note Some methods, such as **Refresh**, do not have arguments and do not return values.

To use a method that requires multiple arguments

- Place the arguments in parentheses and separate them with commas. In the following example, the **MsgBox** method uses arguments that specify the message to display and the style of the message box:

```
MsgBox("Database update complete", _
       MsgBoxStyle.OKOnly Or MsgBoxStyle.Exclamation, _
       "My Application")
```

To use a method that returns a value

- Assign the return value to a variable, or directly use the method call as a parameter to another call. The following code stores the return value:

```
Dim Response As MsgBoxResult
Response = MsgBox("Do you want to exit?", _
                  MessageBoxButtons.YesNo Or MsgBoxStyle.Question, _
                  "My Application")
```

This example uses the value returned from the Len method as an argument to MsgBox.

```
Dim MyStr As String = "Some String"
'   Displays "String length is : 11
MsgBox("String length is : " & Len(MyStr))
```

See Also

The Basics of Working with Objects | Relation of Objects to Each Other | Setting and Retrieving Properties | Creating and Using Objects

Performing Multiple Actions on an Object

Often, you need to perform several different actions on the same object. For example, you may need to set several properties or execute several methods for the same object.

To set multiple properties for the same object

- One way to do this is to write several statements using the same object variable, as in the following code:

```
Private Sub UpdateForm()

   Button1.Text = "OK"

   Button1.Visible = True

   Button1.Top = 24

   Button1.Left = 100

   Button1.Enabled = True

   Button1.Refresh()

End Sub
```

However, you can make this code easier to write and read by using the **With...End With** statement, as in the following code:

```
Private Sub UpdateForm2()

   With Button1

      .Text = "OK"

      .Visible = True

      .Top = 24

      .Left = 100

      .Enabled = True

      .Refresh()

   End With

End Sub
```

You can also nest **With...End With** statements by placing one inside another, as in the following code:

```
Sub SetupForm()
   Dim AnotherForm As New Form1()
   With AnotherForm
      .Show()   ' Show the new form.
      .Top = 250
      .Left = 250
      .ForeColor = Color.LightBlue
      .BackColor = Color.DarkBlue
      With AnotherForm.Textbox1
         .BackColor = Color.Thistle   ' Change the background.
         .Text = "Some Text"   ' Place some text in the text box.
      End With
   End With
End Sub
```

Within the nested **With** statement, however, the syntax refers to the nested object; properties of the object in the outer **With** statement are not set.

See Also

Calling a Property or Method Using a String Name

Treating Forms as Objects

Forms are graphical objects that make up your applications user interface. Within Visual Basic, classes define how forms are displayed and what they can do. When a form is displayed at runtime, Visual Basic .NET creates an instance of the **Form** class that you can use like any other object. You can add custom methods and properties to forms and access them from other forms or classes in your application.

To create a new method for a form

- Add a procedure declared as **Public**, as in the following code:

```
' Create a custom method on a form.
Public Sub PrintMyJob()
   '.
   '.Add code for your method here.
   '.
End Sub
```

To add a new field to a form

- Declare a public variable in the form module, as in the following code:

```
Public IDNumber As Integer
```

To access methods on a different form

1. Create a new instance of the form whose methods you want to access. When you reference a form name, you are actually referencing the class the form belongs to, not the object itself.

   Note Visual Basic 6 provided, for each form class, an implicit global variable with the same name as the form class. Visual Basic .NET does not provide an implicit variable declaration.

2. Assign the form to an object variable. The object variable references a new instance of the form class.

   The following example correctly calls the `PrintMyJob` procedure:

```
Dim newForm1 As New Form1
newForm1.PrintMyJob
```

In the previous example, the new form is not displayed. It is not necessary to show a form object to use its methods. To display the new form, you need to add the following code:

```
newForm1.show
```

See Also

Property Procedures | Using the New Keyword

Using the New Keyword

To create an instance of a class, use the **New** keyword. Unlike value types, such as **Integer** and **Double**, objects are reference types, and you must explicitly create them before you can use them. For example, consider the following two lines of code:

```
Dim Button1 As System.Windows.Forms.Button
```

```
Dim Button2 As New System.Windows.Forms.Button()
```

The first statement declares an object variable that can contain a reference to a button object. However, the variable `Button1` contains the value **Nothing** until you assign an object of type **Button** to it. The second statement also defines a variable that can contain a button object, but the **New** keyword creates a button object and assigns it to the variable `Button2`.

Because forms and controls are actually classes, you can use the **New** keyword to create new instances of these items as needed.

To create new instances of a class with New

1. Open a new Windows Forms project and place a command button and several other controls on a form named `Form1`.
2. Add the following code to the command button's **Click** event procedure:

```
Dim f As New Form1
f.Show
```

3. Run the application, and click the command button several times.
4. Move the front form aside. Because a form is a class with a visible interface, you can see the additional copies. Each copy has the same controls, in the same positions as those on the original form at design time.

You can use the **New** keyword to create objects from within classes. The following procedure provides an example.

To see how New creates instances of a class

1. Open a new project, and place a command button on a form named `Form1`.
2. From the **Project** menu, choose **Add Class** to add a class to the project.
3. Name the new class `ShowMe.vb`.
4. Add the following procedure to `ShowMe`:

```
Public Class ShowMe
   Sub ShowFrm()
      Dim frmNew As Form1
      frmNew = New Form1
      frmNew.Show()
      frmNew.WindowState = 1
   End Sub
End Class
```

5. Add the following code to the **Button1_Click** event of the button on your form:

```
Protected Sub Button1_Click(ByVal sender As System.Object, _
      ByVal e As System.EventArgs) Handles Button1.Click
   Dim clsNew As New ShowMe()
   clsNew.ShowFrm()
End Sub
```

6. To use the example, run the application and click the command button several times. A minimized form icon appears on your taskbar as each new instance of the `ShowMe` class is created.

See Also

Creating and Using Objects

Resource Management

All objects consume system resources such as memory, file handles, and database connections. The Common Language Runtime (CLR) manages resources automatically, and you usually do not need to worry about releasing unneeded objects. However, understanding resource management works can help you design your applications to be more efficient.

Garbage Collection

The CLR uses a system called garbage collection to manage allocated resources. The system garbage collector releases an object's resources when that object can no longer be reached by any running code in your application. The following sections describe some of the changes to the way resources are managed in Visual Basic .NET.

Assigning Objects to Nothing

Nothing is a keyword used by Visual Basic .NET to indicate that an object variable does not contain a reference to an object. Previous versions of Microsoft Visual Basic encouraged you to assign unused objects to **Nothing** to disassociate the object variable from the object and release resources. You can still assign unused objects to **Nothing**, but because of the way Visual Basic .NET manages resources this process does not guarantee that objects will be released immediately. Generally speaking, you should only assign objects to **Nothing** for long-lived objects such as shared members or global variables.

Dispose

Some objects support a method named **Dispose** whose purpose is to release system resources more expeditiously. Classes that support the Dispose method should implement the **IDisposable** interface. The **Dispose** method needs to be explicitly called when you want to release object resources. For example:

```
ThisObject.Dispose
```

Finalize

Another method supported by some classes, **Finalize**, runs automatically when an object is released and can be used to perform other cleanup tasks. The **Finalize** method is similar to the **Class_Terminate()** method used in previous versions of Microsoft Visual Basic. Unlike the **Dispose** method, the CLR automatically calls the **Finalize** method sometime after an object is no longer needed.

See Also

Nothing | Object Lifetime: How Objects Are Created and Destroyed | Initialization and Termination of Components | IDisposable.Dispose Method | IDisposable Interface

Passing Objects to Procedures

Visual Basic .NET allows you to pass objects as arguments to procedures in the same way that you pass other types of arguments. The following procedures demonstrate how.

To pass a new instance of a form to a procedure

1. Open a project, create a new form and add a command button named `Button1` to it.
2. Copy the following code into the `Button1_click` event:

```
Protected Sub Button1_Click(ByVal sender As System.Object, _
       ByVal e As System.EventArgs)
   Dim newForm As New Form1()
   newForm.Show()
   CenterForm(newForm)
End Sub

Sub CenterForm(ByVal TheForm As Form)
   ' Centers the form on the screen.
   Dim RecForm As rectangle = Screen.GetBounds(TheForm)
   TheForm.Left = CInt((RecForm.Width - TheForm.Width) / 2)
   TheForm.Top = CInt((RecForm.Height - TheForm.Height) / 2)
End Sub
```

You can also pass an object as an argument by reference and then, inside the procedure, set the argument to a new object.

To pass an object reference to a procedure on another form

1. Open a project and create a form named `Form1`.
2. Add a second form named `Form2`.
3. Place a picture box control on each form.
4. Name the picture box on form1 `PictureBox1`.
5. Name the picture box on form2 `PictureBox2`.
6. Assign a picture to `PictureBox2` by clicking the **Image** property in the **Properties** window. Any small images will work; you can find .bmp and .jpg files in your Windows directory.
7. Add the following code to `Form2`:

```
Public Sub GetPicture(ByVal x As PictureBox)
   Dim objX As PictureBox
   ' Assign the passed-in picture box to an object variable.
   objX = x
   ' Assign the value of the Picture property to the Form1 picture box.
   objX.Image = PictureBox2.Image
End Sub
```

8. Add the following code to the **Form1_Click** event on `form1`:

```
Protected Sub Form1_Click(ByVal sender As System.Object, _
      ByVal e As System.EventArgs)
   Dim newForm2 As New Form2()
   newForm2.GetPicture(PictureBox1)
End Sub
```

9. Run the application and click `Form1`. The picture from `Form2` appears in the picture box on `Form1`.

   The **Form1_Click** event procedure calls the `GetPicture` procedure in `Form2`, and passes the empty picture box to it. The `GetPicture` procedure in `Form2` assigns the **Image** property of the picture box on `Form2` to the empty picture box on `Form1`, and the image from `Form2` is displayed on `Form1`.

See Also

Object-Oriented Programming in Visual Basic | Programming with Components | Resource Management

Managing Groups of Objects

For many applications, you want to be able to create and manage groups of related objects. There are two ways to group objects: by creating arrays of objects, and by creating collections.

Arrays are relatively inflexible structures. If you want to change the size of the array at run time, you must use the **ReDim** statement to redeclare it. All members of the array must be of the same type. On the other hand, arrays allow you to sequentially process all your objects, and you can have empty members of an array. For these reasons, arrays are most useful for creating and working with a fixed number of strongly typed objects.

Collections provide a more flexible way of working with groups of objects. A collection is a class, so you must declare a new collection before you can add members to that collection. Unlike arrays, the group of objects you work with can grow and shrink dynamically as the needs of the program change. You can assign a key to any object that you put into a collection so that you can retrieve and manipulate it based on that key.

In This Section

Creating Arrays of Objects
Demonstrates how to declare an array and instantiate its members.
Collections in Visual Basic .NET
Provides an overview of collections, discussing the **Collection** class, zero-based and one-based collections, index and key values, and how to add and remove items.
Creating Collections of Objects
Describes how to create a collection and use **For...Each...Next** to process its members.
The Visual Basic .NET Collection Class
Discusses the **Collection** class, what kinds of items you can add to a collection, and how to access the members after adding them.
Properties and Methods of the Collection Class
Describes the properties (**Count** and **Item**) and methods (**Add** and **Remove**) with which you can manipulate items in a **Collection** object.
Adding, Deleting, and Retrieving Items from a Collection
Explains how to add, retrieve, and delete items from a collection.

Related Sections

Creating and Using Objects
Shows how to create and use instances of classes.
Arrays
Discusses making your code more compact and powerful by declaring and using arrays.

Creating Arrays of Objects

You declare and use arrays of an object type just as you would declare and use an array of any data type. The members of this array can be retrieved by their index, and can be manipulated as any object of this type would be. Arrays also have built-in functionality for searching and sorting that can be accessed through the array variable. For more information on these methods, see Array Class.

To create an array of objects

1. Declare the array as shown in the sample code below. Because arrays are zero-based, they contain one member more than the upper bound you declare.

```
Dim x(10) As Widget   ' Contains 11 members, from x(0) to x(10).
```

2. Instantiate each member of the array, or assign each member a reference to an already existing object. An example of each approach is shown below:

```
' Instantiates each member of an array by using a loop.
Dim q As Integer
For q = 0 to 10
   x(q) = New Widget()
Next

' Assigns a member of an array a reference to an existing object.
Dim myWidget As New Widget()
x(0) = myWidget
x(1) = myWidget
```

   Note that you can assign the different members of the array references to the same object.

See Also

Arrays | Array Class | Creating Collections of Objects

Collections in Visual Basic .NET

In general terms, a collection is a way of grouping and managing related objects. This simple definition, however, describes a somewhat complex reality.

Every form, for example, has a **Controls** collection. This is an object that represents all the controls on that form. It allows you to obtain a reference to a control in the collection by its index, and to loop through the members of the collection using **For Each...Next** statements.

Visual Basic also provides a **Collection** class, with which you can define and create your own collections. Like a form's **Controls** collection, the **Collection** class also provides you with the built-in functionality to loop through members using **For Each...Next** and to reference members by index. Since both are collections, why then does the following code from a Windows Forms application generate a compiler error?

```
Dim myControlsCollection As Collection
myControlsCollection = Controls   ' This line generates a compiler error.
```

What's happening here? The **Controls** collection is a collection; the variable `myControlsCollection` is declared as **Collection**; why can't you assign a reference to **Controls** to the variable `myControlsCollection`?

The reason is that the **Collection** class and the **Controls** collection are not polymorphic; that is, they aren't interchangeable, because they are different types with different implementations. They don't have the same methods, store object references in the same way, or use the same kinds of index values.

These differences make the **Collection** class's name seem like an odd choice, because it really represents only one of many possible collection implementations. This topic explores some of these implementation differences.

Zero-Based and One-Based Collections

A collection is either zero-based or one-based, depending on what its starting index is. As you might guess, the former means that the index of the first item in the collection is zero, and the latter means that it is one. An example of a zero-based collection is the **Controls** collection, discussed above. An instance of the **Collection** object, also discussed above, is an example of a one-based collection.

One-based collections are more intuitive to use, because the index ranges from one to **Count**, where the **Count** property returns the number of items in a collection. The index of a zero-based collection, by contrast, ranges from zero to one less than the **Count** property.

The .NET Framework is standardizing collection as being zero-based. The Visual Basic Collection class is one based primarily for the purpose of compatibility with previous versions.

Index and Key Values

Many collections in Visual Basic allow you to access an item using either a numeric index or a **String** key, as do instances of the Visual Basic **Collection** class. You may also add items to **Collection** objects without specifying a key.

By contrast, some collections (such as **System.Collections.ArrayList**) allow only a numeric index. These collections provide access to their members only through the index, and do not allow you to associate a key.

Adding and Removing Items

Collections also differ in whether or not you can add items to them, and if so, how those items are added. Because the Visual Basic **Collection** object is a general-purpose programming tool, it's more flexible than other collections. It has an **Add** method for putting items into the collection, and a **Remove** method for taking items out.

Certain specialized collections, on the other hand, do not allow developers to add and remove members using code. For example, the **CheckedListBox.CheckedItems** property returns references to items by index, but you cannot add and remove items from the collection in code. Rather you can only do so by checking or unchecking the appropriate box in the visual interface. Thus there is no **Add** or **Remove** method for this collection.

See Also

Walkthrough: Creating Your Own Collection Class | Collections in Visual Basic .NET | Creating Collections of Objects | The Visual Basic .NET Collection Class | For Each...Next Statements | Count Property | Add Method | Remove Method |

Creating Collections of Objects

Collections are objects themselves. The first step in creating a collection of objects is to create the **Collection** object itself. Once the **Collection** object is created, items can be added, removed, and otherwise manipulated through it. In this example, you learn how to create and populate a **Collection** object.

To create a collection

1. Declare and instantiate a **Collection** variable as shown in the sample code below:

```
Dim myCollection As New Collection()
```

2. Use the **Add** method to add members to your collection. In this example, you create four strings and add them to your collection. A unique **String** value may optionally be added as the key for the members of your collection. This value is passed to the collection as the second argument of the **Add** method.

```
Dim w, x, y, z As String
w = "Wow!"
x = "It's"
y = "A"
z = "Collection"
myCollection.Add(w, "This")
myCollection.Add(x, "Is")
myCollection.Add(y, "A")
myCollection.Add(z, "Key")
```

3. If you want to remove any members of your collection, you can use the **Remove** method to do so based either on the index of the member, or on the optional key. Examples are shown below:

```
myCollection.Remove(1)   ' Removes the first member of the collection.
myCollection.Remove("Is")   ' Removes the member with the key "Is".
```

   Note that when a member is removed from a collection, the index values are renumbered from one to the **Count** value.

You can use the **For Each...Next** statement to loop through and process the members of your collection, as in the continuation of our example below.

To use For Each...Next to process the members of your collection

1. Declare a variable of the type you want to process in your collection. For example, if you want to loop through all your strings, declare a variable of type **String**, as shown below:

```
' Code from the previous example goes here.
Dim aString As String
```

2. Use the **For Each...Next** statement to examine each member of your collection. In this example, you search for a particular string, and if you find it, display it in a message box.

```
For Each aString in myCollection
   If aString = "Collection" Then
      MsgBox(aString)   ' Displays "Collection".
   End If
Next
```

See Also

Is Operator | The Visual Basic .NET Collection Class | Creating Arrays of Objects | Walkthrough: Creating Your Own Collection Class | Collection Object | Add Method | Remove Method | For Each...Next Statements

The Visual Basic .NET Collection Class

A collection is a way of grouping a set of related items. Many different types of collections exist. Collections are used in Visual Basic to keep track of many things, such as all the controls on a form (the **Controls** collection for example), and users can create their own collections to organize and manipulate objects.

Collections are a good way to keep track of objects that your application might need to dynamically create and destroy. The following code fragment shows how you might use the **Add** method of a collection object to keep a list of `Widget` objects the user has created.

```
' Declares and instantiates the Collection object.
Public widgetCollection As New Collection
' Creates a new Widget and adds it to the widgetCollection.
Private Sub MakeAWidget()
   Dim tempWidget As New Widget()
   widgetCollection.Add(tempWidget)
End Sub
```

The `widgetCollection` collection here organizes and exposes all of the `Widget` objects created through the `MakeAWidget` method. You can retrieve object references to each widget through the index of the collection. The size of the collection is adjusted automatically as each new `Widget` object is added. You can use the **For Each...Next** statements to iterate through the collection. If you want to give the Widget object a key by which it can be retrieved, you can supply a text string as the second parameter of the **Add** method, as described later in this section.

The **New** keyword in the declaration for the variable `widgetCollection` causes a **Collection** object to be created the first time the variable is referred to in code. Because **Collection** is a class, rather than a data type, you must create an instance of it and keep a reference to that instance (object) in a variable.

Like any other object, a **Collection** object is marked for garbage collection when the last variable that contains a reference to it is set to **Nothing** or goes out of scope; all the object references it contains are released when collected. For this reason, the variable `widgetCollection` is declared in the parent class, so that it exists throughout the life of the program.

Inside a Collection Object

A **Collection** object stores each item in an **Object**; thus the list of things you can add to a **Collection** object is the same as the list of things that can be stored in an object variable. This includes standard data types, objects, and arrays, as well as user-defined structures and class instances.

Because the **Collection** object stores each item as an object, **Object** is the type of reference the **Item** property returns. This presents the issue of how to access collection members later. If **Option Strict** is **Off**, you can implicitly convert a reference from the **Collection** to the appropriate type, as shown in this example.

```
Option Strict Off
Dim myString As String = "This is my String"
Dim aString As string
Dim myCollection As New Collection()
myCollection.Add(myString)
aString = myCollection.Item(1)   ' Collection item converted to string.
```

If **Option Strict** is **On,** however, you need to cast from **Object** to the type the collection contains. To obtain a reference from **Item** in this way, you can use the **CType** function to explicitly perform the conversion, as shown below:

```
Option Strict On
Dim myString As String = "This is my String"
Dim aString As String
Dim myCollection As New Collection()
myCollection.Add(myString)
aString = CType(myCollection.Item(1), String)
```

An alternative is to create your own strongly-typed collection. For a discussion on how to do this, see Walkthrough: Creating Your Own Collection Class.

See Also

Collections as an Alternative to Arrays | Collections in Visual Basic .NET | Collection Object |
Properties and Methods of the Collection Class | Add Method | For Each...Next Statements | New | Nothing | Item Property |
Option Strict Statement

Properties and Methods of the Collection Class

Each **Collection** object includes properties and methods you can use to insert, delete, and retrieve the items in the collection. The following table lists these items.

For information about	See
Adding items to the collection	Add Method
Returning the number of items in the collection	Count Property
Returning an item, by index or by key	Item Property
Deleting an item from the collection, by index or by key	Remove Method

These properties and methods provide only the most basic services for collections. For example, the **Add** method cannot check the type of object being added to a collection, which you might want to do to ensure that the collection contains only one kind of object. You can provide more robust functionality — and additional properties, methods, and events — by creating your own collection class, as described in the walkthrough, Walkthrough: Creating Your Own Collection Class.

The basic services of adding, deleting, and retrieving from a collection depend on keys and indexes. A *key* is a **String** value. It could be a name, a driver's license number, a social security number, or simply an integer converted to a string. The **Add** method allows you to associate a key with an item, as described in Adding, Deleting, and Retrieving Items from a Collection.

An *index* in the **Collection** class is an integer between one (1) and the number of items in the collection. You can control the initial value of an item's index, using the *before* and *after* named parameters, but its value may change as other items are added and deleted. For more information, see the **Add** method.

You can use the index to iterate over the items in a collection. For example, the following code shows two ways to give all the employees in a collection of Employee objects a 10 percent raise, assuming that the variable `employeesCollection` contains a reference to a **Collection** object:

```
Option Strict On
Dim counter As Integer
Dim emp As Employee
For counter = 1 To employeesCollection.Count
   emp = CType(employeesCollection(counter), Employee)
   emp.Rate *= 1.1
Next

 ' Alternative code follows.
Dim emp As Employee
For Each emp In employeesCollection
   emp.Rate *= 1.1
Next
```

> **Note** If you attempt to use **For Each** with a collection that contains items of a different type than the one being iterated, an **ArgumentException** is thrown.

See Also

Adding, Deleting, and Retrieving Items from a Collection | The Visual Basic .NET Collection Class | Add Method | Count Property | Item Property | Remove Method | Walkthrough: Creating Your Own Collection Class

Adding, Deleting, and Retrieving Items from a Collection

The **Collection** class contains built-in functionality to allow you to add, delete and retrieve items. The use of this functionality is demonstrated below.

Adding Items to a Collection

Use the **Add** method to add an item to a collection. The syntax is:

```
object.Add(Item, Key)
```

For example, to add a work order object to a collection of work orders using the work order's ID property as the key, you can write:

```
workordersCollection.Add(woNew, woNew.ID)
```

This assumes that the ID property is a string. If the property is a number (for example, a **Long** integer), use its **.ToString** function to convert it to the **String** value required for keys:

```
workordersCollection.Add(woNew, woNew.ID.ToString)
```

The use of a key is optional. If you do not want to associate a key with the object in your collection, you may add it without a key, as shown below:

```
workordersCollection.Add(woNew)
```

You can use the *before* and *after* arguments to maintain an ordered collection of objects. The member to be added is placed in the collection before or after the member identified by the *before* or *after* argument, respectively. For example, setting *before* equal to 1 inserts an item at the beginning of the collection, because **Collection** objects are one-based, as in the example that follows:

```
workordersCollection.Add(woNew, woNew.ID, 1)
```

Similarly, the *after* argument adds an item after the specified index. The following example adds an item as the third element:

```
workordersCollection.Add(woNew, woNew.ID,,2)
```

You can specify a value for either *before* or *after*, but not both.

Deleting Items from a Collection

Use the **Remove** method to delete an item from a collection. The syntax is:

```
object.Remove(index | key)
```

The *index* argument can either be the position of the item you want to delete, or the item's key. If the key of the third element in a collection is "W017493", you can use either of these two statements to delete it:

```
workordersCollection.Remove(3)
workordersCollection.Remove("W017493")
```

Retrieving Items from a Collection

Use the **Item** property to retrieve specific items from a collection. The syntax is:

```
variable = object.Item(index)
```

As with the **Remove** method, the index can be either the item's position in the collection or the item's key. Using the same

example as for the **Remove** method, either of these statements retrieves the third element in the collection:

```
woCurrent = workordersCollection.Item(3)
woCurrent = workordersCollection.Item("W017493")
```

> **Note** If you use whole numbers as keys, you must use the **.ToString** function to convert them to strings before passing them to the **Add**, **Remove**, or **Item** property. A **Collection** object always assumes that a whole number is an index.

Item Is the Default Property

The **Item** property is the default property for a **Collection** object, so you can omit it when you access an item in a collection. Thus, the previous code examples could also be written as follows:

```
woCurrent = workordersCollection(3)
woCurrent = workordersCollection("W017493")
```

You can also use the **!** member access qualifier to access a member of the collection by its key without qualifying the key with quotation marks or parentheses. Thus the previous example can also be written as follows:

```
woCurrent = workordersCollection!W017493
```

> **Note** Collection objects update their numeric index numbers automatically as you add and delete elements; the numeric index of a given element thus often changes. Therefore, do not save a numeric index value and expect it to retrieve the same element later in your program. Use keys for this purpose.

See Also

Properties and Methods of the Collection Class | The Visual Basic .NET Collection Class | Collections as an Alternative to Arrays | Collections in Visual Basic .NET | Add Method | Remove Method | Item Property

Obtaining Class Information at Run Time

In some situations, you may need to work with variables defined as type **Object** rather than as more specific object types. Variables defined as **Object** can contain any other type object, such as integers, doubles, or object data types.

In This Section

Determining Object Type
Discusses determining the class an object belongs to.

Calling a Property or Method Using a String Name
Steps you through the process of specifying arguments by name instead of by position when calling a method.

Related Sections

Objects
Covers the fundamentals of object-oriented programming.

Determining Object Type

Generic object variables (that is, variables you declare as **Object**) can hold objects from any class. When using variables of type **Object**, you may need to take different actions based on the class of the object; for example, some objects might not support a particular property or method. Visual Basic .NET provides two means of determining which type of object is stored in an object variable: the **TypeName** function and the **TypeOf...Is** operator.

The **TypeName** function returns a string and is the best choice when you need to store or display the class name of an object, as shown in the following code fragment:

```
    MsgBox(TypeName(Ctrl))
```

The **TypeOfIs** operator is the best choice for testing an object's type, because it is much faster than an equivalent string comparison using **TypeName**. The following code fragment uses **TypeOfIs** within an **If...Then...Else** statement:

```
If TypeOf Ctrl Is Button Then
   MsgBox("The control is a button.")
End If
```

A word of caution is due here. The **TypeOfIs** operator returns **True** if an object is of a specific type, or is derived from a specific type. Almost everything you do with Visual Basic .NET involves objects, which include some elements not normally thought of as objects, such as **Strings** and **Integers**. These objects are derived from and inherit methods from **System.Object**. When passed an **Integer** and evaluated with **Object**, the **TypeOfIs** operator returns **True**. The following example reports that the parameter `InParam` is both an **Object** and an **Integer**:

```
Sub CheckType(ByVal InParam)
' Both If statements evaluate to True when an
' Integer is passed to this procedure.
   If TypeOf InParam Is Object Then
      MsgBox("InParam is an Object")
   End If
   If TypeOf InParam Is Integer Then
      MsgBox("InParam is an Integer")
   End If
End Sub
```

The following example uses both **TypeOfIs** and **TypeName** to determine the type of object passed to it in the `Ctrl` argument. The `TestObject` procedure calls `ShowType` with three different kinds of controls.

To run the example

1. Create a new Windows Forms project and add a **Button** control, a **CheckBox** control, and a **RadioButton** control to the form.
2. From the button on your form, call the `TestObject` procedure.
3. Add the following code to the **Declarations** section of your form:

```
Sub ShowType(ByVal Ctrl As Object)
   'Use the TypeName function to display the class name as text.
   MsgBox(TypeName(Ctrl))
   'Use the TypeOf function to determine the object's type.
   If TypeOf Ctrl Is Button Then
      MsgBox("The control is a button.")
   ElseIf TypeOf Ctrl Is CheckBox Then
      MsgBox("The control is a check box.")
   Else
      MsgBox("The object is some other type of control.")
   End If
End Sub

Protected Sub TestObject()
```

```
'Test the ShowType procedure with three kinds of objects.
   ShowType(Me.Button1)
   ShowType(Me.CheckBox1)
   ShowType(Me.RadioButton1)
End Sub
```

See Also

Calling a Property or Method Using a String Name | Performing Multiple Actions on an Object | Obtaining Class Information at Run Time | Object Data Type | TypeOf | TypeName Function | If...Then...Else Statements | String Data Type | Integer Data Type

Calling a Property or Method Using a String Name

In most cases, you can discover the properties and methods of an object at design time, and write code to handle them. In a few cases, however, you may not know about an object's properties and methods in advance, or you may simply want the flexibility of allowing an end user to specify properties or execute methods at run time.

Consider, for example, a client application that evaluates expressions entered by the user by passing an operator to a COM component. Suppose you are constantly adding new functions to the component that require new operators. Using standard object access techniques, you would need to recompile and redistribute the client application before it could use the new operators. To avoid this, you can use the **CallByName** function to pass the new operators as strings, without changing the application.

The **CallByName** function allows you to use a string to specify a property or method at run time. The signature for the **CallByName** function looks like this:

```
Result = CallByName(Object, ProcedureName, CallType, Arguments())
```

The first argument, *Object*, takes the name of the object you want to act upon. The *ProcedureName* argument takes a string containing the name of the method or property procedure to be invoked. The *CallType* argument takes a constant representing the type of procedure to invoke: a method (`Microsoft.VisualBasic.CallType.Method`), a property read (`Microsoft.VisualBasic.CallType.Get`), or a property set (`Microsoft.VisualBasic.CallType.Set`). The *Arguments* argument, which is optional, takes an array of type **Object** containing any arguments to the procedure.

You can use **CallByName** with classes in your current solution, but it is most often used to access COM objects or objects from .NET assemblies.

Suppose you add a reference to an assembly that contains a class named `MathClass`, which has a new function named `SquareRoot`, as shown in the following code:

```
Class MathClass
   Function SquareRoot(ByVal X As Double) As Double
      Return Math.Sqrt(X)
   End Function
   Function InverseSine(ByVal X As Double) As Double
      Return Math.Atan(X / Math.Sqrt(-X * X + 1))
   End Function
   Function Acos(ByVal X As Double) As Double
      Return Math.Atan(-X / Math.Sqrt(-X * X + 1)) + 2 * Math.Atan(1)
   End Function
End Class
```

Your application could use text box controls to control the method will be called and its arguments. For example, if `TextBox1` contains the expression to be evaluated, and `TextBox2` is used to enter the name of the function, you can use the following code to invoke the `SquareRoot` function on the expression in `TextBox1`:

```
Private Sub CallMath()
   Dim Math As New MathClass()
   Me.TextBox1.Text = CStr(CallByName(Math, Me.TextBox2.Text, _
      Microsoft.VisualBasic.CallType.Method, TextBox1.Text))
End Sub
```

If you enter "64" in `TextBox1`, "SquareRoot" in `TextBox2`, and then call the `CallMath` procedure, the square root of the number in `TextBox1` is evaluated. The code in the example invokes the `SquareRoot` function (which takes a string containing the expression to be evaluated as a required argument) and returns "8" in `TextBox1` (the square root of 64). Of course, if the user enters an invalid string in `TextBox2`, if the string contains the name of a property instead of a method, or if the method had an additional required argument, a run-time error occurs. You need to add robust error-handling code when you use **CallByName** to anticipate these or any other errors.

> **Note** While the **CallByName** function may be useful in some situations, you need to weigh its usefulness against the performance implications — using **CallByName** to invoke a procedure is slightly slower than a late-bound call. If you are invoking a function that is called repeatedly, such as inside a loop, **CallByName** can have a severe effect on performance.

See Also

Determining Object Type | Performing Multiple Actions on an Object | Obtaining Class Information at Run Time | CallByName Function

Understanding Classes

Classes are important in object-oriented programming because they allow you to group related items as a unit as well as control their visibility and accessibility to other procedures. Classes also can inherit and reuse code defined in other classes.

In This Section

Classes: Blueprints for Objects
Provides an overview of encapsulation, inheritance, and shared members.
Walkthrough: Defining Classes
Steps through the process of creating a class.
Classes vs. Standard Modules
Explains the differences between classes and standard modules.
Object Lifetime: How Objects Are Created and Destroyed
Discusses creating and disposing of class instances.
Using Constructors and Destructors
Discusses how to initialize new instances of classes and dispose of resources when they are no longer needed.
Class Properties, Fields, and Methods
Explains the fields, properties, and methods that make up a class.

Related Sections

Creating and Using Objects
Demonstrates how to create and use instances of classes.
Interfaces in Visual Basic .NET
Describes what interfaces are and how to use them in your applications.
Inheritance
Describes how to define classes that serve as the basis for other classes.

Classes: Blueprints for Objects

Classes are symbolic representations of objects; they describe the properties, fields, methods, and events that make up objects in the same way that blueprints describe the items that make up a building. Just as a blueprint can be used to create multiple buildings, a single class can be used to create as many objects as necessary. Just as a blueprint defines which parts of a building are accessible to people who use the building, so too can classes control user access to object items through encapsulation.

Classes and Objects

The terms "class" and "object" are sometimes used interchangeably, but in fact, classes describe the *structure* of objects, while objects are usable *instances* of classes. Each instance is an exact yet distinct copy of its class. Because an object is an "instance" of a class, the act of creating an object is called *instantiation*.

Using the blueprint analogy, a class is a blueprint, and an object is a building based on that blueprint. For the most part, changing the data in one object does not change the data in any other object. (The exception is *shared members,* class members declared with the **Shared** modifier, which exist independently of specific instances of a class.)

Encapsulation

Encapsulation is the ability to contain and control the access to a group of associated items. Classes provide one of the most common ways to encapsulate items. In the example below, the `BankAccount` class encapsulates the methods, fields, and properties that describe a bank account. Without encapsulation, you would need to declare separate procedures and variables to store and manage information for the bank account, and it would be difficult to work with more than one bank account at a time. Encapsulation lets you use the data and procedures in the `BankAccount` class as a unit. You can work with multiple bank accounts at the same time without confusion because each account is represented by a unique instance of the class.

Encapsulation also allows you to control how the data and procedures are used. You can use access modifiers, such as **Private** or **Protected**, to prevent outside procedures from executing class methods or reading and modifying data in properties and fields. You should declare internal details of a class as **Private** to prevent them from being used outside your class; this technique is called *data hiding*. In the `BankAccount` class, customer information such as account balance is protected this way. One of the basic rules of encapsulation is that class data should be modified or retrieved only via **Property** procedures or methods. Hiding the implementation details of your classes prevents them from being used in undesired ways, and lets you modify such items later without risk of compatibility problems. For example, later versions of the `BankAccount` class listed below could change the data type of the `AccountBalance` field without danger of breaking applications that rely on this field having a specific data type.

Inheritance

Like Visual Basic .NET structures, classes let you define data types that encapsulate a group of related items. Unlike structures, however, Visual Basic .NET classes can inherit and extend the characteristics of other classes. Classes that serve as a basis for new classes are called *base classes*. Classes derived from base classes are called *derived classes*. Derived classes inherit all the fields, properties, methods, and events defined in the base class. This means you can develop and debug a class once, and then reuse it as the basis for other classes.

The following example defines a base class that represents a generic bank account, and a specific class that inherits the properties of the base class but is customized to describe a checking account:

```
Class BankAccount
   Private AccountNumber As String
   Private AccountBalance As Decimal
   Private HoldOnAccount As Boolean = False
   Public Sub PostInterest()
      ' Add code to calculate the interest for this account.
   End Sub
   ReadOnly Property Balance() As Decimal
      Get
         Return AccountBalance 'Returns the available balance.
      End Get
   End Property
End Class

Class CheckingAccount
   Inherits BankAccount
   Sub ProcessCheck()
      ' Add code to process a check drawn on this account.
```

```
    End Sub
End Class
```

For more information about inheritance, see Inheritance Basics.

Shared Members

By default, class data is specific to each instance of the class, but there may be occasions when you want a single data item to be shared among all objects created from a class. In such cases, you can use the **Shared** modifier to cause a variable to share the same value in all instances of a class. (Shared members are sometimes referred to as "static members" in other programming languages.) You can call shared methods directly using a class name without first creating an instance of the class.

For more information about shared members, see Shared Members.

Shadowing

Derived classes can use the **Shadows** keyword to declare a member with the same name as an inherited member. Shadowed members do not need to be the same data type as the member being shadowed. For example, a property can shadow a variable of type **Integer**.

For more information about shared members, see Shadowing.

See Also

Creating and Using Objects | Structures and Classes | Inheritance | Inheritance Basics | Shared Members | Shadowing | Shadows | Shared

Walkthrough: Defining Classes

This walkthrough demonstrates how to use class modules to define classes, from which you can then create objects. It also shows you how to create properties and methods for the new class, and demonstrates how objects are initialized.

To define a class

1. Open a new Windows Application project by clicking **New** on the **File** menu, and then clicking **Project**. The **New Project** dialog box appears.
2. Select Windows Application from the list of Visual Basic project templates. The new project is displayed.
3. Add a new class to the project by clicking **Add Class** on the **Project** menu. The **Add New Item** dialog box appears.
4. Name the new module `StateInfo.vb`, and then click **Open**. The code for the new class is displayed:

```
Public Class StateInfo
End Class
```

   Note You can use the Visual Studio .NET code editor to add a class to your startup form by typing the **Class** keyword followed by the name of the new class. The code editor provides a corresponding **End Class** statement for you.

5. To simplify access to the registry classes, add an **Imports** statement to the top of the source code containing the class statement:

```
Imports Microsoft.Win32
```

6. Define three private fields for the class by placing the following code between the **Class** and **End Class** statements:

```
   Private pVal As String ' Holds the LastFile property value.
   Private KeyName As String ' Holds the name of the registry key.
   Private SubKeyName As String ' Holds the name of the subkey.
```

   These three fields are declared as **Private**, and can be used only from within the class. You can make fields accessible from outside a class by using access modifiers that provide greater access, such as **Public**.

7. Define a property for the class by adding the following code:

```
Public Property LastFile() As String
   Get ' Retrieves the property value.
      Return pVal
   End Get
   Set(ByVal Value As String)
      pVal = Value
   End Set
End Property
```

8. Define methods for the class by adding the following code:

```
Sub SaveStateInfo()
'  Save the current value of the LastFile property to the registry.
   Dim aKey As RegistryKey
'  Create a key if it does not exist.
   aKey = Registry.CurrentUser.CreateSubKey(KeyName)
'  Save the property value to the registry.
   aKey.SetValue(SubKeyName, pVal)
End Sub

Sub GetStateInfo()
'  Restore the property value LastFile from the
'  value stored in the registry.
```

```
   Dim aKey As Object
   Dim myRegKey As RegistryKey = Registry.CurrentUser
   Try
'  This call goes to the Catch block if the registry key is not set.
   myRegKey = myRegKey.OpenSubKey(KeyName)
      Dim oValue As Object = myRegKey.GetValue(SubKeyName)
         pVal = CStr(oValue)
   Catch
      pVal = "" ' Set to default if key value cannot be read.
   End Try
End Sub
```

9. Define a parameterized constructor for the new class by adding a procedure named **Sub New**:

```
Sub New(ByVal RegistryKeyName As String, _
        ByVal RegistrySubKeyName As String)
'  Save the names of the registry key and subkey in two fields.
   KeyName = RegistryKeyName
   SubKeyName = RegistrySubKeyName
'  Restore the value of the LastFile property from the registry.
   MyClass.GetStateInfo()
End Sub
```

The **Sub New** constructor is called automatically when an object based on this class is created. This constructor sets the value of the fields that hold the registry key and subkey names, and calls a procedure that restores the value of the `LastFile` property from the registry.

> **Note** This walkthrough uses the registry to store state information for the class. Information stored in the registry is available to other applications and users, and therefore it should not be used to store security, or critical application information.

10. Define a **Finalize** destructor for the class by adding a procedure called `Finalize`:

```
Protected Overrides Sub Finalize()
'  Save the value of the LastFile property to the registry
'  when the class loses scope.
   SaveStateInfo()
   MyBase.Finalize() ' Call Finalize on the base class.
End Sub
```

The **Finalize** destructor saves the value stored in the property to the registry after the class loses scope.

To create a button to test the class

1. Change the startup form to design mode by right-clicking its name in the Solution Explorer, and then clicking **View Designer**. The startup form for Windows Application projects is named Form1.vb by default. The main form appears.
2. Add a button to the main form and double-click it to display the code for the **Button1_Click** event handler. Add the following code to call the test procedure:

```
'  Create an instance of the class and pass the registry key
'  names to the Sub New constructor.
   Dim SI As New StateInfo("Software\StateInfo", "PropertyValue")
'  Display the value of the property if it has been restored.
   If Len(SI.LastFile) > 1 Then
      MsgBox("The value of the property LastFile is: " _
             & SI.LastFile)
   Else
      MsgBox("The LastFile property has not been set.")
   End If
   SI.LastFile = "C:\BinaryFile.txt" ' Set the property.
```

```
'  Ask the object to save the property value to the registry.
   SI.SaveStateInfo()
```

To run your application

1. Run your application by pressing F5. The application appears.
2. Click the button on the form to call the test procedure. The first time you do so, a message is displayed stating that the `LastFile` property has not been set.
3. Click **OK** to dismiss the message box. The **Button1_Click** procedure sets the value of the `LastFile` property and calls the **SaveState** method. Even if **SaveState** were not called explicitly from **Button1_Click**, it would be called automatically by the **Finalize** method after the startup form is closed.
4. Click the button a second time. The message "The value of the property LastFile is C:\BinaryFile.txt" is displayed.
5. Close the main form and then restart the program by pressing F5. An object based on the class is created, and its **Sub New** constructor calls the **GetStateInfo** procedure that restores the property value from the registry. The message "The value of the property LastFile is C:\BinaryFile.txt" is displayed again when you click the button.

See Also

Classes vs. Standard Modules | Class Properties, Fields, and Methods | Adding Events to a Class | Imports Statement

Classes vs. Standard Modules

Both classes and modules are reference types that encapsulate items defined within, but they differ in how these items are accessed from other procedures.

The primary difference between classes and modules is that classes can be instantiated and standard modules cannot. Because there is never more than one copy of a standard module's data, when one part of your program changes a public variable in a standard module, any other part of the program gets the same value if it subsequently reads that variable. Class data, on the other hand, exists separately for each instantiated object. Another difference is that unlike standard modules, classes can implement interfaces.

Classes and modules also employ different scope for their members. Members defined within a class are scoped within a specific instance of the class, and exist only for the lifetime of the object. The practical result is that, to access class members from outside a class, you must use only fully qualified names; for example, `Object.Member`. Members declared within a standard module, on the other hand, are shared by default, and are scoped to the declaration space of the standard module's containing namespace. This means that public variables in a standard module are effectively global variables because they are visible from anywhere in your project, and they exist for the life of the program. Unlike class members, members of standard modules are implicitly shared and cannot use the Shared keyword.

See Also

Understanding Classes | Structures and Classes | Shared Members | Ways to Implement Component Functionality | Shared

Object Lifetime: How Objects Are Created and Destroyed

Objects begin life when an instance of a class is created using the **New** keyword. New objects often require initialization tasks to be performed before they are used for the first time. Common initialization tasks include opening files, connecting to a database, and reading the values of registry keys. Microsoft Visual Basic .NET controls the initialization of new objects using procedures called constructors.

Objects end life after they leave scope and are released by the common language runtime (CLR). Visual Basic .NET controls the release of system resources using procedures called *destructors*. Together, *constructors* (special methods that allow control over initialization) and destructors support the creation of robust and predictable class libraries.

Sub New and Sub Finalize

The **Sub New** and **Sub Finalize** procedures in Visual Basic .NET initialize and destroy objects; they replace the **Class_Initialize** and **Class_Terminate** methods used in previous versions of Visual Basic. Unlike **Class_Initialize**, the **Sub New** constructor can run only once when a class is created, and it cannot be called explicitly anywhere other than in the first line of code in another constructor from either the same class or from a derived class. Furthermore, the code in the **Sub New** method always runs before any other code in a class. Visual Basic .NET implicitly creates a **Sub New** constructor at run time if you do not explicitly define a **Sub New** procedure for a class.

Before releasing objects, the CLR automatically calls the **Finalize** method for objects that define a **Sub Finalize** procedure. The **Finalize** method can contain code that needs to execute just before an object is destroyed, such as closing files and saving state information. There is a slight performance penalty for executing **Sub Finalize**, so you should define a **Sub Finalize** method only when you need to release objects explicitly.

The **Finalize** destructor is a protected method that can be called only from the class it belongs to, or from derived classes. The system calls **Finalize** automatically when an object is destroyed, so you should not explicitly call **Finalize** from outside of a derived class's **Finalize** implementation. Unlike **Class_Terminate**, which executed as soon as an object was set to nothing, there is usually a delay between when an object loses scope and when Visual Basic .NET calls the **Finalize** destructor. Visual Basic .NET allows for a second kind of destructor, named **Dispose**, which can be explicitly called at any time to release resources immediately.

IDisposable Interface

Class instances often control resources not managed by the CLR, such as Windows handles and database connections. To supplement garbage collection, your classes can provide a mechanism to actively manage system resources if they implement the **IDisposable** interface. **IDisposable** has one method, **Dispose**, which clients should call when they finish using an object. You can use the implementation of **Dispose** to release resources and perform tasks such as closing files and database connections. Unlike the **Finalize** destructor, the **Dispose** method is not called automatically. Clients of a class must explicitly call **Dispose** when you want to release resources.

Garbage Collection and the Finalize Destructor

The .NET Framework uses a system called *reference-tracing garbage collection* that periodically releases unused resources. Previous versions of Visual Basic use a different system called *reference counting* to manage resources. Although both systems perform the same function automatically, there are a few important differences.

The CLR periodically destroys objects when the system determines that such objects are no longer needed. Objects are released more quickly when system resources are in short supply, and less frequently otherwise. The delay between when an object loses scope and when the CLR releases it means that, unlike with objects in previous versions of Visual Basic, you cannot determine exactly when the object will be destroyed. In such a situation, objects are said to have *non-deterministic lifetime*. In most cases, non-deterministic lifetime does not change how you write applications, as long as you remember that the **Finalize** destructor may not execute immediately when an object loses scope.

Another difference between the garbage collection systems involves the use of **Nothing**. To take advantage of reference counting, programmers using previous versions of Visual Basic sometimes assign **Nothing** to object variables in order to release the references those variables hold. If the variable holds the last reference to the object, the object's resources are released immediately. In Visual Basic .NET, while there may be cases in which this procedure is still valuable, performing it never causes the referenced object to release its resources immediately. The only time you should set a variable to **Nothing** is when its lifetime is long relative to how long it takes the garbage collector to detect orphaned objects.

See Also

Using Constructors and Destructors | Initialization and Termination of Components | Language Changes in Visual Basic | New |

Finalize Methods and Destructors | IDisposable.Dispose Method | Nothing

Using Constructors and Destructors

Constructors and destructors control the creation and destruction of objects.

To create a constructor for a class, create a procedure named **Sub New** anywhere in the class definition. To create a parameterized constructor, specify the names and data types of arguments to **Sub New** just as you would specify arguments for any other procedure, as in the following code:

```
Sub New(ByVal sString As String)
```

Constructors are frequently overloaded, as in the following code:

`Sub New(ByVal sString As String, iInt As Integer)`

When you define a class derived from another class, the first line of a constructor must be a call to the constructor of the base class, unless the base class has an accessible constructor that takes no parameters. A call to the base class containing the above constructor, for example, would be `MyBase.New(sString)`. Otherwise, `MyBase.New` is optional, and the Visual Basic .NET runtime calls it implicitly.

After writing code to call the parent object's constructor, you can add any additional initialization code to the **Sub New** procedure. **Sub New** can accept arguments when called as a parameterized constructor. These parameters are passed from the procedure calling the constructor; for example, `Dim AnObject As New ThisClass(X)`.

The following code example shows how to use **Dispose** and **Finalize** to release resources:

```
' Design pattern for a base class.
Public Class Base
    Implements IDisposable
    ' Implement IDisposable.
    Public Overloads Sub Dispose() Implements IDisposable.Dispose
        Dispose(True)
        GC.SuppressFinalize(Me)
    End Sub

    Protected Overridable Overloads Sub Dispose(ByVal disposing As Boolean)
        If disposing Then
            ' Free other state (managed objects).
        End If
        ' Free your own state (unmanaged objects).
        ' Set large fields to null.
    End Sub

    Protected Overrides Sub Finalize()
        ' Simply call Dispose(False).
        Dispose(False)
    End Sub
End Class

' Design pattern for a derived class.
Public Class Derived
    Inherits Base

    Protected Overloads Overrides Sub Dispose(ByVal disposing As Boolean)
        If disposing Then
            ' Release managed resources.
        End If
        ' Release unmanaged resources.
        ' Set large fields to null.
        ' Call Dispose on your base class.
        MyBase.Dispose(disposing)
    End Sub
    ' The derived class does not have a Finalize method
    ' or a Dispose method with parameters because it inherits
    ' them from the base class.
End Class
```

The following example shows a common design pattern for using the **Dispose** destructor:

```
Dim con As Connection, rs As RecordSet
Try
   con = New Connection("xyz")
   rs = New RecordSet(con, "MyTable")
   ' Use the connection if successful.
Finally
' Call the Dispose method when done with any created objects.
If Not con Is Nothing Then
      con.Dispose()
   End If
   If Not rs Is Nothing Then
      rs.Dispose()
   End If
End Try
```

The next example creates an object using a parameterized constructor, and calls destructors when the object is no longer needed:

```
Sub TestConstructorsAndDestructors()
   Dim X As Integer = 6
   Dim AnObject As New ThisClass(X)
   ' Place statements here that use the object.
   AnObject.DoSomething()
   ' Test the parameterized constructor.
   MsgBox("The value of ThisProperty after being initialized " & _
          " by the constructor is " & AnObject.ThisProperty)
   ' Run the Dispose method when you are done with the object.
   AnObject.Dispose()
End Sub

   Public Class BaseClass
      Sub New()
         MsgBox("The base class is initializing with Sub New.")
      End Sub

      Protected Overrides Sub Finalize()
         MsgBox("The base class is destroying objects with Sub Finalize.")
         ' Place final cleanup tasks here.
         MyBase.Finalize()
      End Sub
   End Class

   Public Class ThisClass
      Inherits BaseClass
      Implements IDisposable ' Implements the Dispose method of IDisposable.
      Private m_PropertyValue  As Integer

      Sub New(ByVal SomeValue As Integer)
         MyBase.New() ' Call MyBase.New if this is a derived class.
         MsgBox("Sub New is initializing the class ThisClass.")
         ' Place initialization statements here.
         m_PropertyValue  = SomeValue
      End Sub

      Property ThisProperty() As Integer
         Get
            ThisProperty = m_PropertyValue
         End Get
         Set(ByVal Value As Integer)
            m_PropertyValue  = Value
         End Set
      End Property

      Sub DoSomething()
         ' Place code here that does some task.
      End Sub
```

```
        Protected Overrides Sub Finalize()
           MsgBox("Finalize is destroying objects in ThisClass.")
           ' Place final cleanup tasks here.
           MyBase.Finalize()
        End Sub

        Overridable Sub Dispose() Implements IDisposable.Dispose
           MsgBox("The ThisClass is running Dispose.")
        End Sub
     End Class
```

When you run this example, the class `ThisClass` calls the **Sub New** constructor of the class `BaseClass`. After the constructor in the base class is finished, the class `ThisClass` runs the remaining statements in **Sub New** that initialize a value for the property `ThisProperty`.

When the class is no longer needed, the **Dispose** destructor is called in `ThisClass`.

If you originally created the instance of `ThisClass` from a form, nothing will seem to happen until you close the form. At that point, the **Finalize** destructor runs in the class `ThisClass` and finally in the class `BaseClass`.

See Also

Object Lifetime: How Objects Are Created and Destroyed | Finalize Methods and Destructors | IDisposable.Dispose Method

Class Properties, Fields, and Methods

In general, fields and properties represent information about an object, whereas methods represent actions an object can take.

The following topics describe the mechanisms for adding properties, fields, and methods to your classes, and address issues associated with these items.

In This Section

Adding Fields and Properties to a Class
Provides information on declaring fields and properties.
Properties and Property Procedures
Explains how **Property** procedures work, and how to implement common property types.
Property Procedures vs. Fields
Helps you decide when to use fields to store data in a class, and when properties are a better choice.
Class Methods
Discusses the public procedures you add to your class.
Properties vs. Methods
Helps you decide whether your desired functionality is best implemented as a property or a method.
Default Properties
Discusses properties that are used when no specific property is named.
Overloaded Properties and Methods
Covers how to define multiple properties or methods that have the same name, but work with different data types.
Overriding Properties and Methods
Discusses redefining an inherited property or method.
Object Management with Collections
Provides information on using collections to store and retrieve objects.

Related Sections

Creating and Using Objects
Demonstrates how to create and use instances of classes.
Events and Delegates
Demonstrates how to declare and use events and delegates.

Adding Fields and Properties to a Class

You can use both fields and properties to store information in an object. Although fields and properties are almost indistinguishable from a client-application perspective, they are declared differently within a class. Whereas fields are simply public variables that a class exposes, properties use **Property** procedures to control how values are set or returned.

To add a field to a class

- Declare a public variable in the class definition, as in the following code:

```
Class ThisClass
   Public ThisField As String
End Class
```

To add a property to a class

1. Declare a local variable within the class to store the property value. This step is necessary because properties do not allocate any storage on their own. To protect their values from direct modification, variables used to store property values should be declared as **Private**.
2. Preface property declarations with modifiers, such as **Public** and **Shared**, as appropriate. Use the **Property** keyword to declare the property name, and declare the data type that the property stores and returns.
3. Define **Get** and **Set** property procedures within the property definition. **Get** property procedures are used to return the value of a property and are roughly equivalent to functions in syntax. They accept no arguments and can be used to return the value of private local variables declared within the class used to store the property value. **Set** property procedures are used to set the value of a property; they have a parameter, usually called `Value`, with the same data type as the property itself. Whenever the value of the property is changed, `Value` is passed to the **Set** property procedure, where it can be validated and stored in a local variable.
4. Terminate **Get** and **Set** property procedures with **End Get** and **End Set** statements as appropriate.
5. Terminate the property block with an **End Property** statement.

> **Note** If you are working within the Visual Studio integrated development environment (IDE), you can direct it to stub out empty **Get** and **Set** property procedures. Type **Property** *PropName* **As** *DataType* (where *PropName* is the name of your property, and *DataType* is a specific data type, such as **Integer**), and the property procedures appear in the Code editor.

The following example declares a property in a class:

```
Class ThisClass
   Private m_PropVal As String
   Public Property One() As String
      Get
         Return m_PropVal ' Return the value stored in the local variable.
       ' Optionally, you can use the syntax One = PropVal to return
       ' the property value.
      End Get
      Set(ByVal Value As String)
         m_PropVal = Value ' Store the value in a local variable.
      End Set
   End Property
End Class
```

When you create an instance of `ThisClass` and set the value of the property `One`, the **Set** property procedure is called and the value is passed in the `Value` parameter, which is stored in a local variable named `m_PropVal`. When the value of this property is retrieved, the **Get** property procedure is called like a function and returns the value stored in the local variable `m_PropVal`.

See Also

Adding Events to a Class | Properties and Property Procedures | Property Statement | Public | Shared

Properties and Property Procedures

You can use both properties and fields to store information in an object. Whereas fields are simply public variables, properties use property procedures to control how values are set or returned. Property procedures are blocks of code declared within property definitions that allow you to execute code when a property value is set or retrieved. Visual Basic .NET has two types of property procedures: **Get** property procedures for retrieving a property value, and **Set** property procedures for assigning a value to a property. For example, a property that stores the balance of a bank account might use code in a **Get** property procedure to post interest and check for service fees before returning the available balance. The **Set** property procedure for the available balance could provide validation code to prevent the balance from being updated incorrectly. In short, property procedures allow an object to protect and validate its own data.

The following code fragment compares how fields and properties validate values:

```
Protected Sub TestFieldsAndProperties()
   ' Assume, for this example, that the only valid values for
   ' the field and property are numbers less than 10.
   Dim NewClass As New ThisClass()

   ' Test data validation.
   NewClass.ThisField = 36   ' Works because there is no data validation.
   MsgBox("ThisField = " & NewClass.ThisField)   ' Will print 36.

   ' The attempt to set the field to a value greater than 10 will silently fail.
   NewClass.ThisProperty = 36
   ' The next statement will print the old value of 0 instead.
   MsgBox("ThisProperty = " & NewClass.ThisProperty)
End Sub

Public Class ThisClass
   Public ThisField As Integer   ' This is a field.
   Private m_PropertyValue   As Integer = 0   ' Used for Property Set operations.
   Public Property ThisProperty() As Integer   ' This is a property.
      Get
         Return m_PropertyValue
      End Get
      Set(ByVal Value As Integer)
         ' Only allow Set operation for values less than 10.
         If Value < 10 Then m_PropertyValue = Value
      End Set
   End Property
End Class
```

The procedure `TestFieldsAndProperties` creates an instance of the class and sets and retrieves the values of fields and properties. For the sake of this example, it is assumed that the only valid values are numbers less than 10. Because there is no way to validate values assigned to fields, setting the field to value 36 is allowed. The property, however, only performs assignment for numbers less than 10, so the attempt to set it to 36 is ignored.

Read-Only and Write-Only Properties

Most properties have both **Get** and **Set** property procedures to allow you to both read and modify the value stored inside. However, you can use the **ReadOnly** or **WriteOnly** modifiers to restrict properties from being modified or read. Read-only properties cannot have **Set** property procedures; they are useful for items that you want to expose but not allow to be modified. For example, you could use a read-only property to provide the processor speed of a computer. Write-only properties cannot have **Get** property procedures and are useful for data that you want to store but not expose to other objects. For example, a write-only property could be used to store a password.

> **Note** Previous versions of Visual Basic support **Let** property procedures for use when assigning objects to properties. Visual Basic .NET eliminates the need for **Let** property procedures because object assignment is handled like any other kind of assignment.

See Also

Get Statement | Set Statement | Property Procedures | Property Procedures vs. Fields | ReadOnly | WriteOnly

Property Procedures vs. Fields

Properties and fields both store and retrieve information in an object. Their similarity can make it difficult to determine which is a better programming choice in a given case.

Use property procedures when:

- You need to control when and how a value is set or retrieved.
- The property has a well-defined set of values that need to be validated.
- Setting the value causes some perceptible change in the object's state, such as a visible property.
- Setting the property causes changes to other internal variables or to the values of other properties.
- A set of steps must be performed before the property can be set or retrieved.

Use fields when:

- The value is of a self-validating type. For example, an error or automatic data conversion occurs if a value other than **True** or **False** is assigned to a **Boolean** variable.
- Any value in the range supported by the data type is valid. This is true of many properties of type **Single** or **Double**.
- The property is a **String** data type, and there is no constraint on the size or value of the string.

See Also

Classes vs. Standard Modules | Properties and Property Procedures | Data Types (Conceptual) | Data Types (Language Reference)

Class Methods

The methods of a class are just **Sub** or **Function** procedures declared within the class. For example, to create a `Withdrawal` method for a class named `Account`, you could add this **Public** function to the class module:

```
Public Function WithDrawal(ByVal Amount As Decimal, _
      ByVal TransactionCode As Byte) As Double
   ' Add code here to perform the withdrawal,
   ' return a transaction code,
   ' or to raise an overdraft error.
End Function
```

Shared Methods

Shared methods can be invoked directly from a class variable without first creating an instance of the class. Shared methods are useful when you do not want a method to be associated with a specific instance of a class. Shared methods cannot be declared using the **Overridable**, **NotOverridable**, or **MustOverride** modifiers. Methods declared in modules are implicitly shared and cannot explicitly use the **Shared** modifier.

Example

```
   Class ShareClass
      Shared Sub SharedSub()
         MessageBox.Show("Shared method.")
      End Sub
   End Class

Sub Test()
   ' Create an object variable, but not an instance.
   Dim S As ShareClass
   S.SharedSub ' Call the method.
End Sub
```

Protecting Implementation Details

Utility procedures that are used internally by a class should be declared as **Private**, **Protected**, or **Friend**. Restricting the accessibility of such methods prevents them from being used by other developers, and allows you to make future changes without affecting code that uses the objects.

Hiding the details of an object's implementation is another facet of *encapsulation*. Encapsulation allows you to enhance the performance of methods, or completely change the way a method is implemented, without having to change code that uses the method.

See Also

Adding Events to a Class | Properties vs. Methods | Procedures | Shared Members | Overridable | NotOverridable | MustOverride | Shared | Public | Private | Protected | Friend

Properties vs. Methods

Properties and methods are similar in that both are implemented as procedures that accept arguments. In general, properties store data for an object, and methods are actions an object can be asked to perform. Some object characteristics are obviously properties, such as **Name**, and some are obviously methods, such as **Move** and **Show**. In other cases, it is not obvious which class members should be properties and which should be methods. For example, the **Item** method of a collection class stores and retrieves data and can be implemented as an indexed property. On the other hand, it is also reasonable to implement **Item** as a method.

Property Syntax vs. Method Syntax

One way to decide how to implement a class member is to consider how you want it to be used. Although the syntax for retrieving information from a parameterized property is almost identical to the syntax used with a method implemented as a function, the syntax for modifying such a value is slightly different. For example, if you implement the member of a class as a property, the following syntax describes how it would be used:

```
ThisObject.ThisProperty(Index) = NewValue
```

If you implement the class member as a method, the value being modified must be an argument. The following code fragment describes equivalent syntax usage:

```
ThisObject.ThisProperty(Index,NewValue)
```

Error Messages

Another factor to consider when choosing how to implement a class member is what kind of messages will be generated when your class is used incorrectly. If someone accidentally attempts to assign a value to a read-only property, a different error message is returned than in response to a similar call to a method. Correctly implemented class members return error messages that are easier to interpret.

See Also

Adding Fields and Properties to a Class | Class Methods | Item Property

Default Properties

A property that accepts arguments can be declared as the default property for a class. A *default property* is the property that Microsoft Visual Basic .NET will use when no specific property has been named for an object. Default properties are useful because they allow you to make your source code more compact by omitting frequently used property names.

The best candidates for default properties are those properties that accept parameters and that you think will be used the most often. For example, the **Item** property is a good choice for the default property of a collection class because it is used frequently.

The following rules apply to default properties:

- A type can have only one default property, including properties inherited from a base class. There is one exception to this rule. A default property defined in a base class can be shadowed by another default property in a derived class.
- If a default property from a base class is shadowed by a non-default property in a derived class, the default property is still accessible using default property syntax.
- A default property may not be **Shared** or **Private**.
- If an overloaded property is a default property, all overloaded properties with that same name must also specify **Default**.
- Default properties must accept at least one argument.

Example

The following example declares a property that contains an array of strings as the default property for a class:

```
Class Class2
   ' Define a local variable to store the property value.
   Private PropertyValues As String()
   ' Define the default property.
   Default Public Property Prop1(ByVal Index As Integer) As String
      Get
         Return PropertyValues(Index)
      End Get
      Set(ByVal Value As String)
         If PropertyValues Is Nothing Then
            ' The array contains Nothing when first accessed.
            ReDim PropertyValues(0)
         Else
            ' Re-dimension the array to hold the new element.
            ReDim Preserve PropertyValues(UBound(PropertyValues) + 1)
         End If
         PropertyValues(Index) = Value
      End Set
   End Property
End Class
```

Accessing Default Properties

You can access default properties using abbreviated syntax. For example, the following code fragment uses both standard and default property syntax:

```
Dim C As New Class2()
' The first two lines of code access a property the standard way.
C.Prop1(0) = "Value One"    ' Property assignment.
MessageBox.Show(C.Prop1(0)) ' Property retrieval.

' The following two lines of code use default property syntax.
C(1) = "Value Two"     ' Property assignment.
MessageBox.Show(C(1))  ' Property retrieval.
```

See Also

Default | Default Properties for Your Components | Upgrade Recommendation: Resolve Parameterless Default Properties | Default Property Changes in Visual Basic

Overloaded Properties and Methods

Overloading is the creation of more than one procedure, instance constructor, or property in a class with the same name but different argument types.

Overloading is especially useful when your object model dictates that you employ identical names for procedures that operate on different data types. For example, a class that can display several different data types could have `Display` procedures that look like this:

```
Overloads Sub Display(ByVal theChar As Char)
' Add code that displays Char data.
End Sub
Overloads Sub Display(ByVal theInteger As Integer)
' Add code that displays Integer data.
End Sub
Overloads Sub Display(ByVal theDouble As Double)
' Add code that displays Double data.
End Sub
```

Without overloading, you would need to create distinct names for each procedure, even though they do the same thing, as shown next:

```
Sub DisplayChar(ByVal theChar As Char)
' Add code that displays Char data.
End Sub
Sub DisplayInt(ByVal theInteger As Integer)
' Add code that displays Integer data.
End Sub
Sub DisplayDouble(ByVal theDouble As Double)
' Add code that displays Double data.
End Sub
```

Overloading makes it easier to use properties or methods because it provides a choice of data types that can be used. For example, the overloaded `Display` method discussed previously can be called with any of the following lines of code:

```
Display("9"C) ' Call Display with a literal of type Char.
Display(9)    ' Call Display with a literal of type Integer.
Display(9.9R) ' Call Display with a literal of type Double.
```

At run time, Visual Basic .NET calls the correct procedure based on the data types of the parameters you specify.

Note Overloading, overriding, and shadowing are similar concepts that can be easy to confuse. See Introduction to Objects in Visual Basic for more information.

Overloading Rules

You create an overloaded member for a class by adding two or more properties or methods with the same name. Except for overloaded derived members, each overloaded member must have different parameter lists, and the following items cannot be used as a differentiating feature when overloading a property or procedure:

- Modifiers, such as **ByVal** or **ByRef**, that applies to a member, or parameters of the member.
- Names of parameters
- Return types of procedures

The **Overloads** keyword is optional when overloading, but if any overloaded member uses the **Overloads** keyword, then all other overloaded members with the same name must also specify this keyword.

Derived classes can overload inherited members with members that have identical parameters and parameter types, a process known as *shadowing by name and signature*. If the **Overloads** keyword is used when shadowing by name and signature, the derived classes implementation of the member will be used instead of the implementation in the base class, and all other overloads for that member will be available to instances of the derived class.

If the **Overloads** keyword is omitted when overloading an inherited member with a member that has identical parameters and

parameter types, then the overloading is called *shadowing by name*. Shadowing by name replaces the inherited implementation of a member, and makes all other overloads unavailable to instances of the derived class, and its decedents.

The **Overloads** and **Shadows** modifiers cannot both be used with the same property or method.

Example

The following example creates overloaded methods that accept either a **String** or **Decimal** representation of a dollar amount and return a string containing the sales tax.

To use this example to create an overloaded method

1. Open a new project and add a class named `TaxClass`.
2. Add the following code to the `TaxClass` class.

```
Public Class TaxClass
   Overloads Function TaxAmount(ByVal decPrice As Decimal, _
        ByVal TaxRate As Single) As String
     TaxAmount = "Price is a Decimal. Tax is $" & _
        (CStr(decPrice * TaxRate))
  End Function

  Overloads Function TaxAmount(ByVal strPrice As String, _
        ByVal TaxRate As Single) As String
     TaxAmount = "Price is a String. Tax is $" & _
        CStr((CDec(strPrice) * TaxRate))
   End Function
End Class
```

3. Add the following procedure to your form.

```
Sub ShowTax()
   Const TaxRate As Single = 0.08   ' 8% tax rate
   Dim strPrice As String = "64.00"   ' $64.00 Purchase as a String.
   Dim decPrice As Decimal = 64   ' $64.00 Purchase as a Decimal.
   Dim aclass As New taxclass()
    'Call the same method with two diferent kinds of data.
   MessageBox.Show(aclass.TaxAmount(strPrice, TaxRate))
   MessageBox.Show(aclass.TaxAmount(decPrice, TaxRate))
End Sub
```

4. Add a button to your form and call the `ShowTax` procedure from the `Button1_Click` event of the button.
5. Run the project and click the button on the form to test the overloaded `ShowTax` procedure.

At run time, the compiler chooses the appropriate overloaded function that matches the parameters being used. When you click the button the overloaded method is called first with a `Price` parameter that is a string and the message: "Price is a String. Tax is $5.12" is displayed. `TaxAmount` is called with a Decimal value the second time and the message, "Price is a Decimal. Tax is $5.12" is displayed.

See Also

Shadowing | Sub Statement | Introduction to Objects in Visual Basic | Overriding Properties and Methods | Shadows | ByVal | ByRef | Overloads | Shadows

Overriding Properties and Methods

Derived classes inherit properties and methods defined in their base class. This is useful because it means you can reuse these items when appropriate for the class you are using. If the inherited member cannot be used "as is" you have the option of using the **Overrides** keyword to define a new implementation, provided that the property or method in the base class is marked with the **Overridable** keyword, or shadowing the member by redefining it in the derived class.

In practice, overridden members are often used to implement polymorphism. For more information about polymorphism see Polymorphism.

The following rules apply to overriding methods.

- You can only override members that are marked with the **Overridable** keyword in their base class.
- Properties and methods are **NotOverridable** by default.
- Overridden members must have the same arguments as the inherited members from the base class.
- The new implementation of a member can call the original implementation in the parent class by specifying **MyBase** before the method name.

Note Overloading, overriding, and shadowing are similar concepts that can be easy to confuse. See Introduction to Objects in Visual Basic for more information.

Example

Suppose you want to define classes to handle payroll. You could define a generic Payroll class that contains a RunPayroll method that calculates payroll for a normal week. You could then use Payroll as a base class for a more specialized BonusPayroll class, which could be used when distributing employee bonuses.

The BonusPayroll class can inherit, and override, the `PayEmployee` method defined in the base Payroll class**.**

The following example defines a base class, `Payroll`, and a derived class, `BonusPayroll`, which overrides an inherited method, `PayEmployee`. A procedure, `RunPayroll`, creates and then passes a `Payroll` object and a `BonusPayroll` object to a function, `Pay`, that executes the `PayEmployee` method of both objects.

```
Const BonusRate As Decimal = 1.45
Const PayRate As Decimal = 14.75

Class Payroll
   Overridable Function PayEmployee(ByVal HoursWorked As Decimal, _
         ByVal PayRate As Decimal) As Decimal
      PayEmployee = HoursWorked * PayRate
   End Function
End Class

Class BonusPayroll
   Inherits Payroll
   Overrides Function PayEmployee(ByVal HoursWorked As Decimal, _
         ByVal PayRate As Decimal) As Decimal
      ' The following code calls the original method in the base
      ' class, and then modifies the returned value.
      PayEmployee = MyBase.PayEmployee(HoursWorked, PayRate) * BonusRate
   End Function
End Class

Sub RunPayroll()
   Dim PayrollItem As Payroll = New Payroll()
   Dim BonusPayrollItem As New BonusPayroll()
   Dim HoursWorked As Decimal = 40

   MessageBox.Show("Normal pay is: " & _
   PayrollItem.PayEmployee(HoursWorked, PayRate))
   MessageBox.Show("Pay with bonus is: " & _
   BonusPayrollItem.PayEmployee(HoursWorked, PayRate))
End Sub
```

See Also

Overloaded Properties and Methods |Override Modifiers | Polymorphism | Shadowing

Override Modifiers

You can use the **NotOverridable** and **MustOverride** modifiers to control how properties and methods are overridden in derived classes.

The **NotOverridable** modifier defines a method of a base class that cannot be overridden in derived classes. All methods are **NotOverridable** unless marked with the **Overridable** modifier. You can use the **NotOverridable** modifier when you do not want to allow an **overridden** method to be overridden again in a derived class.

Methods defined with the **MustOverride** modifier have no implementation in the base class and must be implemented in derived classes. Classes that contain **MustOverride** methods must be marked with the **MustInherit** modifier.

Example

```
MustInherit Class BaseClass
   Public MustOverride Sub aProcedure()
End Class

Class DerivedClass
   Inherits BaseClass
   Public NotOverridable Overrides Sub aProcedure()
      ' Override a procedure inherited from the base class
      ' and mark it with the NotOverridable modifier so that
      ' it cannot be overridden in classes derived from this class.
   End Sub
End Class
```

See Also

Overloaded Properties and Methods

Object Management with Collections

Collections provide an ideal way to manage a variety of objects. You can add and remove objects from a collection, retrieve them based on an index or a key, and use **For Each...Next** statements to cycle through the items in your collection. However, the very flexibility of collections can undermine the robustness of your classes.

There are three general approaches you can take to implementing object management using collections. Consider a component that organizes and exposes widgets to client components. To implement this component with collections you might:

- In the `WidgetRepository` class, declare a `Widgets` variable as a collection, and make it public.
- Implement your own `WidgetsCollection` class by inheriting from the **CollectionBase** class. Give the `WidgetRepository` class a property of the `WidgetsCollection` class.
- Implement collection-type functionality in the `WidgetRepository` class by writing the appropriate classes and methods to implement this functionality. This approach is most useful if you want to have collection-type functionality in your class, but cannot inherit from any of the collection-type classes. For example, if you wished to inherit from a class other than a collection class.

You learn how to implement the first strategy in Using Collections In Classes. The second strategy is demonstrated in Walkthrough: Creating Your Own Collection Class.

See Also

Using Collections In Classes | Walkthrough: Creating Your Own Collection Class | Managing Groups of Objects | The Visual Basic .NET Collection Class | For Each...Next Statements | CollectionBase Class

Using Collections in Classes

You can add collections to your classes to manage objects that might be exposed by that class. The simplest way of doing this is to add a public variable of type **Collection** to your class. Consider a hypothetical class called `WidgetRepository` that manages and exposes widgets. You might create a `WidgetCollection` variable to act as a collection of widgets, as shown in the following example:

```
Public Class WidgetRepository
   Public WidgetCollection As New Collection()
   ' Insert code to implement additional functionality.
End Class
```

This class now has a public collection to which Widget objects can be added. But there are a few problems with this approach. First of all, this collection is not strongly typed. What happens if the following code is executed?

```
Dim myString As String = "This is not a Widget object!"
WidgetCollection.Add(myString)
```

The answer is that it is added to the collection, even though it is not a Widget object. In fact, any object can be added to your collection, regardless of type. If you then try to use **For Each...Next** statements to process your widgets, problems ensue, as shown below:

```
Dim aWidget As Widget
For Each aWidget in WidgetCollection
   ' Insert code to process Widgets
Next
```

In the example, the program throws an **ArgumentException** exception at run time, because a member of the collection is not a Widget type.

Another problem involves exposing members of the collection. Since the **Collection** class accepts and returns members of type **Object**, you cannot use any of the functionality of the objects managed by the collection when **Option Strict** statement is **On**. To convert the reference supplied by the collection to the appropriate type, you must instead use the **CType** function, as shown below:

```
Dim myWidget As Widget
MyWidget = CType(WidgetCollection(1), Widget)
```

Note that when **Option Strict** is **Off**, the conversion is performed implicitly, though the late binding causes degradation in performance.

If you need to avoid these problems, or require a more robust type of collection for your class, you should consider using **CollectionBase** to create your own collection class. For details, see the walkthrough, Walkthrough: Creating Your Own Collection Class.

See Also

The Visual Basic .NET Collection Class | Collections in Visual Basic .NET | Option Strict Statement | CType Function | Walkthrough: Creating Your Own Collection Class | For Each...Next Statements

Events and Delegates

Events and delegates are closely associated in Visual Basic .NET. An event is a message sent by an object announcing that something important has happened. Events are implemented using delegates, a form of object-oriented function pointer that allows a function to be invoked indirectly by way of a reference to the function.

In This Section

Events and Event Handlers
Provides an overview of events, event handlers, and associated terminology.
Adding Events to a Class
Describes the process of defining events.
Writing Event Handlers
Demonstrates how to write event handlers using the **Handles** clause or the **AddHandler** statement.
WithEvents and the Handles Clause
Demonstrates how to use the **WithEvents** keyword and the **Handles** clause to associate an event with an event handler.
AddHandler and RemoveHandler
Demonstrates how to use the **AddHandler** and **RemoveHandler** statements to dynamically associate an event with an event handler.
Event Handler Example
Defines a class that raises an event when you call the **CauseEvent** method.
Delegates and the AddressOf Operator
Provides an introduction to delegates and how they are used.
Delegate Example
Creates a sort procedure in a class that uses the standard alphabetic sort on most list boxes, but is able to switch at run time to a custom sort procedure. To do this, you pass the custom sort procedure to the sort class at run time, using delegates.
Walkthrough: Declaring and Raising Events
Steps you through the process of declaring and raising events for a class.
Walkthrough: Handling Events
Demonstrates how to write an event handler procedure.

Related Sections

Handling and Raising Events
Provides an overview of the event model in the .NET Framework.

Event Handling in Windows Forms

Describes how to work with events associated with Windows Forms objects.

Events and Event Handlers

It is tempting to think of a Visual Studio project as a series of procedures that execute in a sequence. In reality, most programs are event driven — meaning the flow of execution is determined by external occurrences called *events*.

An event is a signal that informs an application that something important has happened. For example, when a user clicks a control on a form, the form can raise a **Click** event and call a procedure that handles the event. Events also allow separate tasks to communicate. Say, for example, that your application performs a sort task separately from the main application. If a user cancels the sort, your application can send a cancel event instructing the sort process to stop.

Event Terms and Concepts

This section describes the terms and concepts used with events in Visual Basic .NET.

Declaring Events

You declare events within classes, structures, modules, and interfaces using the **Event** keyword, as in the following example:

```
Event AnEvent(ByVal EventNumber As Integer)
```

Raising Events

An event is like a message announcing that something important has occurred. The act of broadcasting the message is called *raising* the event. In Visual Basic .NET, you raise events with the **RaiseEvent** statement, as in the following example:

```
RaiseEvent AnEvent(EventNumber)
```

Events must be raised within the scope where they are declared. For example, a derived class cannot raise events inherited from a base class.

Event Senders

Any object capable of raising an event is an *event sender*, also known as an *event source*. Forms, controls, and user-defined objects are examples of event senders.

Event Handlers

Event handlers are procedures that are called when a corresponding event occurs. You can use any valid subroutine as an event handler. You cannot use a function as an event handler, however, because it cannot return a value to the event source.

Visual Basic uses a standard naming convention for event handlers that combines the name of the event sender, an underscore, and the name of the event. For example, the click event of a button named `button1` would be named `Sub button1_Click`.

> **Note** It is recommended that you use this naming convention when defining event handlers for your own events, but it is not required; you can use any valid subroutine name.

Associating Events with Event Handlers

Before an event handler becomes usable, you must first associate it with an event by using either the **Handles** or **AddHandler** statement.

The **WithEvents** statement and **Handles** clause provide a declarative way of specifying event handlers. Events raised by an object declared with **WithEvents** can be handled by any subroutine with a **Handles** clause that names this event. Although the **Handles** clause is the standard way of associating an event with an event handler, it is limited to associating events with event handlers at compile time.

The **AddHandler** and **RemoveHandler** statements are more flexible than the **Handles** clause. They allow you to dynamically connect and disconnect the events with one or more event handlers at run time, and they do not require you to declare object variables using **WithEvents**.

In some cases, such as with events associated with forms or controls, Visual Basic .NET automatically stubs out an empty event handler and associates it with an event. For example, when you double-click a command button on a form in design mode, Visual Basic .NET creates an empty event handler and a **WithEvents** variable for the command button, as in the following code:

```
Friend WithEvents Button1 As System.Windows.Forms.Button
Protected Sub Button1_Click(ByVal sender As System.Object, _
   ByVal e As System.EventArgs) Handles Button1.Click
End Sub
```

See Also

AddHandler and RemoveHandler | Writing Event Handlers | Handles | Addhandler Statement

Adding Events to a Class

You add events to a class by declaring them with the **Events** statement. The declaration includes the name of the event and the arguments it uses.

To add an event to a class

- In the Declarations section of the class module that defines the class, use the **Event** statement to declare the event with whatever arguments you want it to have. For example:

```
Public Event PercentDone(ByVal Percent As Single, _
   ByRef Cancel As Boolean)
```

Events cannot have return values, optional arguments, or **ParamArray** arguments.

Adding an event to a class specifies that an object of this class can raise a specific event. To cause an event to actually happen, you must use the **RaiseEvent** statement. You can use **Handles** keyword or **AddHandler** statement to associate the event with an event handler procedure. Events must be raised within the scope where they are declared. For example, a derived class cannot raise events inherited from a base class.

See Also

Walkthrough: Declaring and Raising Events | Walkthrough: Handling Events | Events and Event Handlers | Delegates and the AddressOf Operator | AddHandler and RemoveHandler | Writing Event Handlers | Polymorphism | RaiseEvent Statement | Handles | AddHandler Statement

Writing Event Handlers

The way you construct an event handler depends on how you want to associate it with events. The standard way to create an event handler is to use the **Handles** keyword with the **WithEvents** keyword. Visual Basic .NET provides a second way to handle events: the **AddHandler** statement. **AddHandler** and **RemoveHandler** allow you to dynamically start and stop event handling for a specific event. You can use either approach, but you should not use both **WithEvents** and **AddHandler** with the same event.

Handling Events Using WithEvents

The **WithEvents** keyword allows you to create class or module-level object variables that can be used with the **Handles** clause in event handlers.

To handle events using WithEvents and the Handles clause

1. In the Declarations section of the module that will handle the event, use the **WithEvents** keyword to declare an object variable for the source of your events, as in the following example:

```
Public WithEvents ClassInst As Class1
```

2. In the Code editor, choose the **WithEvents** variable you just declared from the **Class Name** drop-down list on the left.
3. Choose the event you want to handle from the **Method Name** drop-down list on the right. The Code editor creates the empty event handler procedure with a **Handles** clause.

   Note This step is optional. You can create the event handler procedure manually as long as the procedure you create is a subroutine, has the correct argument list to match the event being handled, and has a **Handles** clause that specifies the event being handled.

4. Add event-handling code to the event handler procedure using the supplied arguments. The following code provides an example:

```
Public Sub ClassInst_AnEvent(ByVal EventNumber As System.Integer) _
      Handles ClassInst.AnEvent
   MessageBox.Show("Received event number: " & CStr(EventNumber))
End Sub
```

Handling Events Using AddHandler

You can use the **AddHandler** statement to dynamically connect events with event handler procedures.

To handle events using AddHandler

1. Declare an object variable of the class that is the source of the events you want to handle. Unlike a **WithEvents** variable, this can be a local variable in a procedure; for example:

```
Dim CI As New Class1()
```

2. Use the **AddHandler** statement to specify the name of the event sender, and the **AddressOf** statement to provide the name of your event handler; for example:

```
AddHandler CI.AnEvent, AddressOf EHandler
```

   Any procedure can serve as an event handler as long as it supports the correct arguments for the event being handled.

3. Add code to the event handler, as in the following example:

```
Public Sub EHandler(ByVal EventNumber As Integer)
   MessageBox.Show("Received event number " & CStr(EventNumber))
End Sub
```

Using RemoveHandler to Stop Handling Events

You can use the **RemoveHandler** statement to dynamically disconnect events from event handler procedures.

To stop handling events using RemoveHandler

1. Use the **RemoveHandler** statement to specify the name of the event sender, and the **AddressOf** statement to provide the name of your event handler. The syntax for **RemoveHandler** statements will always closely match the **AddHandler** statement used to start event handling; for example:

```
RemoveHandler CI.AnEvent, AddressOf EHandler
```

Handling Events Inherited from a Base Class

Derived classes — classes that inherit characteristics from a base class — can handle events raised by their base class using the **Handles MyBase** statement.

To handle events from a base class

- Declare an event handler in the derived class by adding a **Handles MyBase.<event name>** statement to the declaration line of your event handler procedure, where **event name** is the name of the event in the base class you are handling. For example:

```
Public Class Class1
   Public Event SE(ByVal i As Integer)
   ' Place methods and properties here.
End Class

Public Class Class2
   Inherits Class1
   Sub EventHandler(ByVal x As Integer) Handles MyBase.SE
   ' Place code to handle events from Class1 here.
   End Sub
End Class
```

See Also

Handles | Addhandler Statement | Events and Event Handlers | Delegates and the AddressOf Operator | AddHandler and RemoveHandler | Writing Event Handlers

WithEvents and the Handles Clause

The **WithEvents** statement and the **Handles** clause provide a declarative way of specifying event handlers. Events raised by an object declared with the **WithEvents** keyword can be handled by any procedure with a **Handles** statement for that event, as shown in the following example:

```
Dim WithEvents EClass As New EventClass() ' Declare a WithEvents variable.
Sub TestEvents()
   EClass.RaiseEvents()
End Sub
' Declare an event handler that handles multiple events.
Sub EClass_EventHandler() Handles Eclass.XEvent, Eclass.YEvent
   MsgBox("Received Event.")
End Sub

Class EventClass
   Public Event XEvent()
   Public Event YEvent()
   Sub RaiseEvents() 'Raises two events handled by EClass_EventHandler.
      RaiseEvent XEvent()
      RaiseEvent YEvent()
   End Sub
End Class
```

The **WithEvents** statement and the **Handles** clause are often the best choice for event handlers, because the declarative syntax they use makes event handling easier to code, read and debug. However, be aware of the following limitations on the use of **WithEvents** variables:

- You cannot use a **WithEvents** variable as a generic object variable. That is, you cannot declare it as **Object** — you must specify the class name when you declare the variable.
- You cannot use **WithEvents** to declaratively handle shared events, since they are not tied to an instance that can be assigned to a **WithEvents** variable.
- You cannot create arrays of **WithEvents** variables.
- **WithEvents** variables allow a single event handler to handle one or more kind of event, or one or more event handlers to handle the same kind of event.

See Also

AddHandler and RemoveHandler | Handles | WithEvents

AddHandler and RemoveHandler

The **AddHandler** statement is similar to the **Handles** clause in that both allow you to specify an event handler that will handle an event. However, **AddHandler** along with **RemoveHandler** provide greater flexibility than the **Handles** clause, allowing you to dynamically add, remove, and change the error handler associated with an event. And unlike **Handles**, **AddHandler** allows you to associate multiple event handlers with a single event.

AddHandler takes two arguments: the name of an event from an event sender such as a control, and an expression that evaluates to a delegate. You do not need to explicitly specify the delegate class when using **AddHandler**, since the **AddressOf** statement always returns a reference to the delegate. The following example associates an event handler with an event raised by an object:

```
AddHandler MyObject.Event1, AddressOf Me.MyEventHandler
```

RemoveHandler, which disconnects an event from an event handler, uses the same syntax as **AddHandler**. For example:

```
RemoveHandler MyObject.Event1, AddressOf Me.MyEventHandler
```

See Also

Addhandler Statement | Events and Event Handlers | WithEvents and the Handles Clause | Writing Event Handlers

Event Handler Example

The following example defines a class that raises an event when you call the `CauseEvent` method. This event is handled by an event handler procedure called`EventHandler`.

To run this example, add the following code to the form class of a Visual Basic .NET Windows Application project, and call the `TestEvents` procedure with an integer argument.

```
Public Class Class1
   ' Declare an event for this class.
   Public Event Event1(ByVal EventNumber As Integer)
   ' Define a method that raises an event.
   Sub CauseEvent(ByVal EventNumber As Integer)
      RaiseEvent Event1(EventNumber)
   End Sub
End Class

Protected Sub TestEvents(ByVal EventNumber As Integer)
   Dim MyObject As New Class1()
   AddHandler MyObject.Event1, AddressOf Me.EventHandler
   ' Cause the object to raise an event.
   MyObject.CauseEvent(EventNumber)
End Sub

Sub EventHandler(ByVal EventNumber As Integer)
   MessageBox.Show("Received event number " & CStr(EventNumber))
End Sub
```

See Also

Events and Event Handlers | Delegates and the AddressOf Operator | AddHandler and RemoveHandler | Writing Event Handlers

Delegates and the AddressOf Operator

Delegates are objects you can use to call the methods of other objects. They are sometimes described as *type-safe function pointers* because they are similar to function pointers used in other programming languages. But unlike function pointers, Visual Basic .NET delegates are a reference type based on the class **System.Delegate**, and can reference both shared methods — methods that can be called without a specific instance of a class — and instance methods.

Delegates are useful in situations where you need an intermediary between a calling procedure and the procedure being called. For example, you might want an object that raises events to be able to call different event handlers under different circumstances. Unfortunately, the object raising events cannot know ahead of time which event handler is handling a specific event. Visual Basic .NET lets you dynamically associate event handlers with events by creating a delegate for you when you use the **AddHandler** statement. At run time, the delegate forwards calls to the appropriate event handler.

Although you can create your own delegates, in most cases Visual Basic .NET creates the delegate and takes care of the details for you. For example, an **Event** statement implicitly defines a delegate class named `<EventName>EventHandler` as a nested class of the class containing the **Event** statement, and with the same signature as the event. The **AddressOf** statement implicitly creates an instance of a delegate. For example, the following two lines of code are equivalent:

```
AddHandler Button1.Click, AddressOf Me.Button1_Click
   ' The previous line of code is shorthand for the next line of code.
AddHandler Button1.Click, New EventHandler(AddressOf Button1_Click)
```

You can use the shorthand way of creating delegates anywhere the compiler can determine the delegate's type by the context.

Declaring Events that Use an Existing Delegate Type

In some situations, you may want to declare an event to use an existing delegate type as its underlying delegate. The following syntax demonstrates how:

```
Event AnEvent As DelegateType
```

This is useful when you want to route multiple events to the same handler.

See Also

Delegate Statement | AddressOf Operator | Events and Event Handlers | AddHandler and RemoveHandler | Writing Event Handlers | Multithreaded Applications

Delegate Example

You can use delegates for other, non-event related tasks, such as free threading or with procedures that need to call different versions of functions at compile time.

For example, suppose you have a classified-ad application that includes a list box with the names of cars. The ads are sorted by title, which is normally the make of the car. A problem you may face occurs when some cars include the year of the car before the make. The problem is that the built-in sort functionality of the list box sorts only by character codes; it places all the ads starting with dates first, followed by the ads starting with the make.

To fix this, you can create a sort procedure in a class that uses the standard alphabetic sort on most list boxes, but is able to switch at run time to the custom sort procedure for car ads. To do this, you pass the custom sort procedure to the sort class at run time, using delegates.

Each delegate class defines a constructor that is passed the specification of an object method. The following code provides an example of a parameter to such a delegate constructor:

```
AddressOf [<expression>.]<method name>
```

The compile-time type of the expression must be a class or an interface with a method by the specified name whose signature matches the signature of the delegate class. The method can be either a shared (class) method or an instance method.

You invoke the method associated with a delegate instance by calling its built-in **Invoke** method, as illustrated in the following example:

```
Delegate Sub MySubDelegate(ByVal x As Integer)
Protected Sub Test()
   Dim c2 As New class2()
   ' Test the delegate.
   c2.DelegateTest()
End Sub

Class class1
   Sub Sub1(ByVal x As Integer)
      MessageBox.Show("The value of x is: " & CStr(x))
   End Sub
End Class

Class class2
   Sub DelegateTest()
      Dim c1 As Class1
      Dim msd As MySubDelegate
      c1 = New Class1()
      ' Create an instance of the delegate.
      msd = AddressOf c1.Sub1
      msd.Invoke(10) ' Call the method.
   End Sub
End Class
```

See Also

Delegate Statement | Delegates and the AddressOf Operator | Events and Event Handlers | AddHandler and RemoveHandler | Writing Event Handlers | Multithreaded Applications

Walkthrough: Declaring and Raising Events

This walkthrough demonstrates how to declare and raise events for a class named Widget. After you complete the steps, you might want to read the companion topic, Walkthrough: Handling Events which shows how to use events from Widget objects to provide status information in an application.

The Widget Class

Assume for the moment that you have a Widget class. Your Widget class has a method that can take a long time to execute, and you want your application to be able to put up some kind of completion indicator.

Of course, you could make the Widget object show a percent-complete dialog box, but then you would be stuck with that dialog box in every project in which you used the Widget class. A good principle of object design is to let the application that uses an object handle the user interface — unless the whole purpose of the object is to manage a form or dialog box.

The purpose of Widget is to perform other tasks, so it is better to add a `PercentDone` event, and let the procedure that calls the Widget's methods handle that event and display status updates. The `PercentDone` event also can provide a mechanism for canceling the task.

To build the code example for this topic

1. Open a new Visual Basic .NET Windows Application project, and create a form named Form1.
2. Add two buttons and a label to Form1.
3. On the **Project** menu, select **Add Class** to add a class to the project.
4. Name the objects as shown in the following table.

Object	Property	Setting
Class	**Name**	Widget
Button1	**Text**	Start Task
Button2	**Text**	Cancel
Label	**Name**, **Text**	lblPercentDone, "0 "

To declare an event for the Widget class

- In the Declarations section of your class module, declare the event using the **Event** keyword. Note that an event can have **ByVal** and **ByRef** arguments, as the Widget's `PercentDone` event demonstrates:

```
Public Event PercentDone(ByVal Percent As Single, _
                         ByRef Cancel As Boolean)
```

When the calling object receives a `PercentDone` event, the `Percent` argument contains the percentage of the task that is complete. The `Cancel` argument can be set to **True** to cancel the method that raised the event.

> **Note** You can declare event arguments just as you do arguments of procedures, with the following exceptions: Events cannot have **Optional** or **ParamArray** arguments, and events do not have return values.

The `PercentDone` event is raised by the `LongTask` method of the Widget class. `LongTask` takes two arguments: the length of time the method will pretend to be doing work, and the minimum time interval before `LongTask` pauses to raise the `PercentDone` event.

To raise the PercentDone event

1. To simplify access to the **Timer** property used by this class, add an **Imports** statement to the top of the declarations section of your class module, above the `Class Widget` statement.

```
Imports Microsoft.VisualBasic.DateAndTime
```

2. Add the following code to the Widget class:

```
Public Sub LongTask(ByVal Duration As Single, _
                    ByVal MinimumInterval As Single)
   Dim Threshold As Single
```

```
    Dim Start As Single
    Dim blnCancel As Boolean

    ' The Timer property of the DateAndTime object returns the seconds
    ' and milliseconds that have passed since midnight.
    Start = CSng(Timer)
    Threshold = MinimumInterval

    Do While CSng(Timer)< (Start + Duration)
        ' In a real application, some unit of work would
        ' be done here each time through the loop.
        If CSng(Timer)> (Start + Threshold) Then
               RaiseEvent PercentDone( _
               Threshold / Duration, blnCancel)
               ' Check to see if the operation was canceled.
               If blnCancel Then Exit Sub
               Threshold = Threshold + MinimumInterval
        End If
      Loop
End Sub
```

The Widget class raises the `PercentDone` event every `MinimumInterval` seconds when your application calls the `LongTask` method. When the event returns, `LongTask` checks to see if the `Cancel` argument was set to **True**.

A few disclaimers are necessary here. For simplicity, the `LongTask` procedure assumes you know in advance how long the task will take. This is almost never the case. Dividing tasks into chunks of even size can be difficult, and often what matters most to users is simply the amount of time that passes before they get an indication that something is happening.

You may have spotted another flaw in this sample. The **Timer** property returns the number of seconds since midnight; therefore, the application gets stuck if it is started just before midnight. A more careful approach to measuring time would take boundary conditions such as this into consideration, or avoid them altogether using properties such as **Now**.

Now that the Widget class can raise events, you can move to the next walkthrough. Walkthrough: Handling Events demonstrates how to use **WithEvents** to associate an event handler with the `PercentDone` event.

See Also

Walkthrough: Handling Events | Events and Event Handlers | Delegates and the AddressOf Operator | AddHandler and RemoveHandler | Writing Event Handlers | Timer Property | WithEvents | Now Property | Event Statement | ByVal | ByRef | Imports Statement

Walkthrough: Handling Events

This is the second of two topics that demonstrate how to work with events. The first topic, Walkthrough: Declaring and Raising Events, showed how to declare and raise events. This section uses the form and class from that walkthrough to show how to handle events when they take place.

The Widget class example uses traditional event-handling statements. Visual Basic .NET provides other techniques for working with events. As an exercise, you can modify this example to use the **AddHandler** and **Handles** statements.

To handle the PercentDone event of the Widget class

- Place the following code in the Declarations section of Form1:

```
Private WithEvents mWidget As Widget
Private mblnCancel As Boolean
```

The **WithEvents** keyword specifies that the variable `mWidget` is used to handle an object's events. You specify the kind of object by supplying the name of the class from which the object will be created.

The variable `mWidget` is declared in the Declarations section of Form1 because **WithEvents** variables must be module-level. This is true regardless of the type of module you place them in.

The variable `mblnCancel` is used to cancel the `LongTask` method.

Writing Code to Handle an Event

As soon as you declare a variable using **WithEvents**, the variable name appears in the left drop-down list of the module's Code editor. When you select `mWidget`, the Widget class's events appear in the right drop-down list. Selecting an event displays the corresponding event procedure, with the prefix `mWidget` and an underscore. All the event procedures associated with a **WithEvents** variable are given the variable name as a prefix.

To handle an event

1. Select `mWidget` from the left drop-down list in the Code editor.
2. Select the `PercentDone` event from the right drop-down list. The Code editor opens the `mWidget_PercentDone` event procedure.
3. Add the following code to the `mWidget_PercentDone` event handler:

```
Private Sub mWidget_PercentDone(ByVal Percent As _
                                Single, ByRef Cancel As _
                                Boolean) _
                                Handles mWidget.PercentDone
   lblPercentDone.Text = CInt(100 * Percent) & "%"
   Application.DoEvents
   If mblnCancel Then Cancel = True
End Sub
```

   Whenever the `PercentDone` event is raised, the event procedure displays the percent complete in a **Label** control. The **DoEvents** statement allows the label to repaint, and also gives the user the opportunity to click the **Cancel** button.

4. Add the following code to the `Button2_Click` event handler:

```
Private Sub Button2_Click(ByVal sender As Object, _
                          ByVal e As System.EventArgs) _
                          Handles Button2.Click
   mblnCancel = True
End Sub
```

If the user clicks the **Cancel** button while `LongTask` is running, the `Button2_Click` event is executed as soon as the **DoEvents** statement allows event processing to occur. The module-level variable `mblnCancel` is set to **True**, and the `mWidget_PercentDone` event then tests it and sets the `ByRef Cancel` argument to **True**.

Connecting a WithEvents Variable to an Object

Form1 is now set up to handle a Widget object's events. All that remains is to find a Widget somewhere.

When you declare a variable **WithEvents** at design time, no object is associated with it. A **WithEvents** variable is just like any other object variable. You have to create an object and assign a reference to it with the **WithEvents** variable.

To create an object and assign a reference to it

1. Select **Base Class Events** for Form1 from the left drop-down list in the Code editor.
2. Select the `Activated` event from the right drop-down list. The Code editor opens the `Form1_Activated` event procedure.
3. Add the following code to the `Form1_Activated` event procedure to create the Widget:

```
Private Sub Form1_Activated(ByVal sender As System.Object, _
                            ByVal e As System.EventArgs) _
                            Handles Form1.Activated
   mWidget = New Widget
End Sub
```

When this code is executed, Visual Basic creates a Widget object and connects its events to the event procedures associated with `mWidget`. From that point on, whenever the Widget raises its `PercentDone` event, the `mWidget_PercentDone` event procedure is executed.

To call the LongTask method

- Add the following code to the `Button1_Click` event handler:

```
Private Sub Button1_Click(ByVal sender As Object, _
                        ByVal e As System.EventArgs) _
                        Handles Button1.Click
   mblnCancel = False
   lblPercentDone.Text = "0%"
   lblPercentDone.Refresh()
   mWidget.LongTask(12.2, 0.33)
   If Not mblnCancel Then lblPercentDone.Text = CStr(100)& "%"
End Sub
```

Before the `LongTask` method is called, the label that displays the percent complete must be initialized, and the module-level **Boolean** flag for canceling the method must be set to **False**.

`LongTask` is called with a task duration of 12.2 seconds. The `PercentDone` event is raised once every one-third of a second. Each time the event is raised, the `mWidget_PercentDone` event procedure is executed.

When `LongTask` is done, `mblnCancel` is tested to see if `LongTask` ended normally, or if it stopped because `mblnCancel` was set to **True**. The percent complete is updated only in the former case.

To run the program

1. Press F5 to put the project in run mode.
2. Click the **Start Task** button. Each time the `PercentDone` event is raised, the label is updated with the percentage of the task that is complete.
3. Click the **Cancel** button to stop the task. Notice that the appearance of the **Cancel** button does not change immediately when you click it. The **Click** event cannot happen until the **Application.DoEvents** statement allows event processing.

You may find it instructive to run the program with F11 and step through the code a line at a time. You can clearly see how execution enters `LongTask`, and then re-enters `Form1` briefly each time the `PercentDone` event is raised.

What would happen if, while execution was back in the code of `Form1`, the `LongTask` method were called again? At worst, a stack overflow might occur if `LongTask` were called every time the event was raised.

You can cause the variable `mWidget` to handle events for a different Widget object by assigning a reference to the new Widget to `mWidget`. In fact, you can make the code in `Button1_Click` do this every time you click the button.

To handle events for a different widget

- Add the following line of code to the `Button1_Click` procedure, immediately preceding the line that reads `mWidget.LongTask(12.2, 0.33)`:

```
mWidget = New Widget      ' Create a new Widget object.
```

The code above creates a new Widget each time the button is clicked. As soon as the `LongTask` method completes, the reference to the Widget is released, and the Widget is destroyed.

A **WithEvents** variable can contain only one object reference at a time, so if you assign a different Widget object to `mWidget`, the previous Widget object's events will no longer be handled. If `mWidget` is the only object variable containing a reference to the old Widget, the object is destroyed. If you want to handle events from several Widget objects, use the **AddHandler** statement to process events from each object separately.

> **Note** You can declare as many **WithEvents** variables as you need, but arrays of **WithEvents** variables are not supported.

See Also

Walkthrough: Declaring and Raising Events | Events and Event Handlers | Delegates and the AddressOf Operator | AddHandler and RemoveHandler | Handles | WithEvents | Writing Event Handlers | Polymorphism

Interfaces in Visual Basic .NET

Interfaces define the properties, methods, and events that classes can implement. Interfaces allow you to define features as small groups of closely related properties, methods, and events; this reduces compatibility problems because you can develop enhanced implementations for your interfaces without jeopardizing existing code. You can add new features at any time by developing additional interfaces and implementations.

Versions of Visual Basic prior to Visual Basic .NET could consume interfaces but not create them directly. Visual Basic .NET introduces the **Interface** statement, which allows you to define true interfaces as distinct entities from classes, and to implement them with an improved version of the **Implements** keyword.

In This Section

Interfaces Overview
Provides an overview of what interfaces are and how Visual Basic .NET implements them.
Interface Definition
Describes how to define interfaces with the **Interface** and **End Interface** statements.
Implements Keyword
Describes how to signify that a section of code implements a specific interface.
Interface Implementation Examples
Provides three examples of interface implementation.
When to Use Interfaces
Discusses when to use interfaces instead of an inheritance hierarchy.
Creating and Implementing an Interface
Provides the steps involved in defining and implementing an interface, plus examples.
Walkthrough: Creating and Implementing Interfaces
Provides a detailed procedure that takes you through the process of defining and implementing your own interface.

Related Sections

Inheritance
Describes how Visual Basic .NET supports inheritance, allowing you to define classes that serve as the basis for derived classes.

Interfaces Overview

Interfaces, like classes, define a set of properties, methods, and events. But unlike classes, interfaces do not provide implementation. They are implemented by classes, and defined as separate entities from classes.

An interface represents a contract, in that a class that implements an interface must implement every aspect of that interface exactly as it is defined.

With interfaces, you can define features as small groups of closely related members. You can develop enhanced implementations for your interfaces without jeopardizing existing code, thus minimizing compatibility problems. You can also add new features at any time by developing additional interfaces and implementations.

Although interface implementations can evolve, interfaces themselves cannot be changed once published. Changes to a published interface may break existing code. If you think of an interface as a contract, it is clear that both sides of the contract have a role to play. The publisher of an interface agrees never to change that interface, and the implementer agrees to implement the interface exactly as it was designed.

Versions of Visual Basic prior to Visual Basic .NET could consume interfaces but not create them directly. Visual Basic .NET allows you to define true interfaces using the **Interface** statement, and to implement interfaces with an improved version of the **Implements** keyword.

See Also

Interface Definition | Interface Statement | Implements Keyword | Interface Implementation Examples | When To Use Interfaces | Creating and Implementing an Interface | Walkthrough: Creating and Implementing Interfaces | Inheritance

Interface Definition

Interface definitions are enclosed within the **Interface** and **End Interface** statements. Following the **Interface** statement, you can add an optional **Inherits** statement that lists one or more inherited interfaces. The **Inherits** statements must precede all other statements in the declaration except comments. The remaining statements in the interface definition should be **Event**, **Sub**, **Function**, and **Property** statements. Interfaces cannot contain any implementation code or statements associated with implementation code, such as **End Sub** or **End Property**.

Interface statements are public by default, but they can also be explicitly declared as **Public**, **Friend**, **Protected**, or **Private**.

> **Note** The only valid modifiers for **Sub**, **Function**, or **Property** statements declared in an interface definition are the **Overloads** and **Default** keywords. None of the other modifiers — **Public**, **Private**, **Friend**, **Protected**, **Shared**, **Static**, **Overrides**, **MustOverride**, or **Overridable** — are allowed.

See Also

Interfaces Overview | Interface Statement | Implements Keyword | Interface Implementation Examples | When To Use Interfaces | Creating and Implementing an Interface | Walkthrough: Creating and Implementing Interfaces | Inheritance | Inherits Statement | Overloads | Default

Implements Keyword

The **Implements** keyword is used to signify that a class member implements a specific interface.

An **Implements** statement requires a comma-separated list of interface members to be implemented. Generally, only a single interface member is specified, but you can specify multiple members. The specification of an interface member consists of the interface name, which must be specified in an implements statement within the class, a period, and the name of the member function, property or event to be implemented. The name of a member that implements an interface member can use any legal identifier, and is not limited to the **InterfaceName_MethodName** convention used in earlier versions of Visual Basic. For example, the following code fragment shows how to declare a subroutine named `Sub1` that implements a method of an interface:

```
Sub Sub1(ByVal i As Integer) Implements interfaceclass.interface2.Sub1
```

The parameter types and return types of the implementing member must match the interface property or member declaration in the interface. The most common way to implement an element of an interface is with a member that has the same name as the interface, as shown in the previous example.

To declare the implementation of an interface method, you can use any attributes that are legal on instance method declarations, including **Overloads**, **Overrides**, **Overridable**, **Public**, **Private**, **Protected**, **Friend**, **Protected Friend**, **MustOverride**, **Default**, and **Static**. The **Shared** attribute is not legal since it defines a class rather than an instance method.

Using **Implements**, you can also write a single method that implements multiple methods defined in an interface, as in the following example:

```
Protected Sub M1 Implements I1.M1, I1.M2, I2.M3, I2.M4
```

You can use a private member to implement a class member. When a private member implements a member of an interface, that member becomes available by way of the interface even though it is not available directly on object variables for the class.

See Also

Interfaces Overview | Interface Definition | Interface Implementation Examples | When To Use Interfaces | Creating and Implementing an Interface | Walkthrough: Creating and Implementing Interfaces | Inheritance | Implements Statement | Implements

Interface Implementation Examples

Classes that implement an interface must implement all its properties, methods, and events.

The following example defines two interfaces. The second interface, `Interface2`, inherits `Interface1` and defines an additional property and method.

```
Interface Interface1
    Sub sub1(ByVal i As Integer)
End Interface

Interface Interface2
    Inherits Interface1 ' Demonstrates interface inheritance.
    Sub M1(ByVal y As Integer)
    ReadOnly Property Num() As Integer
End Interface
```

The next example implements `Interface1`, the interface defined in the previous example:

```
Public Class ImplementationClass1
   Implements Interface1
   Sub Sub1(ByVal i As Integer) Implements Interface1.Sub1
      ' Place code here to implement this method.
    End Sub
End Class
```

The final example implements `Interface2`, including a method inherited from `Interface1`:

```
Public Class ImplementationClass2
   Implements Interface2
   Dim INum As Integer = 0
   Sub sub1(ByVal i As Integer) Implements Interface2.Sub1
      ' Place code here that implements this method.
   End Sub
   Sub M1(ByVal x As Integer) Implements Interface2.M1
      ' Place code here to implement this method.
   End Sub

   ReadOnly Property Num() As Integer Implements _
      Interface2.Num
      Get
        Num = INum
      End Get
   End Property
End Class
```

See Also

Interfaces Overview | Interface Statement | Interface Definition | Implements Keyword | When To Use Interfaces | Creating and Implementing an Interface | Walkthrough: Creating and Implementing Interfaces | Inheritance | Implements Statement | Inherits Statement

When to Use Interfaces

Interfaces are a powerful programming tool because they allow you to separate the definition of objects from their implementation, allowing your objects to evolve without risk of breaking existing applications. Both interfaces and class inheritance each have advantages and disadvantages, and you may end up using a combination of both in your projects. This topic and When to Use Inheritance will help you determine which approach is best for your situation.

Safe Changes to Code

Interfaces eliminate a significant problem of class inheritance: the likelihood of breaking code when you make post-implementation changes to your design. Even though class inheritance allows your classes to inherit implementation from a base class, it also forces you to make most of your design decisions when the class is first published. If your original assumptions turn out to be incorrect, you cannot always change your code safely in later versions.

For instance, suppose you define a method of a base class that expects an **Integer** argument, and then later determine that the argument should be of the **Long** data type. You cannot safely change the original class, because applications designed for classes derived from the original may not compile correctly. This problem can be magnified because a single base class can affect hundreds of subclasses.

One solution is to define a new method that overloads the original and that takes an argument of type **Long**. However, this might not be satisfactory because a derived class may need to override the method that takes the integer, and may not function properly if the method that takes a **Long** is not overridden also. Interfaces solve this problem by allowing you to publish an updated interface that accepts the new data type.

Flexibility in Implementation

There are several other reasons why you might want to use interfaces instead of class inheritance:

- Interfaces are better suited to situations in which your applications require many possibly unrelated object types to provide certain functionality.
- Interfaces are more flexible than base classes because you can define a single implementation that can implement multiple interfaces.
- Interfaces are better in situations in which you do not need to inherit implementation from a base class.
- Interfaces are useful in cases where you cannot use class inheritance. For example, structures cannot inherit from classes, but they can implement interfaces.

See Also

When to Use Inheritance | Interfaces Overview | Interface Definition | Implements Keyword | Interface Implementation Examples | Creating and Implementing an Interface | Walkthrough: Creating and Implementing Interfaces | Inheritance | Integer Data Type | Long Data Type

Creating and Implementing an Interface

As explained in Interfaces Overview, interfaces describe the properties, methods, and events of a class without providing any implementation.

To create an interface

1. Define your interface by adding code to it that begins with the **Interface** keyword and the name of the interface, and ends with the **End Interface** statement. For example, the following code defines an interface named `Encryption`:

```
Interface Encryption
End Interface
```

2. Add statements that define the properties, methods, and events your interface supports. For example, the following code defines two methods, a property, and an event:

```
Interface Encryption
   Function Encrypt(ByVal estring As String) As String
   Function Decrypt(ByVal dstring As String) As String
   Property EncryptionBits() As Integer
   Event FinishedDecoding(ByVal RetVal As Integer)
End Interface
```

To implement an interface

1. If the interface you are implementing is not part of your project, add a reference to the assembly containing the interface.
2. Create a new class that implements your interface, and include the **Implements** keyword in the line following the class name. For example, to implement the `Encryption` interface using **Xor** encryption, you could name the implementation class `XorEnc`, as in the following code:

```
Class XorEnc
   Implements Encryption
End Class
```

3. Add procedures to implement the properties, methods, and events of the class as in the following code, which builds on the example in the previous step:

```
Class XorEnc
   Implements Encryption
   Event FinishedDecoding(ByVal RetVal As Integer) _
      Implements Encryption.FinishedDecoding

   Function Encrypt(ByVal estring As String) _
      As String Implements Encryption.Encrypt
      ' Place code to perform Xor encryption here.
   End Function
   Function Decrypt(ByVal dstring As String) _
      As String Implements Encryption.Decrypt
      ' Place code to perform Xor decryption here.
   End Function

   Property EncryptionBits() As Integer _
      Implements Encryption.EncryptionBits
      Get
         'Place code to return the value of this property here.
      End Get
      Set
         'Place code to set this property here.
      End Set
```

```
    End Property
End Class
```

Note This walkthrough uses simple Xor encryption to demonstrate interface implementation. You should use a more sophisticated form of encryption to store sensitive information.

See Also

Interfaces Overview | Interface Definition | Interface Statement | Implements Keyword | Interface Implementation Examples | When To Use Interfaces | Creating and Implementing an Interface | Walkthrough: Creating and Implementing Interfaces | Inheritance | Xor Operator

Walkthrough: Creating and Implementing Interfaces

Interfaces describe the characteristics of properties, methods, and events, but leave the implementation details up to structures or classes.

This walkthrough demonstrates how to declare and implement an interface.

To define an interface

1. Open a new Windows Application project by clicking **New** on the **File** menu, and then clicking **Project**. The **New Project** dialog box appears.
2. Select **Visual Basic Projects** in the **Project Types** pane.
3. Select Windows Application from the list of Visual Basic project templates.
4. Add a new module to the project by clicking **Add Module** on the **Project** menu.
5. Name the new module `Module1.vb` and click **Open**. The code for the new module is displayed.
6. Define an interface named `MyInterface` within `Module1` by typing `Interface MyInterface` between the **Module** and **End Module** statements and then pressing ENTER. The code editor indents the **Interface** keyword and adds an **End Interface** statement to form a code block.
7. Define a property, method, and event for the interface by placing the following code between the **Interface** and **End Interface** statements:

```
    Property Prop1() As Integer
    Sub Method1(ByVal X As Integer)
    Event Event1()
```

You may notice that the syntax used to declare interface members is different from the syntax used to declare class members. This difference reflects the fact that interfaces cannot contain implementation code.

To implement the interface

1. Add a class named `ImplementationClass` by adding the following statement to `Module1` after the `End Interface` statement but before the `End Module` statement and then pressing **ENTER**:

```
Class ImplementationClass
```

   If you are working within the integrated development environment, the code editor will supply a mathching `End Class` statement when you press **ENTER**.

2. Add the following **Implements** statement to `ImplementationClass`, which names the interface the class implements:

```
Implements MyInterface
```

   When listed separately from other items at the top of a class or structure, the **Implements** statement indicates that the class or structure implements an interface. If you are working within the integrated development environment, the Code Editor may underline the word `MyInterface` and warn you that this class must implement all the members of the interface `MyInterface`. The next two steps in this walkthrough implement these members and take care of this warning.

3. Add the following code to `ImplementationClass` to implement `Event1`.

```
Event Event1() Implements MyInterface.Event1
```

   This code contains an **Event** statement to implement `Event1`, and an **Implements** statement that names the interface and interface member being implemented.

4. Add the following code to `ImplementationClass` to implement the remaining members of the interface:

```
Private pval As Integer ' Holds the value of the property.
Property Prop1() As Integer Implements MyInterface.Prop1
   Get
      Return pval
```

```
    End Get
    Set(ByVal Value As Integer)
       pval = Value
    End Set
End Property

Sub Method1(ByVal X As Integer) Implements MyInterface.Method1
    MsgBox("The X parameter for Method1 is " & X)
    RaiseEvent Event1()
    ' Add more code to implement this method.
End Sub
```

For each member being implemented in this code, an **Implements** statement indicates the name of the interface and member being implemented.

To test the implementation of the interface

1. Right click the startup form for your project in the Solution Explorer and click **View Code**. The editor displays the class for your startup form. By default the startup form is called `Form1`.
2. Add the following event handler to the `Form1` class to handle events raised by instances of `ImplementationClass`:

```
Sub EventHandler()
    MsgBox("The event handler caught the event.")
End Sub
```

3. Add a subroutine named `Test` to the `Form1` class to test the implementation class:

```
Sub Test()
'   Create an instance of the class.
    Dim T As New ImplementationClass()
'   Create a variable of type MyInterface.
    Dim I As MyInterface
'   Assign the class instance to the interface.
'   Calls to the interface members are
'   executed through the class instance.
    I = T
'   Associate an event handler with an event raised by the object.
    AddHandler I.Event1, AddressOf EventHandler
    I.Prop1 = 9 'Set a property.
    MsgBox("Prop1 was set to " & I.Prop1) ' Read the property.
'   Test the method and raise an event.
    I.Method1(5)
End Sub
```

The `Test` procedure creates an instance of the class that implements `MyInterface`, creates a variable of the interface's type, associates an event handler with the event raised by the instance, sets a property, and runs a method through the interface.

4. Add code to call the `Test` procedure from the `Form1_Load` procedure of your startup form:

```
Private Sub Form1_Load(ByVal sender As System.Object, _
                       ByVal e As System.EventArgs) _
                       Handles MyBase.Load
    Test() ' Test the class.
End Sub
```

5. Run the `Test` procedure by pressing F5. The message "Prop1 was set to 9" is displayed. After you click OK, the message "The X parameter for Method1 is 5" is displayed. Click OK and the message "The event handler caught the event" is displayed.

See Also

Implements Statement | Interfaces in Visual Basic .NET | When to Use Interfaces | Creating and Implementing an Interface | Interface Statement | Event Statement

Inheritance

Visual Basic .NET supports *inheritance*, the ability to define classes that serve as the basis for derived classes. Derived classes inherit, and can extend, the properties, methods, and events of the base class. Derived classes can also override inherited methods with new implementations. All classes created with Visual Basic .NET are inheritable by default.

Inheritance allows you to write and debug a class once, and then reuse that code over and over as the basis of new classes. Inheritance also allows you to use inheritance-based *polymorphism*, the ability to define classes that can be used interchangeably by client code at run time, but with functionally different, yet identically named methods or properties.

In This Section

Inheritance Basics
Covers inheritance modifiers, overriding methods and properties, **MyClass**, and **MyBase**.
Using Inherits to Create a Derived Class
Steps you through the process of creating derived classes.
When to Use Inheritance
Discusses when to use inheritance instead of interfaces.
Inheritance and the .NET Framework
Explains how the .NET framework supports inheritance.
Walkthrough: Implementing Inheritance with COM Objects
Describes using existing COM objects as the basis for new objects.
Inheritance and the Base Object Class
Provides an overview of the class that serves as the basis for all other classes.
How New and Finalize Methods Work in a Class Hierarchy
Describes how constructors and destructors are called in a class hierarchy.
Polymorphism
Summarizes polymorphism and how it is used.
Designing an Inheritance Hierarchy
Provides information about designing and implementing a class hierarchy.

Related Sections

Interfaces in Visual Basic .NET
Summarizes what interfaces are and how they are used.
Object-Oriented Programming in Visual Basic
Introduces the concepts of object-oriented programming as they are used in Visual Basic .NET.

Inheritance Basics

The **Inherits** statement is used to declare a new class, called a *derived class*, based on an existing class, known as a *base class*. Derived classes inherit, and can extend, the properties, methods, events, fields, and constants defined in the base class. The following section describes some of the rules for inheritance, and the modifiers you can use to change the way classes inherit or are inherited:

- All classes are inheritable by default unless marked with the **NotInheritable** keyword. Classes can inherit from other classes in your project or from classes in other assemblies that your project references.
- Unlike languages that permit multiple inheritance, Visual Basic .NET permits only single inheritance in classes; that is, derived classes can have only one base class. Although multiple inheritance is not allowed in classes, classes *can* implement multiple interfaces, which can effectively accomplish the same ends.
- To prevent exposing restricted items in a base class, the access type of a derived class must be equal to or more restrictive than its base class. For example, a **Public** class cannot inherit a **Friend** or a **Private** class, and a **Friend** class cannot inherit a **Private** class.

Inheritance Modifiers

Visual Basic .NET introduces the following class-level statements and modifiers to support inheritance:

- **Inherits** statement — Specifies the base class.
- **NotInheritable** modifier — Prevents programmers from using the class as a base class.
- **MustInherit** modifier — Specifies that the class is intended for use as a base class only. Instances of **MustInherit** classes cannot be created directly; they can only be created as base class instances of a derived class. (Other programming languages, such as C++ and C#, use the term *abstract class* to describe such a class.)

Overriding Properties and Methods in Derived Classes

By default, a derived class inherits methods from its base class. If an inherited property or method needs to behave differently in the derived class it can be *overridden*; that is, you can define a new implementation of the method in the derived class. The following modifiers are used to control how properties and methods are overridden:

- **Overridable** — Allows a property or method in a class to be overridden in a derived class.
- **Overrides** — Overrides an **Overridable** property or method defined in the base class.
- **NotOverridable** — Prevents a property or method from being overridden in an inheriting class. **Public** methods are **NotOverridable** by default.
- **MustOverride** — Requires that a derived class override the property or method. When the **MustOverride** keyword is used, the method definition consists of just the **Sub**, **Function**, or **Property** statement. No other statements are allowed, and specifically there is no **End Sub** or **End Function** statement. **MustOverride** methods must be declared in **MustInherit** classes.

For more information about overriding methods see Overriding Properties and Methods.

The MyBase Keyword

You can use the **MyBase** keyword to call methods in a base class when overriding methods in a derived class. For example, suppose you are designing a derived class that overrides a method inherited from the base class. The overridden method can call the method in the base class and modify the return value as shown in the following code fragment:

```
Class DerivedClass
   Inherits BaseClass
   Public Overrides Function CalculateShipping(ByVal Dist As Double, _
                                   ByVal Rate As Double) As Double
      ' Call the method in the base class and modify the return value.
      Return MyBase.CalculateShipping(Dist, Rate) * 2
   End Function
End Class
```

The following list describes restrictions on using **MyBase**:

- **MyBase** refers to the immediate base class and its inherited members. It cannot be used to access **Private** members in the

class.

- **MyBase** is a keyword, not a real object. **MyBase** cannot be assigned to a variable, passed to procedures, or used in an **Is** comparison.
- The method that **MyBase** qualifies does not need to be defined in the immediate base class; it may instead be defined in an indirectly inherited base class. In order for a reference qualified by **MyBase** to compile correctly, some base class must contain a method matching the name and types of parameters appearing in the call.
- You cannot use **MyBase** to call **MustOverride** base class methods.
- **MyBase** cannot be used to qualify itself. Therefore, the following code is illegal:

```
MyBase.MyBase.BtnOK_Click()   ' Syntax error.
```

- **MyBase** cannot be used in modules.
- **MyBase** cannot be used to access base class members that are marked as **Friend** if the base class is in a different assembly.

The MyClass Keyword

The **MyClass** keyword allows you to call an **Overridable** method implemented in your class and make sure that implementation of the method in this class is called rather than an overridden method in a derived class.

- **MyClass** is a keyword, not a real object. **MyClass** cannot be assigned to a variable, passed to procedures, or used in an **Is** comparison.
- **MyClass** refers to the containing class and its inherited members.
- **MyClass** can be used as a qualifier for **Shared** members.
- **MyClass** cannot be used in standard modules.
- **MyClass** can be used to qualify a method that is defined in a base class and that has no implementation of the method provided in that class. Such a reference has the same meaning as **MyBase.Method**.

See Also

Using Inherits to Create a Derived Class | Inherits Statement | Overriding Properties and Methods | Language Changes in Visual Basic

Using Inherits to Create a Derived Class

The **Inherits** statement causes a class to inherit all the non-private members of the specified class.

To inherit from another class

- Add an **Inherits** statement with the name of a class you want to use as a base class as the first statement in your derived class. The **Inherits** statement must be the first non-comment statement after the class statement.

Inheritance Example

The following example defines two classes. The first class is a base class that has two methods. The second class inherits both methods from the base class, overrides the second method, and defines a field named `Field`.

```
Class Class1
   Sub Method1()
      MessageBox.Show("This is a method in the base class.")
   End Sub
   Overridable Sub Method2()
      MessageBox.Show("This is another method in the base class.")
   End Sub
End Class

Class Class2
   Inherits Class1
   Public Field2 as Integer
   Overrides Sub Method2()
      Messagebox.Show("This is a method in a derived class.")
   End Sub
End Class

Protected Sub TestInheritance()
   Dim C1 As New class1()
   Dim C2 As New class2()
   C1.Method1() ' Calls a method in the base class.
   C1.Method2() ' Calls another method from the base class.
   C2.Method1() ' Calls an inherited method from the base class.
   C2.Method2() ' Calls a method from the derived class.
End Sub
```

When you run the procedure `TestInheritance`, you see the following messages:

```
"This is a method in the base class."
"This is another method in the base class."
"This is a method in the base class."
"This is a method in a derived class."
```

See Also

Inheritance | Overriding Properties and Methods | Override Modifiers | Class Properties, Fields, and Methods

When to Use Inheritance

Inheritance is a useful programming concept, but it is easy to use inappropriately. Often interfaces do the job better. This topic and When to Use Interfaces help you understand when each approach should be used.

Inheritance is a good choice when:

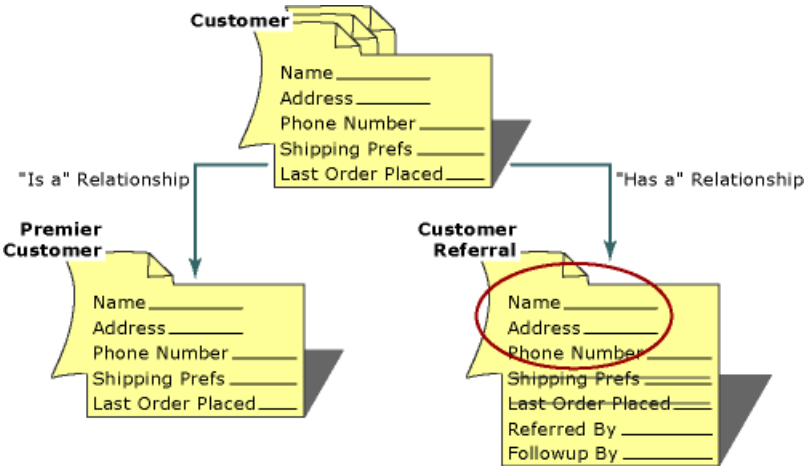
- Your inheritance hierarchy represents an *is-a* relationship and not a *has-a* relationship.
- You can reuse code from the base classes.
- You need to apply the same class and methods to different data types.
- The class hierarchy is reasonably shallow, and other developers are not likely to add many more levels.
- You want to make global changes to derived classes by changing a base class.

These considerations are discussed in order below.

Inheritance and "Is a" Relationships

Two ways to show class relationships in object-oriented programming are "is a" and "has a" relationships. In an "is a" relationship, the derived class is clearly a kind of the base class. For example, a class named PremierCustomer represents an "is a" relationship with a base class named Customer because a premier customer is a customer. However, a class named CustomerReferral represents a "has a" relationship with the Customer class because a customer referral has a customer, but a customer referral is not a kind of customer.

Objects in an inheritance hierarchy should have an "is a" relationship with their base class because they inherit the fields, properties, methods, and events defined in the base class. Classes that represent a "has a" relationship with other classes are not suited to inheritance hierarchies because they may inherit inappropriate properties and methods. For example, if the CustomerReferral class were derived from the Customer class discussed previously, it might inherit properties that make no sense, such as ShippingPrefs and LastOrderPlaced. "Has a" relationships such as this should be represented using unrelated classes or interfaces. The following illustration shows examples of both "is a" and "has a" relationships.



Base Classes and Code Reuse

Another reason to use inheritance is the advantage of code reuse. Well-designed classes can be debugged once and used over and over as a basis for new classes.

A common example of effective code reuse is in connection with libraries that manage data structures. Suppose, for example, that you have a large business application that manages several kinds of in-memory lists. One is an in-memory copy of your customer database, read in from a database at the beginning of the session for speed. The data structure might look something like the following:

```
Class CustomerInfo
   Public PreviousCustomer As CustomerInfo
   Public NextCustomer As CustomerInfo
   Public ID As Integer
   Public FullName As String

   Function InsertCustomer As CustomerInfo
      ' Add code to add a CustomerInfo item to the list.
   End Function
```

```
    Function DeleteCustomer As CustomerInfo
      ' Add code to remove a CustomerInfo item from the list.
    End Function

   Function GetNextCustomer As CustomerInfo
      ' Add code to get the next CustomerInfo item from the list.
   End Function

   Function GetPrevCustomer As CustomerInfo
        'Add code to get the previous CustomerInfo item from the list.
   End Function
End Class
```

Your application may also have a similar list of products the user has added to a shopping cart list, as shown in the following code fragment:

```
Class ShoppingCartItem
   Public PreviousItem As ShoppingCartItem
   Public NextItem As ShoppingCartItem
   Public ProductCode As Integer
   Function GetNextItem As ShoppingCartItem
      ' Add code to get the next ShoppingCartItem from the list.
   End Function
End Class
```

You can see a pattern here: two lists behave the same way (insertions, deletions, and retrievals) but operate on different data types. Maintaining two code bases to perform essentially the same functions is asking for trouble. The most efficient solution is to factor out the list management into its own class, and then inherit from that class for different data types:

```
Class ListItem
   Public PreviousItem As ListItem
   Public NextItem As ListItem
   Function GetNextItem() As ListItem
      ' Add code to get the next item in the list.
   End Function
   Function InsertNextItem As ListItem
      ' Add code to add a item to the list.
   End Function

    Function DeleteNextItem As ListItem
      ' Add code to remove a item from the list.
    End Function

   Function GetPrevItem As ListItem
        'Add code to get the previous item from the list.
   End Function

End Class
```

The `ListItem` class needs only to be debugged once. Then you can build classes that use it without ever having to think about list management again. For example:

```
Class CustomerInfo
   Inherits ListItem
   Public ID As Integer
   Public FullName As String
End Class
Class ShoppingCartItem
   Inherits ListItem
   Public ProductCode As Integer
End Class
```

Although inheritance-based code reuse is powerful tool, it also has associated risks. Even the best-designed systems sometimes

change in ways the designers could not foresee. Changes to an existing class hierarchy can sometime have unintended consequences; some examples are discussed in "The Fragile Base Class Problem," in Class Design Changes After Deployment.

Interchangeable Derived Classes

Derived classes in a class hierarchy can sometimes be used interchangeably with their base class, a process called inheritance-based polymorphism. This approach combines the best features of interface-based polymorphism with the option of reusing or overriding code from a base class.

An example where this can be useful is in a drawing package. For example, consider the following code fragment, which does not use inheritance:

```
Sub Draw(ByVal Shape As DrawingShape, ByVal X As Integer, _
      ByVal Y As Integer, ByVal Size As Integer)
   Select Case Shape.Type
      Case shpCircle
      ' Circle drawing code here.
      Case shpLine
      ' Line drawing code here.
   End Select
End Sub
```

This approach poses some problems. If someone decides to add an ellipse option later, it will be necessary to alter the source code; it is possible that your target users will not even have access to your source code. A more subtle problem is that drawing an ellipse requires another parameter (ellipses have both a major and a minor diameter) that would be irrelevant to the line case. If someone then wants to add a polyline (multiple connected lines), then another parameter would be added, and it would be irrelevant to the other cases.

Inheritance solves most of these problems. Well-designed base classes leave the implementation of specific methods up to the derived classes, so that any kind of shape can be accommodated. Other developers can implement methods in derived classes by using the documentation for the base class. Other class items (such as the x- and y-coordinates) can be built into the base class because all descendants use them. For example, `Draw` could be a **MustOverride** method:

```
MustInherit Class Shape
   Public X As Integer
   Public Y As Integer
   MustOverride Sub Draw()
End Class
```

Then you could add to that class as appropriate for different shapes. For example, a `Line` class might only need a `Length` field:

```
Class Line
   Inherits Shape
   Public Length As Integer
   Overrides Sub Draw()
       ' Place code here to implement Draw for this shape.
   End Sub
End Class
```

This approach is useful because other developers, who do not have access to your source code, can extend your base class with new derived classes as needed. For example, a class named `Rectangle` could be derived from the `Line` class:

```
Class Rectangle
   Inherits Line
   Public Width As Integer
   Overrides Sub Draw()
      ' Place code here to implement Draw for the Rectangle shape.
   End Sub
End Class
```

This example shows how you can move from general-purpose classes to very specific classes by adding implementation details at

each level.

At this point it might be good to reevaluate if the derived class truly represents a "is a" relationship, or have we inadvertently used a "has a" relationship. If the new rectangle class is just composed of lines, then inheritance is not the best choice. However, if the new rectangle is a line with a width property, then the "is a" relationship is maintained.

Shallow Class Hierarchies

Inheritance is best suited for relatively shallow class hierarchies. Excessively deep and complex class hierarchies can be difficult to develop. The decision to use a class hierarchy involves weighing the benefits of using a class hierarchy against complexity. As a general rule, you should limit hierarchies to six levels or fewer. However, the maximum depth for any particular class hierarchy depends on a number of factors, including the amount of complexity at each level.

Global Changes to Derived Classes Through the Base Class

One of the most powerful features of inheritance is the ability to make changes in a base class that propagate to derived classes. When used carefully, you can update the implementation of a single method, and dozens, or even hundreds of derived classes can use the new code. However, this can be a dangerous practice because such changes may cause problems with inherited classes designed by other people. Care must be taken to ensure that the new base class is compatible with classes that use the original. You should specifically avoid changing the name or type of base class members.

Suppose, for example, that you design a base class with a field of type **Integer** to store zip code information, and other developers have created derived classes that use the inherited zip code field. Suppose further that your zip code field stores 5 digits, and the post office has expanded zip codes with a hyphen and 4 more digits. In a worst-case scenario, you could modify the field in the base class to store a 10-character string, but other developers would need to change and recompile the derived classes to use the new size and data type.

The safest way to change a base class is to simply add new members. For example, you could add a new field to store the additional 4 digits in the zip code example discussed previously. That way, client applications can be updated to use the new field without breaking existing applications. The ability to extend base classes in an inheritance hierarchy is an important benefit that does not exist with interfaces.

See Also

When to Use Interfaces | Class Design Changes After Deployment

Inheritance and the .NET Framework

Visual Studio .NET applications can use many of the classes found in the .NET Framework as base classes. Attribute classes, for instance, are always based on the class **System.Attribute**. Other useful .NET Framework base classes include **System.ComponentModel.Component**, which is inherited from to define components, and **System.Exception**, which is inherited from to define custom exception classes.

The .NET Framework is more than a source for base classes; it ensures compatibility between classes created with different programming languages. Visual Basic .NET classes can inherit from classes in assemblies created by any language supported by Visual Studio .NET. The classes you create in Visual Basic .NET can also be used as base classes by other .NET programming languages.

See Also

Walkthrough: Implementing Inheritance with COM Objects | Inheritance and the Base Object Class | When to Use Inheritance

Walkthrough: Implementing Inheritance with COM Objects

You can derive Visual Basic .NET classes from **Public** classes in COM objects, even those created with previous versions of Visual Basic. The properties and methods of classes inherited from COM objects can be overridden or overloaded in the same way that properties and methods of any other base class can be overridden or overloaded. Inheritance from COM objects is useful when you have an existing class library that you do not want to recompile.

The following procedure shows how to use Visual Basic 6.0 to create a COM object containing a class, and then use it as a base class in Visual Basic .NET.

To build the COM object used in this walkthrough

1. In Visual Basic 6.0, open a new ActiveX DLL project. A project named **Project1** is created with a class named `Class1`.
2. In the Project Explorer, right-click **Project1**, and then click **Project1 Properties**. The **Project Properties** dialog box is displayed.
3. On the **General** tab of the **Project Properties** dialog box, change the project name by typing `ComObject1` in the **Project Name** field.
4. In the Project Explorer, right-click `Class1`, and then click **Properties**. The **Properties** window for the class is displayed.
5. Change the **Name** property to `MathFunctions`.
6. In the Project Explorer, right-click `MathFunctions`, and then click **View Code**. The Code editor is displayed.
7. Add a local variable to hold the property value:

```
' Local variable to hold property value
Private mvarProp1 As Integer
```

8. Add Property **Let** and Property **Get** property procedures:

```
Public Property Let Prop1(ByVal vData As Integer)
'Used when assigning a value to the property.
    mvarProp1 = vData
End Property
Public Property Get Prop1() As Integer
'Used when retrieving a property's value.
    Prop1 = mvarProp1
End Property
```

9. Add a function:

```
Function AddNumbers(SomeNumber As Integer, _
                    AnotherNumber As Integer) As Integer
   AddNumbers = SomeNumber + AnotherNumber
End Function
```

10. Create and register the COM object by selecting **Make ComObject1.dll** from the **File** menu.

> **Note** You can also create COM objects using Visual Basic .NET. For details, see COM Interoperability in Visual Basic and Visual C#.

Interop Assemblies

In the following procedure, you will create an interop assembly, which acts as a bridge between unmanaged code (such as a COM object) and the managed code Visual Studio .NET uses. The interop assembly that Visual Basic creates handles many of the details of working with COM objects, such as *interop marshaling,* the process of packaging parameters and return values into equivalent data types as they move to and from COM objects. The reference in the Visual Basic .NET application points to the interop assembly, not the actual COM object.

To use a COM object with Visual Basic .NET

1. Open a new Visual Basic .NET Windows Application project.
2. On the **Project** menu, click **Add Reference**.

The **Add Reference** dialog box is displayed.

3. On the **COM** tab, double click `ComObject1` in the **Component Name** list and click OK.
4. On the **Project** menu, click **Add New Item**.

   The **Add New Item** dialog box is displayed.

5. In the **Templates** pane, click **Class**.

   The default file name, `Class1.vb`, appears in the **Name** field. Change this field to MathClass.vb and click **Open**. This will create a class named `MathClass`, and display its code.

6. Add the following code to the top of `MathClass` to inherit from the COM class.

```
' The inherited class is called MathFunctions in the base class,
' but the interop assembly appends the word Class to the name.
Inherits ComObject1.MathFunctionsClass
```

7. Overload and extend the inherited members by adding the following code to `MathClass`:

```
'  This method overloads the method AddNumbers from the base class.
   Overloads Function AddNumbers(ByRef SomeNumber As Integer, _
                                 ByRef AnotherNumber As Integer _
                                 ) As Integer
      AddNumbers = SomeNumber + AnotherNumber
   End Function

'  The following function extends the inherited class.
   Function SubtractNumbers(ByVal SomeNumber As Integer, _
                            ByVal AnotherNumber As Integer) As Integer
      SubtractNumbers = AnotherNumber - SomeNumber
   End Function
```

The new class inherits the properties of the base class in the COM object, overloads a method, and defines a new method to extend the class.

To test the inherited class

1. Add a button to your startup form, and then double-click it to view its code.
2. In the button's **Click** event handler procedure, add the following code to create an instance of MathClass and call the overloaded methods:

```
Dim Result1 As Integer
Dim Result2 As Short
Dim Result3 As Integer
Dim MathObject As New MathClass()
Result1 = MathObject.AddNumbers(4S, 2S) ' Add two Shorts.
Result2 = MathObject.AddNumbers(4, 2) 'Add two Integers.
Result3 = MathObject.SubtractNumbers(2, 4) ' Subtract 2 from 4.
MathObject.Prop1 = 6 ' Set an inherited property.

MsgBox("Calling the AddNumbers method in the base class " & _
       "using Short type numbers 4 and 2 = " & Result1)
MsgBox("Calling the overloaded AddNumbers method using " & _
       "Integer type numbers 4 and 2 = " & Result2)
MsgBox("Calling the SubtractNumbers method " & _
       "subtracting 2 from 4 = " & Result3)
MsgBox("The value of the inherited property is " & _
        MathObject.Prop1)
```

3. Run the project by pressing F5.

When you click the button on the form, the `AddNumbers` method is first called with **Short** data type numbers, and Visual Basic .NET chooses the appropriate method from the base class. The second call to `AddNumbers` is directed to the overload method from `MathClass`. The third call calls the `SubtractNumbers` method, which extends the class. The property in the base class is set, and the value is displayed.

Data Type Differences

You may have noticed that the overloaded `AddNumbers` function appears to have the same data type as the method inherited from the COM object's base class. This is because the base class method's arguments and parameters are defined as 16-bit integers in Visual Basic 6.0, but they are exposed as 16-bit integers of type **Short** in Visual Basic .NET. The new function accepts 32-bit integers, and overloads the base class function.

When working with COM objects, it is important to verify the size and data types of parameters. For example, when using a COM object that accepts a Visual Basic 6.0 collection object as an argument, you cannot provide a Visual Basic .NET collection. For a more information about data type changes, see Language Changes in Visual Basic.

Overriding Properties and Methods Inherited from COM classes

Properties and methods inherited from COM classes can be overridden, meaning that you can declare a local property or method that replaces a property or method inherited from a base COM class. The rules for overriding inherited COM properties are similar to the rules for overriding other properties and methods with the following exceptions:

- If you override any property or method inherited from a COM class, you must override all the other inherited properties and methods.
- Properties that use ByRef parameters cannot be overridden.

See Also

COM Interoperability in Visual Basic and Visual C# | Language Changes in Visual Basic | Using Inherits to Create a Derived Class | Inherits Statement | Overriding Properties and Methods | Short Data Type

Inheritance and the Base Object Class

Almost everything you do with Visual Basic .NET involves objects, including some entities not normally thought of as objects, such as strings and integers. All objects in Visual Basic .NET are derived from, and inherit methods from, a base class called **System.Object**. The **System.Object** class provides a common base class for all objects in the .NET framework, ensuring interoperability of objects developed using Visual Studio .NET. New classes implicitly inherit the **System.Object** class; it is never necessary to explicitly name this class with an Inherits statement.

Among the methods that objects inherit from **System.Object**, one of the most useful is **Object.GetType**, which is used to return the exact type of the current object.

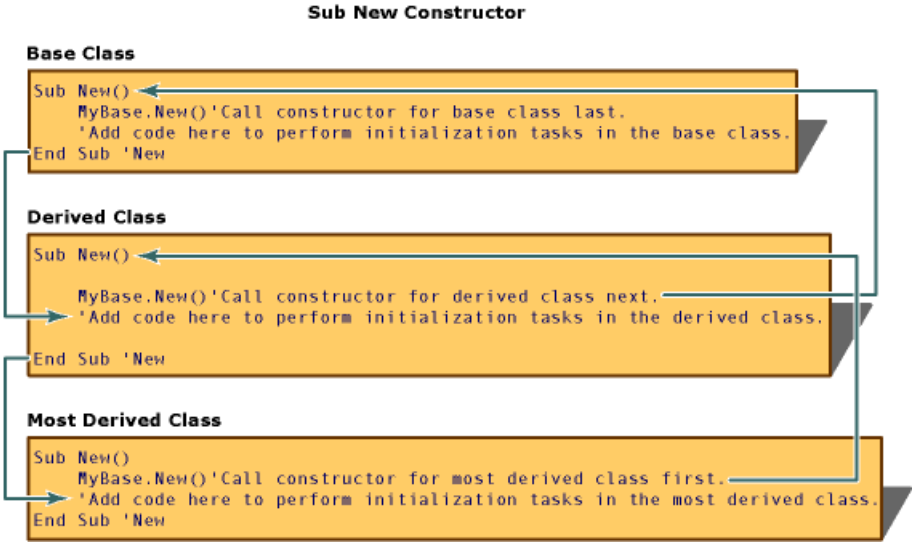
See Also

Object Class | Object.GetType Method | Overriding Properties and Methods | Inheritance Basics

How New and Finalize Methods Work in a Class Hierarchy

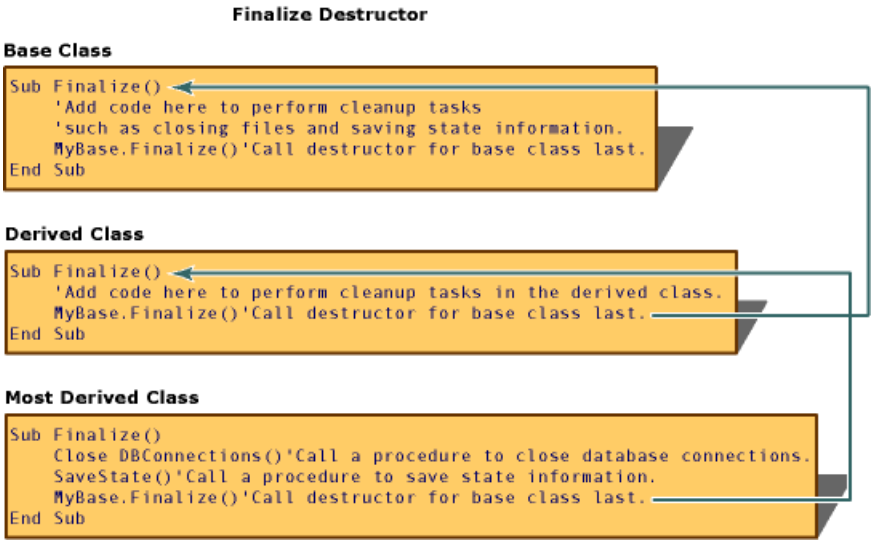
Whenever an instance of a class is created, the common language runtime (CLR) attempts to execute a procedure named **New**, if it exists in that object. **New** is a type of procedure called a *constructor* that is used to initialize new objects before any other code in an object executes. A **New** constructor can be used to open files, connect to databases, initialize variables, and take care of any other tasks that need to be done before an object can be used.

When an instance of a derived class is created, the **Sub New** constructor of the base class executes first, followed by constructors in derived classes. This happens because the first line of code in a **Sub New** constructor uses the syntax `MyBase.New()` to call the constructor of the class immediately above itself in the class hierarchy. The **Sub New** constructor is then called for each class in the class hierarchy until the constructor for the base class is reached. At that point, the code in the constructor for the base class executes, followed by the code in each constructor in all derived classes and the code in the most derived classes is executed last.



When an object is no longer needed, the CLR calls the **Finalize** method for that object before freeing its memory. The **Finalize** method is called a *destructor* because it performs cleanup tasks, such as saving state information, closing files and connections to databases, and other tasks that must be done before releasing the object.

When the **Sub Finalize** method of a derived class is called, it first performs any cleanup tasks that are needed and then uses the syntax `MyBase.Finalize()` to explicitly call the **Sub Finalize** method of its base class. Therefore, **Sub Finalize** methods run from the most-derived classes first, and execute code in the base class last.



See Also

Object Lifetime: How Objects Are Created and Destroyed

Polymorphism

Polymorphism refers to the ability to define multiple classes with functionally different, yet identically named methods or properties that can be used interchangeably by client code at run time.

In This Section

How Visual Basic Provides Polymorphism
Illustrates different approaches to polymorphism.
Inheritance-Based Polymorphism
Discusses using inheritance to achieve polymorphism.
Interface-Based Polymorphism
Discusses using interfaces to achieve polymorphism.

Related Sections

When To Use Interfaces
Discusses when to use interfaces instead of an inheritance hierarchy.
When to Use Inheritance
Provides information about when you should use inheritance instead of interfaces.

How Visual Basic Provides Polymorphism

Traditionally, polymorphism in Visual Basic was accomplished with interfaces, and interfaces can still be used for this purpose. Now, however, Visual Basic .NET provides the option of using inheritance to provide polymorphism as well. As with other issues in object-oriented programming, the option to use depends on your specific requirements. In general, use inheritance when you want to create baseline functionality that derived classes can extend; use interfaces in situations where similar functionality needs to be provided by multiple implementations that have little in common.

See Also

Inheritance-Based Polymorphism | Interface-Based Polymorphism | Designing an Inheritance Hierarchy

Inheritance-Based Polymorphism

Most object-oriented programming systems provide polymorphism through inheritance. Inheritance-based polymorphism involves defining methods in a base class and overriding them with new implementations in derived classes.

For example, you could define a class, `BaseTax`, that provides baseline functionality for computing sales tax in a state. Classes derived from `BaseTax`, such as `CountyTax` or `CityTax`, could implement methods such as `CalculateTax` as appropriate.

Polymorphism comes from the fact that you could call the `CalculateTax` method of an object belonging to any class that derived from `BaseTax`, without knowing which class the object belonged to.

The `TestPoly` procedure in the following example demonstrates inheritance-based polymorphism:

```
Const StateRate As Double = 0.053 ' %5.3 State tax
Const CityRate As Double = 0.028  ' %2.8 City tax
Public Class BaseTax
   Overridable Function CalculateTax(ByVal Amount As Double) As Double
      Return Amount * StateRate ' Calculate state tax.
   End Function
End Class

Public Class CityTax
   ' This method calls a method in the base class
   ' and modifies the returned value.
   Inherits BaseTax
   Private BaseAmount As Double
   Overrides Function CalculateTax(ByVal Amount As Double) As Double
      ' Some cities apply a tax to the total cost of purchases,
      ' including other taxes.
      BaseAmount = MyBase.CalculateTax(Amount)
      Return CityRate * (BaseAmount + Amount) + BaseAmount
   End Function
End Class

Sub TestPoly()
    Dim Item1 As New BaseTax()
    Dim Item2 As New CityTax()
    ShowTax(Item1, 22.74) ' $22.74 normal purchase.
    ShowTax(Item2, 22.74) ' $22.74 city purchase.
 End Sub

Sub ShowTax(ByVal Item As BaseTax, ByVal SaleAmount As Double)
   ' Item is declared as BaseTax, but you can
   ' pass an item of type CityTax instead.
   Dim TaxAmount As Double
   TaxAmount = Item.CalculateTax(SaleAmount)
      MsgBox("The tax is: " & Format(TaxAmount, "C"))
End Sub
```

In this example, the `ShowTax` procedure accepts a parameter named `Item` of type `BaseTax`, but you can also pass any of the classes derived from the shape class, such as `CityTax`. The advantage of this design is that you can add new classes derived from the `BaseTax` class without changing the client code in the `ShowTax` procedure.

See Also

Interface-Based Polymorphism | Designing an Inheritance Hierarchy

Interface-Based Polymorphism

Interfaces provide another way you can accomplish polymorphism in Visual Basic .NET. Interfaces describe properties and methods like classes, but unlike classes, interfaces cannot provide any implementation. Multiple interfaces have the advantage of allowing systems of software components to evolve without breaking existing code.

To achieve polymorphism with interfaces, you implement an interface in different ways in several classes. Client applications can use either the old or the new implementations in exactly the same way. The advantage to interface based polymorphism is that you do not need to re-compile existing client applications to get them to work with new interface implementations.

The following example defines an interface named `Shape2` that is implemented in a class named `RightTriangleClass2` and `RectangleClass2`. A procedure named `ProcessShape2` calls the `CalculateArea` method of instances of `RightTriangleClass2` or `RectangleClass2`:

```
Sub TestInterface()
   Dim RectangleObject2 As New RectangleClass2()
   Dim RightTriangleObject2 As New RightTriangleClass2()
   ProcessShape2(RightTriangleObject2, 3, 14)
   ProcessShape2(RectangleObject2, 3, 5)
End Sub

Sub ProcessShape2(ByVal Shape2 As Shape2, ByVal X As Double, _
      ByVal Y As Double)
   MessageBox.Show("The area of the object is " _
      & Shape2.CalculateArea(X, Y))
End Sub

Public Interface Shape2
   Function CalculateArea(ByVal X As Double, ByVal Y As Double) As Double
End Interface

Public Class RightTriangleClass2
   Implements Shape2
   Function CalculateArea(ByVal X As Double, _
         ByVal Y As Double) As Double Implements Shape2.CalculateArea
      ' Calculate the area of a right triangle.
      Return 0.5 * (X * Y)
   End Function
End Class

Public Class RectangleClass2
   Implements Shape2
   Function CalculateArea(ByVal X As Double, _
         ByVal Y As Double) As Double Implements Shape2.CalculateArea
      ' Calculate the area of a rectangle.
      Return X * Y
   End Function
End Class
```

See Also

Inheritance-Based Polymorphism | Creating and Implementing an Interface

Designing an Inheritance Hierarchy

Inheritance hierarchies are easier to implement than to design, which makes it risky to start coding before clearly identifying your needs. Correcting design mistakes in a class hierarchy after implementation can require code changes that disable existing applications. This section discusses inheritance hierarchy design considerations with an eye to avoiding such mistakes.

In This Section

Class Hierarchy Design Considerations for Extensibility
Discusses how to design class hierarchies so they can be updated or extended by other developers.
Considerations for Choosing Access Levels for Methods
Describes the correct use of accessibility levels within classes.
Class Design Changes After Deployment
Explains issues that developers face when attempting to make changes to a class hierarchy.

Related Sections

When to Use Interfaces
Discusses when to use interfaces instead of an inheritance hierarchy.
Interfaces in Visual Basic .NET
Summarizes how to design and implement interfaces.

Class Hierarchy Design Considerations for Extensibility

Even well-designed class hierarchies need to evolve over time. The choices you make early on when designing a class hierarchy can simplify things later on.

The following list contains some suggestions for making your class hierarchies easier to extend:

- Class hierarchies are defined from the general to the specific. Define classes at each level of an inheritance hierarchy to be as general as possible. Derived classes inherit, and can reuse and extend methods from base classes.

  Suppose, for example, you are designing a class hierarchy that models computer hardware. When you start modeling output devices you could define classes named Display, Printer, and File. You could then define the classes that implement the methods defined in the base classes. For example, the LCDDisplay class could be derived from Display and implement a method named EnterPowerSaveMode.

- Be generous in defining data types and storage to avoid difficult changes later on. For example, you might consider using a variable of type **Long** even though your current data may only require a standard **Integer** variable.
- Only expose items that you are sure derived classes need. **Private** fields and methods reduce naming conflicts and prevent other users from using items that you may need to change down the road.
- Members that are only needed by derived classes should be marked as protected. This reduces the number of classes that are dependent on these members to just derived classes making it easier to update those members during development of the class.

See Also

MustInherit | MustOverride | Considerations for Choosing Access Levels for Methods | Class Design Changes After Deployment

Considerations for Choosing Access Levels for Methods

Applying the optimal access levels to members of a class hierarchy makes the hierarchy easier to maintain by allowing you to control how such members will be used.

As a rule, you should declare class members with access modifiers that provide the least amount of access feasible. Restricting access to class members reduces naming conflicts and prevents methods from being used in unintended ways.

Internal class members should be declared as **Private**; such members are only accessible from within the class in which they are defined.

Methods that are only used within a class or by descendants of a class should use the **Protected** access modifier. A **Protected** member is accessible within the class in which it is declared, and from within any class derived from the class that declared this member.

Friend data members can be accessed from outside a class but only by modules that are part of the project where the class is defined.

Public data members are visible to everyone and are often used at the bottom of a class hierarchy.

See Also

Class Hierarchy Design Considerations for Extensibility | Class Design Changes After Deployment

Class Design Changes After Deployment

Ideally, class hierarchies never change after deployment because even small changes risk unintended consequences. In the real world, such changes are sometimes unavoidable: product requirements can change, and design specifications sometimes omit key elements. One category of inheritance problems is called the *fragile base class problem*.

The Fragile Base Class Problem

The main drawback to using inheritance hierarchies is a problem called *fragile base classes*. Changes to base classes often require the base class and all the code in derived classes to be changed, recompiled, and redistributed. This problem is even more difficult when multiple developers author the base class and derived classes, because each party may deny access to their source code. In a worst-case scenario, a customer might inadvertently use the compiled binary form of an updated base class with the original and incompatible binary version of the derived classes.

Base class changes that can potentially break derived classes include overriding or changing the data type of base class members.

Minimizing Fragile Base Class Problems

The best way to avoid the fragile base class problem is to make changes only in derived classes. However, this not always possible because it requires you to consider every possibility when the base class is first released; despite one's best efforts, unforeseeable changes to the base class are sometimes inevitable.

Using **MustInherit** classes and **MustOverride** methods help reduce fragile base class problems because doing so moves implementation details to derived classes and reduces the need for base class changes. Again, however, this is not always possible, even with the best planning.

Shadowed data members in derived classes are also helpful, because they reduce naming conflicts with base class members.

The safest way to extend the functionality of a base class is to add new members. The only way that new members can break a base class is if the derived class uses the **Overloads** keyword to inherit methods from the base class and introduce new methods with the same name. This problem can be avoided by explicitly specifying in the derived class what base class methods you want to inherit from the base by redefining those methods and delegating to them.

In a sense, all base classes are fragile to some degree. In the final analysis, the fragile base class problem cannot be eliminated, but it can be minimized through careful design of class hierarchies that reduces the need for base class changes, and through thorough testing when such changes are unavoidable.

See Also

Designing an Inheritance Hierarchy | Shadows

COM Interop

The Component Object Model (COM) allows an object to expose its functionality to other components and to host applications. Most of today's software includes COM objects. Although .NET assemblies are the best choice for new applications, you may at times need to employ COM objects. This section covers some of the issues associated with creating and using COM objects with Visual Basic .NET.

In This Section

Introduction to COM Interop
Provides an overview of COM interoperability.
Referencing COM Objects from Visual Basic .NET
Covers how to add references to COM objects that have type libraries.
Working With ActiveX Controls
Demonstrates how to use existing ActiveX controls to add features to the Visual Studio .NET toolbox.
Walkthrough: Creating COM Objects with Visual Basic .NET
Steps you through the process of creating COM objects with and without the COM class template.
Troubleshooting Interoperability
Covers some of the problems you may encounter when interoperating with COM.

Related Sections

Walkthrough: Implementing Inheritance with COM Objects
Demonstrates how to use classes in COM objects as the basis for inheritance in Visual Basic .NET.
Interoperating with Unmanaged Code
Describes interoperability services provided by the common language runtime.
Exposing COM Components to the .NET Framework
Describes the process of calling COM types through COM interop.
Exposing .NET Framework Components to COM
Describes the preparation and use of managed types from COM.
Applying Interop Attributes
Covers attributes you can use when interoperating with unmanaged code.

Introduction to COM Interop

The Component Object Model (COM) allows an object to expose its functionality to other components and to host applications. While COM objects have been fundamental to Windows programming for many years, applications designed for the common language runtime (CLR) used by the .NET Platform offer many advantages.

.NET Platform applications will eventually replace those developed with COM. Until then, you may need to use or create COM objects with Visual Studio .NET. Interoperability with COM, or *COM interop*, enables you to use existing COM objects while transitioning to the .NET Platform at your own pace.

Managed Code and Data

Code developed for the .NET Platform is referred to as *managed code*, and contains metadata that is used by the common language runtime (CLR). Data used by .NET applications is called *managed data* because the .NET runtime manages data-related tasks such as allocating and reclaiming memory, and type checking. By default, Visual Basic .NET uses managed code and data, but you can access the unmanaged code and data of COM objects using interop assemblies (described later in this topic).

Assemblies

An assembly is the primary building block of a .NET Framework application. It is a collection of functionality that is built, versioned, and deployed as a single implementation unit containing one or more files. Each assembly contains an assembly manifest.

Type Libraries and Assembly Manifests

Type libraries describe characteristics of COM objects, such as member names and data types. Assembly manifests perform the same function for .NET applications. They include information about:

- Assembly identity, version, culture, and digital signature
- Files that make up the assembly implementation
- Types and resources that make up the assembly, including those that are exported from it
- Compile-time dependencies on other assemblies
- Permissions required for the assembly to run properly

For more information on assemblies and assembly manifests, see Assemblies.

Importing and Exporting Type Libraries

Visual Studio .NET contains a utility, Tlbimp, that lets you import information from a type library into a .NET application. You can generate type libraries from assemblies by using the Tlbexp utility.

For information on Tlbimp and Tlbexp, see Type Library Importer (Tlbimp.exe) and Type Library Exporter (Tlbexp.exe).

Interop Assemblies

Interop assemblies are .NET assemblies that act as a bridge between managed and unmanaged code, mapping COM object members to equivalent .NET managed members. Interop assemblies created by Visual Basic .NET handle many of the details of working with COM objects, such as interoperability marshaling.

Interoperability Marshaling

All .NET applications share a set of common types that allow interoperability of objects, regardless of the programming language used. The parameters and return values of COM objects sometimes use data types that are different than those used in managed code. *Interoperability marshaling* is the process of packaging parameters and return values into equivalent data types as they move to and from COM objects. For more information, see Interop Marshaling Overview.

See Also

COM Interop | Walkthrough: Implementing Inheritance with COM Objects | Interoperating with Unmanaged Code | Troubleshooting Interoperability | Assemblies | Type Library Importer (Tlbimp.exe) | Type Library Exporter (Tlbexp.exe) | Interop Marshaling Overview

Referencing COM Objects from Visual Basic .NET

In Visual Basic .NET, adding references to COM objects that have type libraries is similar to doing so in previous versions of Visual Basic. However, Visual Basic .NET adds the creation of an interop assembly to the procedure. References to the members of the COM object are routed to the interop assembly and then forwarded to the actual COM object. Responses from the COM object are routed to the interop assembly and forwarded to your .NET application.

To add references to COM objects

1. On the **Project** menu, select **Add Reference** and then click the **COM** tab.
2. Select the component you want to use from the list of COM objects.
3. To simplify access to the interop assembly, add an **Imports** statement to the top of the class or module in which you will use the COM object.

Manually Creating Interop Assemblies Using Tlbimp

Visual Basic .NET automatically creates interop assemblies when you add a reference to a type library from the integrated development environment (IDE). When working from the command line, you can use the Tlbimp utility to manually create interop assemblies.

To create an interop assembly using Tlbimp

1. Add the location of Tlbimp to the DOS search path, if it is not already part of the search path and you are not currently in the directory where it is located.
2. Call Tlbimp from a DOS prompt, providing the following information:
   - Name and location of the DLL that contains the type library
   - Name and location of the namespace where the information should be placed
   - Name and location of the target interop assembly

The following code provides an example:

```
Tlbimp test3.dll /out:NameSpace1 /out:Interop1.dll
```

You can use Tlbimp to create interop assemblies for type libraries, even for unregistered COM objects. However, the COM objects referred to via interop assemblies must be properly registered on the computer where they are to be used. You can register a COM object by using the Regsvr32 utility included with the Windows operating system.

See Also

COM Interop | Type Library Importer (Tlbimp.exe) | Type Library Exporter (Tlbexp.exe) |
Walkthrough: Implementing Inheritance with COM Objects | Troubleshooting Interoperability | Imports Statement

Working with ActiveX Controls

ActiveX controls are COM components or objects you can insert into a Web page or other application to reuse packaged functionality someone else programmed. You can use ActiveX controls developed for previous versions of Visual Basic to add features to the toolbox of Visual Studio .NET.

To add ActiveX controls to the toolbox

1. On the **Tools** menu, select **Add/Remove Toolbox Items**. The **Customize Toolbox** dialog is displayed.
2. Click the **COM Components** tab.
3. Select the check box next to the ActiveX control you want to use, and then click **OK**. The new control appears with the other tools in the toolbox.

> **Note** You can use the Aximp utility to manually create an interop assembly for ActiveX controls. For details, see Windows Forms ActiveX Control Importer (Aximp.exe).

See Also

COM Interop | Customize Toolbox Dialog Box | Adding ActiveX Controls to Windows Forms | Windows Forms ActiveX Control Importer (Aximp.exe) | Considerations When Hosting an ActiveX Control on a Windows Form | Troubleshooting Interoperability

Walkthrough: Calling Windows APIs

Windows APIs are dynamic link libraries (DLLs) that are part of the Windows operating system. You use them to perform tasks when it is difficult to write equivalent procedures of your own. For example, Windows provides a function named `FlashWindowEx` that lets you make the title bar for an application alternate between light and dark shades.

The advantage of using Windows APIs in your code is that they can save development time because they contain dozens of useful functions that are already written and waiting to be used. The disadvantage is that Windows APIs can be difficult to work with and unforgiving when things go wrong.

Windows dynamic-link libraries (DLLs) represent a special category of interoperability. Windows APIs do not use managed code, do not have built-in type libraries, and use data types that are different than those used with Visual Studio .NET. Because of these differences, and because Windows APIs are not COM objects, interoperability with Windows APIs and the .NET Platform is performed using platform invoke, or PInvoke. Platform invoke is a service that enables managed code to call unmanaged functions implemented in DLLs. For more information, see Consuming Unmanaged DLL Functions. You can use PInvoke in Visual Basic .NET by using either the **Declare** statement or applying the **DllImport** attribute to an empty procedure.

Windows API calls were an important part of Visual Basic programming in the past, but are seldom necessary with Visual Basic .NET. Whenever possible, you should use managed code from the .NET Framework to perform tasks instead of Windows API calls. This walkthrough provides information for those situations in which using Windows APIs is unavoidable.

API Calls Using Declare

The most common way to call Windows APIs is via the **Declare** statement.

To declare a DLL procedure

1. Determine the name of the function you want to call plus its arguments, argument types, and return value, as well as the name and location of the DLL that contains it.

   > **Note** For complete information about the Windows APIs, see the Win32 SDK documentation in the Platform SDK Windows API. For more information about the constants that Windows APIs use, examine the header files such as Windows.h included with the Platform SDK.

2. Open a new Windows Application project by clicking **New** on the **File** menu, and then clicking **Project**. The **New Project** dialog box appears.
3. Select **Windows Application** from the list of Visual Basic project templates. The new project is displayed.
4. Add the following **Declare** function either to the declaration section of the startup form for your project, or to the declaration section of the class or module where you want to use the DLL:

```
Declare Auto Function MBox Lib "user32.dll" _
Alias "MessageBox" (ByVal hWnd As Integer, _
   ByVal txt As String, ByVal caption As String, _
   ByVal Typ As Integer) As Integer
```

Parts of the Declare Statement

The **Declare** statement includes the following elements.

Auto modifier

The **Auto** modifier instructs the runtime to convert the string based on the method name according to common language runtime rules (or alias name if specified).

Lib and Alias keywords

The name following the **Function** keyword is the name your program uses to access the imported function. It can be the same as the real name of the function you are calling, or you can use any valid procedure name and then employ the **Alias** keyword to specify the real name of the function you are calling.

Specify the **Lib** keyword followed by the name and location of the DLL that contains the function you are calling. You do not need to specify the path for files located in the windows system directories.

Use the **Alias** keyword if the name of the function you are calling is not a valid Visual Basic procedure name, or conflicts with the name of other items in your application. **Alias** indicates the true name of the function being called.

Argument and Data Type Declarations

Declare the arguments and their data types. This part can be challenging because the data types that Windows uses do not correspond to Visual Studio data types. Visual Basic does a lot of the work for you by converting arguments to compatible data types, a process called *marshaling*. You can explicitly control how arguments are marshaled by using the **MarshalAs** attribute defined in the **System.Runtime.InteropServices** namespace.

> **Note** Previous versions of Visual Basic allowed you to declare parameters **As Any**, meaning that data of any data type could be used. Visual Basic .NET requires that you use a specific data type for all declare statements.

Windows API Constants

Some arguments are combinations of constants. For example, the **MessageBox** API shown in this walkthrough accepts an integer argument called `Typ` that controls how the message box is displayed. You can determine the numeric value of these constants by examining the **#define** statements in the file WinUser.h. The numeric values generally are shown in hexadecimal, so you may want to use a calculator to add them and convert to decimal. For example, if you want to combine the constants for the exclamation style **MB_ICONEXCLAMATION** 0x00000030 and the Yes/No style **MB_YESNO** 0x00000004, you can add the numbers and get a result of 0x00000034, or 52 decimal. Although you can use the decimal result directly, it is better to declare these values as constants in your application and combine them using the **Or** operator.

To declare constants for Windows API calls

1. Consult the documentation for the Windows function you are calling and determine the name the constants it uses, and the name of the .h file that contains the numeric values for these constants.
2. Use a text editor, such as Notepad, to view the contents of the .h file and find the values associated with the constants you are using. For example, the **MessageBox** API uses the constant **MB_ICONQUESTION** to show a question mark in the message box. The definition for MB_ICONQUESTION is in WinUser.h and appears as follows:

```
#define MB_ICONQUESTION             0x00000020L
```

3. Add equivalent **Const** statements to the declarations section of your class or module to make these constants available to your application. For example:

```
Const MB_ICONQUESTION = &H20L
Const MB_YESNO = &H4
Const IDYES = 6
Const IDNO = 7
```

To call the DLL procedure

1. Add a button named `Button1` to the startup form for your project, and then double-click it to view its code. The event handler for the button is displayed.
2. Add code to the **Click** event handler for the button you added to call the procedure, and provide the appropriate arguments:

```
Private Sub Button1_Click(ByVal sender As System.Object, _
                          ByVal e As System.EventArgs) _
                          Handles Button1.Click
   Dim RetVal As Integer ' Stores the return value.
   RetVal = MBox(0, "Declare DLL Test", "Windows API MessageBox", _
                 MB_ICONQUESTION Or MB_YESNO)
' Check the return value.
   If RetVal = IDYES Then
      MsgBox("You chose Yes")
   Else
      MsgBox("You chose No")
   End If
End Sub
```

3. Run the project by pressing F5. The message box is displayed with both **Yes** and **No** response buttons. Click either one.

Data Marshaling

Visual Basic .NET automatically converts the data types of parameters and return values for Windows API calls, but you can use the **MarshalAs** attribute to explicitly specify unmanaged data types that an API expects. For more information about interop marshaling, see Interop Marshaling

To use Declare and MarshalAs in an API call

1. Determine the name of the function you want to call, plus its arguments, data types, and return value.
2. To simplify access to the **MarshalAs** attribute, add an **Imports** statement to the top of the code for the class or module, as in the following example:

```
Imports System.Runtime.InteropServices
```

3. Add a function prototype for the imported function to the declarations section of the class or module you are using, and apply the **MarshalAs** attribute to the parameters or return value. In the following example, an API call that expects the type **Void*** is marshaled as **UnmanagedType.AsAny**:

```
Declare Sub SetData Lib "..\LIB\UnmgdLib.dll" (ByVal x As Short, _
      <MarshalAsAttribute(UnmanagedType.AsAny)> ByVal o As Object)
```

API Calls Using DllImport

The **DllImport** attribute provides a second way to call functions in DLLs without type libraries. **DllImport** is roughly equivalent to using a **Declare** statement but provides more control over how functions are called.

You can use **DllImport** with most Windows API calls as long as the call refers to a shared (sometimes called *static*) method. You cannot use methods that require an instance of a class. Unlike **Declare** statements, **DllImport** calls cannot use the **MarshalAs** attribute.

To call a Windows API using the DllImport attribute

1. Open a new Windows Application project by clicking **New** on the **File** menu, and then clicking **Project**. The **New Project** dialog box appears.
2. Select **Windows Application** from the list of Visual Basic project templates. The new project is displayed.
3. Add a button named `Button2` to the startup form.
4. Double-click `Button2` to open the code view for the form.
5. To simplify access to **DllImport**, add an **Imports** statement to the top of the code for the startup form class:

```
Imports System.Runtime.InteropServices
```

6. Declare an empty function preceding the **End Class** statement for the form, and name the function `MoveFile`.
7. Apply the **Public** and **Shared** modifiers to the function declaration and set parameters for `MoveFile` based on the arguments the Windows API function uses:

```
Public Shared Function _
MoveFile(ByVal src As String, ByVal dst As String) As Boolean
   ' Leave the body of the function empty.
End Sub
```

   Your function can have any valid procedure name; the **DllImport** attribute specifies the name in the DLL. It also handles interoperability marshaling for the parameters and return values, so you can choose Visual Studio .NET data types that are similar to the data types the API uses.

8. Apply the **DllImport** attribute to the empty function. The first parameter is the name and location of the DLL containing the function you are calling. You do not need to specify the path for files located in the Windows system directories. The second parameter is a named argument that specifies the name of the function in the Windows API:

```
<DllImport("KERNEL32.DLL", EntryPoint:="MoveFileW", SetLastError:=True, _
```

```
CharSet:=CharSet.Unicode, ExactSpelling:=True, _
CallingConvention:=CallingConvention.StdCall)> _
Public Shared Function _
MoveFile(ByVal src As String, ByVal dst As String) As Boolean
   ' This function copies a file from the path src to the path dst.
   ' Leave function empty - DllImport attribute forces calls
   ' to MoveFile to be forwarded to MoveFileW in KERNEL32.DLL.
End Function
```

9. Add code to the **Button2_Click** event handler to call the function:

```
Private Sub Button2_Click(ByVal sender As System.Object, _
                          ByVal e As System.EventArgs) _
                          Handles Button2.Click
   Dim RetVal As Boolean
   RetVal = MoveFile("c:\tmp\Test.txt", "c:\Test.txt")
   If RetVal = True Then
      MsgBox("The file was moved successfully.")
   Else
      MsgBox("The file could not be moved.")
   End If
End Sub
```

10. Create a file named Test.Txt and place it in the C:\Tmp directory on your hard drive. Create the Tmp directory if necessary.
11. Press F5 to start the application. The main form appears.
12. Click `Button2`. The message "The file was moved successfully" is displayed if the file can be moved.

See Also

Anatomy of a Declare Statement | Declare Statement | DllImportAttribute Class | MarshalAsAttribute Class | Auto | Alias | Lib | COM Interop | Creating Prototypes in Managed Code | Callback Sample

Walkthrough: Creating COM Objects with Visual Basic .NET

Although .NET assemblies are the best choice for new applications, you may at times need to employ COM objects instead. This walkthrough demonstrates two ways to create COM objects with Visual Basic .NET: with and without the COM class template.

The easiest way to create COM objects is by using the COM class template. The COM class template creates a new class, then configures your project to generate the class as a COM object and register it with the operating system.

To create a COM object using the COM class template

1. Open a new Windows Application project by clicking **New** on the **File** menu, and then clicking **Project**. The **New Project** dialog box appears.
2. With **Visual Basic Projects** highlighted in the **Project Types** list, select **Class Library** from the **Templates** list, and then click **OK**. The new project is displayed.
3. Select **Add New Item** from the **Project** menu. The **Add New Item** dialog box is displayed.
4. Select **COM Class** from the **Templates** list, and then click **Open**. Visual Basic .NET adds a new class and configures the new project for COM interop.
5. Add code, such as properties, methods, and events to the COM class.
6. Select **Build Solution** from the **Build** menu. Visual Basic .NET builds the assembly and registers the COM object with the operating system.

Creating COM Objects Without the COM Class Template

You can create a COM class manually instead of using the COM class template. This procedure can be helpful when working from the command line or when you want more control over how COM objects are defined.

To set up your project to generate a COM object

1. Open a new Windows Application project by clicking **New** on the **File** menu, and then clicking **Project**. The **New Project** dialog box appears.
2. With **Visual Basic Projects** highlighted in the **Project Types** list, select **Class Library** from the **Templates** list, and then click **OK**. The new project is displayed.
3. In the Solution Explorer, right-click your project, and then click **Properties**. The **Property Pages** dialog box is displayed.
4. Click the **Configuration Properties** folder, and then click the **Build** folder. The **Build** properties are displayed.
5. Check the **Register for COM Interop** check box.

To set up the code in your class to create a COM object

1. In the Solution Explorer, double-click Class1.vb to display its code.
2. Rename the class.
3. Add the **ComClass** attribute to the class, specifying the guids for the Class ID, Interface ID, and events Events ID as in the following example:

```
<ComClass(ComClass1.ClassId, ComClass1.InterfaceId, ComClass1.EventsId)> _
Public Class ComClass1
```

4. Supply the Globally Unique Identifier (GUID) constants for the **ComClass** attribute. To create GUIDs for your COM object, click **Create GUID** on the **Tools** menu, or launch guidgen.exe to start the guidgen utility. Select **Registry Format** from the list of formats provided by the guidgen application. Click the **New GUID** button to generate the GUID and click the **Copy** button to copy the GUID to the clipboard. Paste the GUID into the Visual Studio .Net code editor. Remove the leading and trailing braces from the GUID provided. For example, if the GUID provided by guidgen is `"{2C8B0AEE-02C9-486e-B809-C780A11530FE}"` then the GUID should appear as: `"2C8B0AEE-02C9-486e-B809-C780A11530FE"`.
5. Click the **New GUID** and **Copy** buttons until you have generated GUIDS for the ClassId, InterfaceId, and EventsId. For example:

```
Public Const ClassId As String = "832F34A5-5CF5-403f-B4A8-428C8351FD02"
Public Const InterfaceId As String = "3D8B5BA4-FB8C-5ff8-8468-11BF6BD5CF91"
Public Const EventsId As String = "2B691787-6ED7-401e-90A4-B3B9C0360E31"
```

6. COM classes must have a parameterless `Public Sub New()` constructor, or the class will not register correctly. Add a

parameterless constructor to the class:

```
Public Sub New()
   MyBase.New()
End Sub
```

7. Add properties, methods and events to the class, ending it with an **End Class** statement. Select **Build Solution** from the **Build** menu. Visual Basic .NET builds the assembly and registers the COM object with the operating system.

   Note The COM objects you generate with Visual Studio .NET cannot be used by other Visual Studio .NET applications. Attempts to add references to such COM objects raise an error.

See Also

COM Interop | ComClassAttribute Class | Walkthrough: Implementing Inheritance with COM Objects | #Region Directive

Troubleshooting Interoperability

When interoperating between COM and the managed code of the .NET platform, you may encounter one or more of the following common issues.

Interop Marshaling

Exporting Fixed-Length Strings to Unmanaged Code

Exporting Inheritance Hierarchies

Use of COM Objects Through Interop Assemblies

Classes Exposed as Both Interfaces and Classes

Creating Instances of a .NET Class

Data Types for Parameters and Return Values

Module level COM Methods

Unhandled Errors in Event Handlers

ActiveX Control Issues

Passing ReadOnly Properties of Controls ByRef

Deploying Assemblies That Expose Interop

Interop Marshaling

At times, you may need to use data types that are not part of the .NET Framework. Interop assemblies handle most of the work for COM objects, but you may need to control the data types used when managed objects are exposed to COM. For example, structures in class libraries must specify the **BStr** unmanaged type on strings sent to COM objects created by previous versions of Visual Basic. In such cases, you can use the **MarshalAs** attribute to cause managed types to be exposed as unmanaged types.

Exporting Fixed-Length Strings to Unmanaged Code

In previous versions of Visual Basic, strings are exported to COM objects as sequences of bytes without a null termination character. For compatibility with other languages, Visual Basic .NET includes a termination character when exporting strings. The best way to address this incompatibility is to export strings lacking the termination character as arrays of **Byte** or **Char**.

Exporting Inheritance Hierarchies

Managed class hierarchies flatten out when exposed as COM objects. For example, if you define a base class with a member, and then inherit the base class in a derived class that is exposed as a COM object, clients that use the derived class in the COM object will not be able to use the inherited members. Base class members are accessible from COM objects only as instances of a base class, and then only if the base class is also created as a COM object.

Use of COM Objects Through Interop Assemblies

You use interop assemblies almost as though they are managed code replacements for the COM objects they represent. However, because they are wrappers and not actual COM objects, there are some differences between using interop assemblies and standard assemblies. These areas of difference include the exposure of classes, and data types for parameters and return values.

Classes Exposed as Both Interfaces and Classes

Unlike classes in standard assemblies, COM classes are exposed in interop assemblies as both an interface and a class that represents the COM class. The interface's name is identical to that of the COM class. The name of the interop class is the same as that of the original COM class, but with the word "Class" appended. For example, suppose you have a project with a reference to an interop assembly for a COM object. If the COM class is named `MyComClass`, IntelliSense and the Object Browser show an interface named `MyComClass` and a class named `MyComClassClass`.

Creating Instances of a .NET Class

Generally, you create an instance of a .NET class using the **New** statement with a class name. Having a COM class represented by an interop assembly is the one case in which you can use the **New** statement with an interface. Unless you are using the COM class with an **Inherits** statement, you can use the interface the same way as you would a class, as in the following code:

```
Dim Class1 As New MyComClass()
```

However, if you are using the COM class as the base for a derived class, you must use the interop class that represents the COM class, as in the following code:

```
Class DerivedClass
   Inherits MyComClass.AddClassClass
End Class
```

> **Note** Interop assemblies implicitly implement interfaces that represent COM classes. You should not attempt to use the implements statement to implement these interfaces or an error will result.

Data Types for Parameters and Return Values

Unlike members of standard assemblies, interop assembly members may have data types that are different than those used in the object's original declaration. Although interop assemblies implicitly convert COM types to compatible common language runtime types, you should pay attention to the data types used by both sides to prevent runtime errors. For example, in COM objects created with Visual Basic 6.0, values of type **Integer** assume the .NET equivalent type, **Short**. It is recommended that you use the Object Browser to examine the characteristics of imported members before using them.

Module level COM methods

Most COM objects are used by creating an instance of a COM class using the **New** keyword and then calling methods of the object. One exception to this rule involves COM objects that contain "AppObj" or "GlobalMultiUse" COM classes. Such classes are similar to module level methods in Visual Basic .NET classes. Visual Basic 6.0 implicitly creates instances of such objects for you the first time you call one of their methods. For example, in Visual Basic 6.0 you can add a reference to the **Microsoft DAO 3.6 Object Library** and call the `DBEngine` method without first creating an instance:

```
Dim db As DAO.Database
' Open the database
Set db = DBEngine.OpenDatabase("C:\nwind.mdb")
' Use the database object
```

Visual Basic .NET requires that you always create instances of COM objects before you can use their methods. To use these methods in Visual Basic .NET, declare a variable of the desired class and use the new keyword to assign the object to the object variable. The **Shared** keyword can be used when you want to make sure that only one instance of the class is created.

```
' Declarations level variable.
Shared DBEngine As New DAO.DBEngine()

   Sub DAOOpenRecordset()
      Dim db As DAO.Database
      Dim rst As DAO.Recordset
      Dim fld As DAO.Field
      ' Open the database
      db = DBEngine.OpenDatabase("C:\nwind.mdb")

      ' Open the Recordset
      rst = db.OpenRecordset _
        ("SELECT * FROM Customers WHERE Region = 'WA'", _
         DAO.RecordsetTypeEnum.dbOpenForwardOnly, _
         DAO.RecordsetOptionEnum.dbReadOnly)
      ' Print the values for the fields in the debug window.
      For Each fld In rst.Fields
         Debug.WriteLine(fld.Value & ";")
      Next
      Debug.WriteLine("")
      ' Close the recordset
      rst.Close()
   End Sub
```

Unhandled Errors in Event Handlers

One common interop problem involves errors in event handlers that handle events raised by COM objects. Such errors are ignored unless you specifically check for errors using **On Error** or **Try...Catch...Finally** statements. For example, the following example is from a Visual Basic .NET project that has a reference to the ADODB COM object.

```
' To use this example, add a reference to ADODB from the
' COM tab of the project references page.
Dim WithEvents cn As New ADODB.Connection()
Sub ADODBConnect()
   cn.ConnectionString = _
   "Provider=Microsoft.Jet.OLEDB.4.0;" & _
   "Data Source=C:\NWIND.MDB"
   cn.Open()
   MsgBox(cn.ConnectionString)
End Sub

Private Sub Form1_Load(ByVal sender As System.Object, _
                       ByVal e As System.EventArgs) Handles MyBase.Load
   ADODBConnect()
End Sub

Private Sub cn_ConnectComplete(ByVal pError As ADODB.Error, _
                               ByRef adStatus As ADODB.EventStatusEnum, _
                               ByVal pConnection As ADODB.Connection) _
                               Handles cn.ConnectComplete
'  This is the event handler for the cn_ConnectComplete event raised
'  by the ADODB.Connection object when a database is opened.
   Dim x As Integer = 6
   Dim y As Integer = 0
   Try
      x = x / y ' Attempt to divide by zero.
      ' This procedure would fail silently without exception handling.
   Catch ex As Exception
      MsgBox("There was an error: " & ex.Message)
   End Try
End Sub
```

This example raises an error as expected. However, if you try the same example without the **Try...Catch...Finally** block, the error is ignored as if you used the **OnError Resume Next** statement. Without error handling, the division by zero silently fails. Because such errors never raise unhandled exception errors, it is vital that you use some form of exception handling in event handlers that handle events from COM objects.

Understanding COM interop errors

Without error handling, interop calls often generate errors that provide little information. Whenever possible, use structured error handling to provide more information about problems when they occur. This can be particularly helpful when debugging applications For example:

```
Try
' Place call to COM object here.
Catch ex As Exception
' Display information about the failed call.
End Try
```

You can find out information such as the error description, HRESULT, and the source of COM errors by examining the contents of the exception object.

ActiveX Control Issues

Most ActiveX controls that work with Visual Basic 6.0 work with Visual Basic .NET without trouble. The main exceptions are container controls, or controls that visually contain other controls. Some examples of older controls that do not work correctly with Visual Studio .NET are:

- Microsoft Forms 2.0 Frame control
- Up-Down control, also known as the spin control

- Sheridan Tab Control

There are only a few workarounds for unsupported ActiveX control problems. You can migrate existing controls to Visual Studio .NET if you own the original source code. Otherwise, you can check with software vendors for updated .NET compatible versions of controls to replace unsupported ActiveX controls.

Passing ReadOnly Properties of Controls ByRef

Visual Basic .NET sometimes raises COM errors such as "**Error 0x800A017F CTL_E_SETNOTSUPPORTED**" when you to pass **ReadOnly** properties of some older ActiveX controls as **ByRef** parameters to other procedures. Similar procedure calls from Visual Basic 6.0 do not raise an error, and the parameters are treated as if you passed them by value. The error message you see in Visual Basic .NET is the COM object reporting that you are attempting to change a property that does not have a property **Set** procedure.

If you have access to the procedure being called, you can prevent this error by using the **ByVal** keyword to declare parameters that accept **ReadOnly** properties. For example:

```
Sub ProcessParams(ByVal c As Object)
   'Use the arguments here.
End Sub
```

If you do not have access to the source code for the procedure being called, you can force the property to be passed by value by adding an extra set of brackets around the calling procedure. For example:

```
Sub PassByVal ()
' The extra set of parens around the arguments
' forces them to be passed by value.
   ProcessParams((Me.AxListView1.ListItems))
End Sub
```

Deploying Assemblies That Expose Interop

Deploying assemblies that expose COM interfaces presents some unique challenges. For example, a potential problem arises when separate applications reference the same COM assembly. This situation is common when a new version of an assembly is installed and another application is still using the old version of the assembly. Uninstalling any assembly that shares a dll can unintentionally make it unavailable to the other assemblies.

To avoid this problem, you should install shared assemblies to the Global Assembly Cache (GAC) and use a MergeModule for the component. If you are unable to install the application in the GAC, then it should be installed to CommonFilesFolder in a version-specific subdirectory.

Assemblies that are not shared should be placed side by side in the directory with the calling application.

See Also

COM Interop | MarshalAsAttribute Class | Interop Marshaling | Data Type Changes in Visual Basic | Type Library Importer (Tlbimp.exe) | Type Library Exporter (Tlbexp.exe) | Walkthrough: Implementing Inheritance with COM Objects | Inherits Statement | About Merge Modules | Merge Module Projects | Global Assembly Cache

Multithreading in Visual Basic .NET

Visual Basic .NET applications can perform multiple tasks simultaneously using *multithreading* (or *free threading*), a process in which individual tasks execute on separate threads. Multithreading improves the performance and responsiveness of your applications.

In This Section

Multithreaded Applications
Describes how to create and use threads.
Thread States
Describes how to determine and change a thread's state.
Parameters and Return Values for Multithreaded Procedures
Describes how to pass and return parameters with multithreaded applications.
Thread Synchronization
Covers how to synchronize multithreaded applications.
Thread Pooling
Describes how tasks can be added to a queue and then started as new threads are created.
Thread Timers
Describes how to run procedures on separate threads at fixed intervals.
Multithreading with Forms and Controls
Covers special considerations when working with multithreading, forms, and controls.
Walkthrough: Multithreading
Shows how to create a simple multithreaded application.

Related Sections

Multithreading in Components
Provides links to topics that describe how to use multithreading in component programming.
Walkthrough: Authoring a Simple Multithreaded Component with Visual Basic
Shows how to create multithreaded components.

Multithreaded Applications

With Visual Basic .NET, you can write applications that perform multiple tasks simultaneously. Tasks with the potential of holding up other tasks can execute on separate threads, a process known as *multithreading* or *free threading*. Applications that use multithreading are more responsive to user input because the user interface stays active while processor-intensive tasks execute on separate threads. Multithreading is also useful when creating scalable applications, because you can add threads as the workload increases.

Creating and Using Threads

You create a new thread in Visual Basic .NET by declaring a variable of type **System.Threading.Thread** and calling the constructor with the **AddressOf** statement and the name of the procedure or method you want to execute on the new thread. The following code provides an example:

```
Dim MyThread As New System.Threading.Thread(AddressOf MySub)
```

Starting and Stopping Threads

To start the execution of a new thread, use the **Start** method, as in the following code:

```
MyThread.Start()
```

To stop the execution of a thread, use the **Abort** method, as in the following code:

```
MyThread.Abort()
```

Besides starting and stopping threads, you can also pause threads by calling the **Sleep** or **Suspend** methods, resume a suspended thread with the **Resume** method, and destroy a thread using the **Abort** method, as in the following code:

```
MyThread.Sleep ()
MyThread.Suspend ()
MyThread.Abort ()
```

The following table shows some of the methods you can use to control individual threads.

Method	Action
Start	Causes a thread to start running.
Sleep	Pauses a thread for a specified time.
Suspend	Pauses a thread when it reaches a safe point.
Abort	Stops a thread when it reaches a safe point.
Resume	Restarts a suspended thread
Join	Causes the current thread to wait for another thread to finish. If used with a timeout value, this method ret urns **True** if the thread finishes in the allotted time.

Most of these methods are self explanatory, but the concept of *safe points* may be new to you. Safe points are places in code where it is safe for the common language runtime to perform automatic *garbage collection*, the process of releasing unused variables and reclaiming memory. When you call the **Abort** or **Suspend** methods of a thread, the common language runtime analyzes the code and determines the location of an appropriate place for the thread to stop running.

Threads also contain a number of useful properties, as shown in the following table:

Property	Value
thread.IsAlive	Contains the value **True** if a thread is active.
IsBackground	Gets or sets a Boolean that indicates if a thread is or should be a background thread. Background t hreads are like foreground threads, but background threads do not prevent a process from termin ating. Once all foreground threads belonging to a process have terminated, the common languag e runtime ends the process by calling the **Abort** method on background threads that are still alive.
Name	Gets or sets the name of a thread. Most commonly used to discover individual threads when debu gging.

Priority	Gets or sets a value used by the operating system to prioritize thread scheduling.
ApartmentState	Gets or sets the threading model used for a particular thread. Threading models are important wh en a thread calls unmanaged code.
ThreadState	Contains a value that describes a thread's state or states.

See Thread States for more information on thread states and methods.

Thread Priorities

Every thread has a priority property — that determines how large of a time slice it gets to execute. The operating system allocates longer time slices to high-priority threads than it does to low-priority threads. New threads are created with the value of **Normal**, but you can adjust the **Priority** property to any of the other values in the **System.Threading.ThreadPriority** enumeration.

See ThreadPriority Enumeration for a detailed description of the various thread priorities.

Foreground and Background Threads

A *foreground thread* runs indefinitely, while a *background thread* terminates once the last foreground thread has stopped. You can use the **IsBackground** property to determine or change the background status of a thread.

See Also

Thread States | Parameters and Return Values for Multithreaded Procedures | Thread Synchronization | Multithreading with Forms and Controls | Multithreading in Components

Thread States

The **ThreadState** property provides information about the status of a thread. Because threads can be in more than one state at any given time, the value stored in **ThreadState** can be a combination of the values in the **System.Threading.Threadstate** enumeration. For example, if a thread is blocked on a call to **System.Threading.Monitor.Wait**, and another thread calls the **Abort** method on that same thread, the thread is in both the **Waiting** and the **AbortRequested** state at the same time.

Changing Thread States

Once a thread has started, you can call its methods to change its state. For example, you can cause a thread to pause for a fixed number of milliseconds by calling **Thread.Sleep**. The **Sleep** method takes as a parameter a timeout, which is the number of milliseconds that the thread remains blocked. Calling **Thread.Sleep(System.Threading.Timeout.Infinite)** causes a thread to sleep until it is interrupted by another thread that calls **Thread.Interrupt**. The **Thread.Interrupt** method wakes the destination thread out of any wait it may be in and causes an exception to be raised.

You can also pause a thread by calling **Thread.Suspend**. When a thread calls **Thread.Suspend** on itself, the call blocks until another thread resumes it by calling **Thread.Resume**. When a thread calls **Thread.Suspend** on another thread, the call is non-blocking and causes the other thread to pause. Calling **Thread.Resume** breaks another thread out of its suspended state and causes it to resume execution. Unlike **Thread.Sleep**, **Thread.Suspend** does not immediately stop a thread; the suspended thread does not pause until the common language runtime determines that it has reached a safe point.

The **Thread.Abort** method stops a running thread by raising a **ThreadAbortException** exception that causes the thread to die.

See Thread Methods for detailed information about these methods.

See Also

Multithreaded Applications | Thread Synchronization | Parameters and Return Values for Multithreaded Procedures | Multithreading with Forms and Controls | System.Threading Namespace | SyncLock Statement | Delegates and the AddressOf Operator | Multithreading in Components

Parameters and Return Values for Multithreaded Procedures

Supplying and returning values in a multithreaded applications is complicated because the constructor for the thread class must be passed a reference to a procedure that takes no arguments and returns no value. The following sections show some simple ways to supply parameters and return values from procedures on separate threads.

Supplying Parameters for Multithreaded Procedures

The best way to supply parameters for a multithreaded method call is to wrap the target method in a class and define fields for that class that will serve as parameters for the new thread. The advantage of this approach is that you can create a new instance of the class, with its own parameters, every time you want to start a new thread. For example, suppose you have a function that calculates the area of a triangle, as in the following code:

```
Function CalcArea(Base As Double, Height As Double) As Double
   CalcArea= 0.5 * Base * Height
End Sub
```

You can write a class that wraps the `CalcArea` function and creates fields to store input parameters, as follows:

```
Class AreaClass
    Public Base As Double
    Public Height As Double
    Public Area As Double
    Sub CalcArea()
        Area = 0.5 * Base * Height
        MsgBox("The area is: " & Area)
    End Sub
End Class
```

To use the `AreaClass`, you can create an `AreaClass` object, and set the `Base` and `Height` properties as shown in the following code:

```
Protected Sub TestArea()
   Dim AreaObject As New AreaClass()
   Dim Thread As New System.Threading.Thread _
                   (AddressOf AreaObject.CalcArea)
   AreaObject.Base = 30
   AreaObject.Height = 40
   Thread.Start()
End Sub
```

Notice that the `TestArea` procedure does not check the value of the `Area` field after calling the `CalcArea` method. Because `CalcArea` runs on a separate thread, the `Area` field is not guaranteed to be set if you check it immediately after calling `Thread.Start`. The next section discusses a better way to return values from multithreaded procedures.

Returning Values From Multithreaded Procedures

Returning values from procedures that run on separate threads is complicated by the fact that the procedures cannot be functions and cannot use **ByRef** arguments. The best way to return values is to have your multithreaded procedure call a procedure in your application, and then pass the results as arguments. To do this, raise an event when a task is done, and process the results with an event handler.

The following example returns a value by raising an event from a procedure running on a separate thread:

```
Class AreaClass2
    Public Base As Double
    Public Height As Double
    Private Area As Double
    Public Event ThreadDone(ByVal Area As Double)
    Sub CalcArea()
        Area = 0.5 * Base * Height  ' Calculate the area of a triangle.
        RaiseEvent ThreadDone(Area) ' Raise an event to return the area.
    End Sub
```

```
End Class

Dim WithEvents AreaObject2 As AreaClass2
Protected Sub TestArea2()
   AreaObject2 = New AreaClass2()
   Dim Thread As New System.Threading.Thread _
                           (AddressOf AreaObject2.CalcArea)
   AreaObject2.Base = 30
   AreaObject2.Height = 40
   Thread.Start()
End Sub

Sub AreaDoneEventHandler(ByVal Area As Double) _
                         Handles AreaObject2.ThreadDone
   MsgBox("The area is: " & Area)
End Sub
```

You can provide parameters and return values to thread pool threads by using the optional **ByVal** state object variable of the **QueueUserWorkItem** method. Thread timer threads also support a state object for this purpose. For information on the thread pooling and thread timers, see Thread Pooling and Thread Timers.

See Also

Thread States | Thread Pooling | Thread Synchronization | Events and Event Handlers | Multithreaded Applications | Multithreading with Forms and Controls| Delegates and the AddressOf Operator | Multithreading in Components

Thread Synchronization

As you write a threaded application, you may need to synchronize individual threads with other parts of your program. Synchronization provides a compromise between the unstructured nature of multithreaded programming and the structured order of synchronous processing.

You use synchronization techniques:

- To explicitly control the order in which code runs whenever tasks must be performed in a specific sequence

  -or-

- To prevent the problems that can occur when two threads share the same resource at the same time.

For example, you could use synchronization to cause a display procedure to wait until a data retrieval procedure that is running on another thread is complete.

Using events is a common way to synchronize threaded applications. In this scenario, each thread raises an event to signal its status to the main program or other threads. Other parts of the application, outside the thread that raised the event, handle the event and process the information.

> **Note** When an event is raised from one object to another object, the event handler will execute on the same thread that the **RaiseEvent** occurred. Therefore, if the event handler accesses shared data it must itself use synchronization if that data may be simultaneously accessed on a different thread

Although using events is a simple way to synchronize threads, there are limitations to this approach. Visual Basic .NET includes other features for controlling individual threads and their access to specific sections of code. The following sections discuss synchronization techniques.

Synchronization Techniques

There are two approaches to synchronization, polling and using synchronization objects. Polling repeatedly checks the status of an asynchronous call from within a loop. Polling is the least efficient way to manage threads because it wastes resources by repeatedly checking the status of the various thread properties.

For example, the **IsAlive** property can be used when polling to see if a thread has exited. Use this property with caution because a thread that is alive is not necessarily running. You can use the thread's **ThreadState** property to get more detailed information about a thread's status. Because threads can be in more than one state at any given time, the value stored in **ThreadState** can be a combination of the values in the **System.Threading.Threadstate** enumeration. Consequently, you should carefully check all relevant thread states when polling. For example, if a thread's state indicates that it is not **Running,** it may be done. On the other hand, it may be suspended or sleeping.

Waiting for a Thread to Finish

The **Thread.Join** method is useful for determining if a thread has completed before starting another task. The **Join** method waits a specified amount of time for a thread to end. If the thread ends before the timeout, **Join** returns **True;** otherwise it returns **False**. For information on **Join**, see Thread.Join Method

As you can imagine, polling sacrifices many of the advantages of multithreading in return for control over the order that threads run. Because it is so inefficient, polling generally not recommended. A more efficient approach would use the **Join** method to control threads. **Join** causes a calling procedure to wait either until a thread is done or until the call times out if a timeout is specified. The name, join, is based on the idea that creating a new thread is a fork in the execution path. You use **Join** to merge separate execution paths into a single thread again.

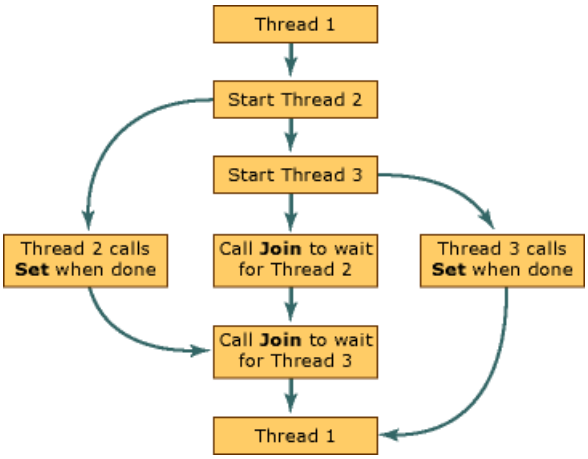


Figure 1 Threading

One point should be clear: **Join** is a synchronous or *blocking* call. Once you call Join or a wait method of a wait handle, the calling procedure stops and waits for the thread to signal that it is done.

```
Sub JoinThreads()
   Dim Thread1 As New System.Threading.Thread(AddressOf SomeTask)
   Thread1.Start()
   Thread1.Join()      ' Wait for the thread to finish.
   MsgBox("Thread is done")
End Sub
```

These simple ways of controlling threads, which are useful when you are managing a small number of threads, are difficult to use with large projects. The next section discusses some advanced techniques you can use to synchronize threads.

Advanced Synchronization Techniques

Multithreaded applications often use wait handles and monitor objects to synchronize multiple threads. These sections explain how to use the following .NET Framework classes when synchronizing threads: AutoResetEvent Class, Interlocked Class, ManualResetEvent Class, Monitor Class, Mutex Class, ReaderWriterLock Class, Timer Class, WaitHandle Class

Wait Handles

Wait handles are objects that signal the status of one thread to another thread. Threads can use wait handles to notify other threads that they need exclusive access to a resource. Other threads must then wait to use this resource until the wait handle is no longer in use. Wait handles have two states, *signaled* and *nonsignaled*. A wait handle that is not owned by any thread is in the *signaled* state. A wait handle that is owned by a thread is in the *nonsignaled* state.

Threads request ownership of a wait handle by calling one of the wait methods, such as **WaitOne**, **WaitAny**, or **WaitAll**. The wait methods are blocking calls that are similar to the **Join** method of an individual thread.

- If no other thread owns the wait handle, the call immediately returns **True**, the wait handle's status is changed to nonsignaled, and the thread that owns the wait handle continues to run.
- If a thread calls one of a wait handle's wait methods, but the wait handle is owned by another thread, the calling thread will either wait for a specified time (if a timeout is specified) or wait indefinitely (if no timeout is specified) for the other thread to release the wait handle. If a timeout is specified and the wait handle is released before the timeout expires, the call returns **True**. Otherwise, the call to wait returns **False**, and the calling thread continues to run.

Threads that own a wait handle call the **Set** method when they are done or when they no longer need the wait handle. Other threads can reset the status of a wait handle to nonsignaled by either calling the **Reset** method or by calling **WaitOne**, **WaitAll**, or **WaitAny** and successfully waiting for a thread to call **Set**. **AutoResetEvent** handles are automatically reset to nonsignaled by the system after a single waiting thread has been released. If no threads are waiting, the event object's state remains signaled.

There are three kinds of wait handles commonly used with Visual Basic .NET: mutex objects, **ManualResetEvent**, **AutoResetEvent**. The last two are often referred to as synchronization events.

Mutex Objects

Mutex objects are synchronization objects that can only be owned by a single thread at a time. In fact, the name, mutex, is derived from the fact that ownership of mutex objects is mutually exclusive. Threads request ownership of the mutex object when they require exclusive access to a resource. Because only one thread can own a mutex object at any time, other threads must wait for ownership of a mutex object before using the resource.

The **WaitOne** method causes a calling thread to wait for ownership of a mutex object. If a thread terminates normally while owning a mutex object, the state of the mutex object is set to signaled and the next waiting thread gets ownership.

Synchronization Events

Synchronization events are used to notify other threads that something has occurred or that a resource is available. Do not be fooled by the fact that these items use the word *event*. Synchronization events, which are unlike other Visual Basic events, are really wait handles. Like other wait handles, synchronization events have two states, signaled and nonsignaled. Threads that call one of the wait methods of a synchronization event must wait until another thread signals the event by calling the **Set** method. There are two synchronization event classes. Threads set the status of **ManualResetEvent** instances to signaled using the **Set** method. Threads set the status of **ManualResetEvent** instances to nonsignaled using the **Reset** method or when control returns to a waiting **WaitOne** call. Instances of the **AutoResetEvent** class can also be set to signaled using **Set**, but they automatically return to nonsignaled as soon as a waiting thread is notified that the event became signaled.

Monitor Objects and SyncLock

Monitor objects are used to ensure that a block of code runs without being interrupted by code running on other threads. In other words, code in other threads cannot run until code in the synchronized code block has finished.

Suppose, for example, that you have a program that repeatedly and asynchronously reads data and displays the results. With operating systems that use preemptive multitasking, a running thread can be interrupted by the operating system to allow time for some other thread to run. Without synchronization, it is possible that you could get a partially updated view of the data if the object that represents the data is modified by another thread while the data is being displayed. Monitor objects guarantee that a section of code will run without being interrupted. Visual Basic .NET provides the **SyncLock** and **End SyncLock** statements to simplify access to monitor objects. Visual C#® .NET uses the **Lock** keyword in the same way.

Note Access to an object is locked out only if the accessing code is contained within a **SyncLock** block on the same object instance.

For information on the **SyncLock** statement, see SyncLock Statement

Interlocked Class

You can use the methods of the **Interlocked** class to prevent problems that can occur when multiple threads attempt to simultaneously update or compare the same value. The methods of this class let you safely increment, decrement, exchange, and compare values from any thread. The following example shows how to use the **Increment** method to increment a variable that is shared by procedures running on separate threads.

```
Sub ThreadA(ByRef IntA As Integer)
   System.Threading.Interlocked.Increment(IntA)
End Sub

Sub ThreadB(ByRef IntA As Integer)
   System.Threading.Interlocked.Increment(IntA)
End Sub
```

For information on the Interlocked Class, see Interlocked Class

ReaderWriter Locks

In some cases, you may want to lock a resource only when data is being written and permit multiple clients to simultaneously read data when data is not being updated. The **ReaderWriterLock** class enforces exclusive access to a resource while a thread is modifying the resource, but it allows non-exclusive access when reading the resource. ReaderWriter locks are a useful alternative to exclusive locks that cause other threads to wait, even when those threads do not need to update data.

For information on the ReaderWriterLock Class, see ReaderWriterLock Class

Deadlocks

Thread synchronization is invaluable in multithreaded applications, but there is always the danger of *deadlocks* where multiple threads wait for each other. Analogous to when cars are stopped at a four-way stop and each person is waiting for another to go, deadlocks bring everything to a halt. Needless to say, avoiding deadlocks is important. There are many ways to get into deadlock situations but careful planning is key to avoiding them. You can often predict deadlock situations by diagramming multithreaded applications before you start coding.

See Also

System.Threading Namespace | Thread States | Multithreaded Applications |
Parameters and Return Values for Multithreaded Procedures | Multithreading with Forms and Controls | SyncLock Statement |
Delegates and the AddressOf Operator | Multithreading in Components | WaitOne Method | WaitAll Method | WaitAny Method

Thread Pooling

Thread pooling is a form of multithreading where tasks are added to a queue and automatically started as threads are created. With thread pooling, you call the **Threadpool**.**QueueUserWorkItem** method with a delegate for the procedure you want to run, and Visual Basic .NET creates the thread and runs your procedure. The following example shows how you could use thread pooling to start several tasks.

```
Sub DoWork()
   Dim TPool As System.Threading.ThreadPool
   ' Queue a task
   TPool.QueueUserWorkItem(New System.Threading.WaitCallback _
                        (AddressOf SomeLongTask))
   ' Queue another task
   TPool.QueueUserWorkItem(New System.Threading.WaitCallback _
                        (AddressOf AnotherLongTask))
End Sub
```

Thread pooling is useful when you want to start many separate tasks without individually setting the properties of each thread. Each thread starts with a default stack size and priority. By default, up to 25 thread pool threads can run per system processor. Additional threads in excess of the limit can be queued, but they do not start until other threads finish.

One advantage of thread pooling is that you can pass arguments in a state object to the task procedure. If the procedure you are calling requires more than one argument, you can cast a structure or an instance of a class into an **Object** data type.

Thread Pool Parameters and Return Values

Returning values from a thread pool thread is not straightforward. The standard way of returning values from a function call is not allowed because Sub procedures are the only type of procedure that can be queued to a thread pool. One way you can provide parameters and return values is by wrapping the parameters, return values, and methods in a wrapper class as described in Parameters and Return Values for Multithreaded Procedures. An easer way to provide parameters and return values is by using the optional **ByVal** state object variable of the **QueueUserWorkItem** method. If you use this variable to pass a reference to an instance of a class, the members of the instance can be modified by the thread pool thread and used as return values. At first it may not be obvious that you can modify an object referred to by a variable that is passed by value. This is possible because only the object reference is passed by value. When you make changes to members of the object referred to by the object reference, the changes apply to the actual class instance.

Structures cannot be used to return values inside state objects. Because structures are value types, changes that the asynchronous process makes do not change the members of the original structure. Use structures to provide parameters when return values are not needed.

See Also

Multithreaded Applications | ThreadPool.QueueUserWorkItem Method | Thread Timers | Thread Synchronization | Parameters and Return Values for Multithreaded Procedures | Multithreading with Forms and Controls | System.Threading Namespace | SyncLock Statement

Thread Timers

The **Threading.Timer** class is useful to periodically run a task on a separate thread. For example, you could use a thread timer to check the status and integrity of a database or to back up critical files. The following example starts a task every two seconds and uses a flag to initiate the **Dispose** method that stops the timer. This example posts status to the output window, so you should make this window visible by pressing Control+Alt+O before you test the code.

```
Class StateObjClass
' Used to hold parameters for calls to TimerTask
      Public SomeValue As Integer
      Public TimerReference As System.Threading.Timer
      Public TimerCanceled As Boolean
End Class

Sub RunTimer()
   Dim StateObj As New StateObjClass()
   StateObj.TimerCanceled = False
   StateObj.SomeValue = 1
   Dim TimerDelegate As New Threading.TimerCallback(AddressOf TimerTask)
   ' Create a timer that calls a procedure every 2 seconds.
   ' Note: There is no Start method; the timer starts running as soon as
   ' the instance is created.
   Dim TimerItem As New System.Threading.Timer(TimerDelegate, StateObj, _
                                               2000, 2000)
   StateObj.TimerReference = TimerItem  ' Save a reference for Dispose.

   While StateObj.SomeValue < 10 ' Run for ten loops.
      System.Threading.Thread.Sleep(1000)  ' Wait one second.
   End While

   StateObj.TimerCanceled = True  ' Request Dispose of the timer object.
End Sub

Sub TimerTask(ByVal StateObj As Object)
   Dim State As StateObjClass = CType(StateObj, StateObjClass)
   Dim x As Integer
   ' Use the interlocked class to increment the counter variable.
   System.Threading.Interlocked.Increment(State.SomeValue)
   Debug.WriteLine("Launched new thread  " & Now)
   If State.TimerCanceled Then    ' Dispose Requested.
      State.TimerReference.Dispose()
      Debug.WriteLine("Done  " & Now)
   End If
End Sub
```

Thread timers are particularly useful when the System.Windows.Forms.Timer Class is unavailable, such as when you are developing console applications.

See Also

Multithreaded Applications | Thread Pooling | Thread Synchronization |
Parameters and Return Values for Multithreaded Procedures | Multithreading with Forms and Controls |
System.Threading Namespace | SyncLock Statement

Multithreading with Forms and Controls

While multithreading is best suited to running procedures and class methods, you can also use it with forms and controls. If you do so, be aware of the following points:

- Whenever possible, execute the methods of a control only on the thread with which it was created. If you must call a method of a control from another thread, you must use the `Control.Invoke` style of calling the method.
- Do not use the **SyncLock** statement to lock threads that manipulate controls or forms. Because the methods of controls and forms sometimes call back to a calling procedure, you can end up inadvertently creating a deadlock — a situation in which two threads wait for each other to release the lock, causing the application to halt.

See Also

Control.Invoke Method | Control.InvokeRequired Property | Thread States |
Parameters and Return Values for Multithreaded Procedures| Multithreaded Applications | Multithreading in Components

Walkthrough: Multithreading

This walkthrough demonstrates how to create a multithreaded application that searches a text file for occurrences of a word. The walkthrough consists of four main parts:

- Defining a class with a method that can be run on a thread independently
- Handling events raised by other threads
- Starting a new thread to run a method
- Implementing a `Cancel` button that stops the thread

To build the code example for this topic

1. Open a new Visual Basic .NET Windows Application project, and create a form named `Form1`.
2. Add two buttons and four text boxes to `Form1`.
3. Name the objects as shown in the following table.

Object	Property	Setting
First button	**Name**, **Text**	Start, Start
Second button	**Name**, **Text**	Cancel, Cancel
First text box	**Name**, **Text**	SourceFile, ""
Second text box	**Name**, **Text**	CompareString, ""
Third text box	**Name**, **Text**	WordsCounted, "0"
Fourth text box	**Name**, **Text**	LinesCounted, "0"

4. Add a label next to each text box. Set the **Text** property for each label as shown in the following table.

Object	Property	Setting
First label	**Text**	Source File
Second label	**Text**	Compare String
Third label	**Text**	Matching Words
Fourth label	**Text**	Lines Counted

To define the method that will run on a separate thread

1. From the **Project** menu, choose **Add Class** to add a class to the project. The **Add New Item** dialog is displayed.
2. Select **Class** from the templates window and type `Words.vb` in the name field.
3. Click **Open**. The `Words` class is displayed.
4. Add an Option Compare statement to the top of the `Words` class module, above the **Class** statement:

```
Option Compare Text ' Case insensitive search.
' Use Option Compare Binary for case sensitive search.
```

5. Add the following code to the `Words` class:

```
Public Class Words
   Private strloc As Integer
   Private RemainingString As String
   Private SourceString As String
   Public SourceFile As String
   Public CompareString As String
   Public WordCount As Integer
   Private LinesCounted As Integer = 0
   Public Event Status(ByVal LinesCounted As Integer, _
                       ByVal WordsMatched As Integer)
   Public Event FinishedCounting(ByVal NumberOfMatches As Integer)

   Sub CountWords()
      Dim f As System.IO.File
```

```
        Dim mystream As System.IO.StreamReader
        Dim mystr As String = " " ' Initialize to not empty.

        If SourceFile = System.String.Empty _
           Or CompareString = System.String.Empty Then
           MsgBox("You must set the SourceFile " & _
                  " and CompareString fields before calling CountWords.")
           Exit Sub
        End If

        If f.Exists(SourceFile) = False Then
           MsgBox("Error; Unable to open the input file.")
           Exit Sub
        End If

        Try
           mystream = f.OpenText(SourceFile) ' Open a new stream.
           ' Do until the stream returns Nothing at end of file.
           Do Until IsNothing(mystr)
              mystr = mystream.ReadLine
              WordCount += CountInString(1, mystr, CompareString)
              LinesCounted += 1 ' Increment line count.
              ' Raise an event so the form can monitor progress.
              RaiseEvent Status(LinesCounted, WordCount)
           Loop
        Catch eof As IO.EndOfStreamException
        ' No action is necessary, the end of the stream has been reached.
        Catch IOExcep As IO.IOException
           ' Some kind of error occurred.
           MsgBox(IOExcep.Message)
        Finally
           mystream.Close() ' Close the file.
        End Try
        RaiseEvent FinishedCounting(WordCount)
     End Sub

     Private Function CountInString(ByVal StartingPoint As Integer, _
                                    ByVal SourceString As String, _
                                    ByVal CompareString As String) As Integer
        ' This function counts the number of times
        ' a word is found in a sentence.
        ' If the word is found, the function is called recursively on
        ' the remainder of the string.
        strloc = Strings.InStr(StartingPoint, SourceString, CompareString)
        If strloc <> 0 Then
           CountInString += 1
           CountInString += CountInString(strloc + Strings.Len(CompareString), _
                                          SourceString, CompareString)
        End If
     End Function
  End Class
```

To handle events from the thread

- Add the following event handlers to your main form:

```
  Sub FinishedCountingEventHandler(ByVal WordCount As Integer)
     ' This event handler is called when the word-counting thread
     ' raises a FinishedCounting event indicating that
```

```
    ' the whole file has been checked. The WordCount parameter
    ' is the number of matches found.
    Me.WordsCounted.Text = CStr(WordCount)
    MsgBox("Finished counting words.")
End Sub
Sub LineCountEventHandler(ByVal LinesCounted As Integer, _
                          ByVal WordsMatched As Integer)
    ' This event handler is called when the word-counting thread
    ' raises a status event after reading each line
    ' from the source file.
    Me.LinesCounted.Text = CStr(LinesCounted)
    Me.WordsCounted.Text = CStr(WordsMatched)
End Sub
```

To start and call a new thread that runs the WordCount method

1. Add the following code to the **Declarations** section of `Form1`:

```
Dim Thread As System.Threading.Thread
```

2. Add the following procedure to your program:

```
Sub StartThread()
    Dim WC As New words()
    Me.WordsCounted.Text = "0"
    ' Set the values of some object fields, because
    ' you cannot create a thread and pass it a method
    ' signature that contains parameters.
    WC.CompareString = Me.CompareString.Text ' The string to look for.
    WC.SourceFile = Me.SourceFile.Text       ' The file to search.
    ' Associate an event handler with the event that
    ' is raised when the new thread is done.
    AddHandler WC.FinishedCounting, AddressOf _
                                    FinishedCountingEventHandler
    ' Associate an event handler with the event that
       ' is raised after each line is read.
    AddHandler WC.Status, AddressOf LineCountEventHandler
    ' Create the new thread.
    Thread = New System.Threading.Thread(AddressOf WC.CountWords)
    ' Start the new thread.
    Thread.Start()
End Sub
```

3. Call the `StartThread` method from the `Start` button on your form:

```
Private Sub Start_Click(ByVal sender As System.Object, _
                        ByVal e As System.EventArgs) Handles Start.Click
    StartThread()
End Sub
```

To implement a Cancel button that stops the thread

1. Add the following procedure to your main form:

```
Protected Sub StopThread(Thread as System.Threading.Thread)
'   Call StopThread from a Cancel button procedure.
    Thread.Abort()
End Sub
```

2. Call the `StopThread` procedure from the `click` event handler for the `Cancel` button.

```
Private Sub Cancel_Click(ByVal sender As System.Object, _
                         ByVal e As System.EventArgs) _
                         Handles Cancel.Click
   StopThread(Thread)
End Sub
```

Testing

You can now test the application to make sure it works correctly.

To test the application

1. Press **F5** to run the application.
2. When the form is displayed, enter the file path in the `sourceFile` textbox for a file you want to test. For example, assuming your test file is named Test.txt, enter C:\Test.txt.
3. In the second text box, enter a word or phrase for the application to search for in the text file.
4. Click the `Start` button. The `LinesCounted` button should begin incrementing immediately. The application displays the message "Finished Counting" when it is done.

To test the Cancel button

1. Press **F5** to start the application, and enter the file name and search word as described in the previous procedure. Make sure that the file you choose is large enough so you will have time to cancel the procedure before it is finished.
2. Click the `Start` button to start the application.
3. Click the `Cancel` button. The application should stop counting immediately.

Error Handling

This application contains some basic error handling. It detects and handles source file problems and blank search strings. You can make this program more robust by handling other errors, such as exceeding the maximum number of words or lines that can be counted.

See Also

Multithreading in Visual Basic .NET | Walkthrough: Authoring a Simple Multithreaded Component with Visual Basic

Processing Drives, Folders, and Files

Visual Basic allows you to process drives, folders, and files in several different ways: through the use of the .NET **System.IO** model, through traditional methods such as the **FileOpen** and **Write** functions, and through the File System Object (FSO) model. The following sections discuss these methods in detail.

In This Section

File Access with Visual Basic .NET
Discusses how to use the .NET **System.IO** model to work with files, drives, and folders.

File Access with Visual Basic Run-Time Functions
Discusses how to process files directly to create, manipulate, and store text and other data.

Accessing Files with FileSystemObject
Covers using **FileSystemObject** to create and manipulate files.

Related Sections

Program Structure and Code Conventions
Provides guidelines for the physical structure and appearance of programs.

Choosing Among File I/O Options in Visual Basic .NET

Visual Basic .NET presents three possible approaches to file I/O for the programmer: runtime methods contained in Visual Basic .NET; FileSystemObject; and the Common Language Runtime File methods.

File Methods in Visual Basic .NET

The **Microsoft.VisualBasic** namespace is populated by a host of functions and methods familiar to users of earlier versions of Visual Basic. While some names have shifted slightly, the vast majority are the same: **Dir**, **Input**, **Print**, **Seek**, **Write**, and so forth.

A few changes have occurred in the functions used in these different types of file access. The most noticeable change is the substitution of the **FilePut/FilePutObject** and **FileGet/FileGetObject** functions for **Put** and **Get**. In Visual Basic .NET, **FilePut** and **FileGet** map to the **Put** and **Get** functions; **FilePutObject** and **FileGetObject** are used to write an object of type **Object** (the equivalent of **Variant** type in earlier versions of Visual Basic) to the file. For more information, see Variant Type Constant Changes in Visual Basic .NET.

Advantages and Disadvantages

Two principal advantages of the functions provided by the Visual Basic .NET runtime are familiarity and ease of use. The core functionality of Visual Basic .NET, which remains familiar, intuitive, and flexible, provides a comfortable launching place for exploration of .NET.

Similarly, upgrading a program written in an earlier version of Visual Basic to its .NET counterpart or creating an application that interacts well with applications written in earlier versions should prove relatively simple. For more information on upgrading previous versions of code, see Preparing a Visual Basic 6.0 Application for Upgrading.

The Visual Basic .NET functions also provide binary file access, which the **FileSystemObject** object (described in the next section) does not, making it a better choice when working with variable length fields.

Finally, there are no special considerations, such as the need to import namespaces, when using the Visual Basic .NET built-in file I/O functions from within Visual Basic .NET.

The fact that Visual Basic .NET file I/O functions do not support writing types other than **String**, **Date**, **Integer**, **Long**, **Single**, **Double**, **Decimal**, and structures and arrays of those types poses a disadvantage when using these functions. Beyond this, classes cannot be serialized and performance in such cases may not be as optimal as using the **System.IO** classes directly.

Code Examples

The following example checks to see if a file exists and if so, uses the **FileCopy** function to copy it to a new file.

```
Private Sub CopyFiles()
   Dim checkFile As String
   checkFile = Dir$("c:\test.txt")
   If checkFile = "test.txt"
      FileCopy "c:\test.txt", "c:\new.txt"
   End If
End Sub
```

The following example uses the **FilePut** function, which corresponds to the **Put** function in previous versions of Visual Basic, to write data to a file. Three records of the structure `CustomerRecord` are written to the file.

```
Structure CustomerRecord
   Public OrderNumber As Integer
   Public Name As String
   Public OrderDate As Date
End Structure

Sub WriteData()
   Dim MyRecord As CustomerRecord
   Dim RecordNumber As Integer
   Dim RecordDate As Date
   ' Open file for random access.
   FileOpen(1, "C:\TESTFILE.txt", OpenMode.Binary)
   For RecordNumber = 1 To 3   ' Loop 3 times.
      MyRecord.OrderNumber = RecordNumber    ' Define OrderNumber.
```

```
      MyRecord.OrderDate = RecordDate          ' Define OrderDate.
      MyRecord.Name = "My Name" & RecordNumber ' Create a string.
      FilePut(1, MyRecord)                 ' Write record to file.
   Next RecordNumber
   FileClose(1)
End Sub
```

.NET Framework Classes

The **System.IO** namespace contains **File** and **Directory** classes, which provide the basic functionality that manipulates files and directories. Because all the methods are static or shared members of these objects, you can use them directly without creating an instance of the class first.

Advantages and Disadvantages

Venturing into the common language runtime yields a rich class library. For example, the **System.IO** namespace provides services beyond basic file I/O, such as the **FileSystemWatcher** class, which allows you to monitor changes to files, and the **FileInfo** and **DirectoryInfo** classes, which allow you to gather information about a specified file or directory. For more information, see FileSystemWatcher Class, FileInfo Class, and DirectoryInfo Class.

Similarly, the common language runtime offers the **FileIOPermission** class, which allows you to control access to files and directories. This may be particularly important for developers working with Web Forms, which by default run within the context of a special local user named ASPNET, which is created as part of the ASP.NET and .NET Framework installations. When such an application requests access to a resource, that request is made within the context of that user. However, by default, the ASPNET user has limited permissions, which may prevent them from performing actions such as writing to a file from a Web application. For more information, see FileIOPermission Class.

The common language runtime is also compatible with other .NET languages, such as Visual C#™, and may aid in achieving consistency across applications, which may be particularly important at the enterprise level.

Lack of familiarity with the common language runtime may create difficulties for a Visual Basic programmer who is still becoming familiar with these new approaches. If you are in that category, you may prefer to rely on the methods you know well.

Code Examples

The following example uses the **StreamReader** class to read the contents of a text file.

```
Imports System
Imports System.IO
Function ReadTextFile(ByVal path As String) As String
   Dim sr As System.IO.StreamReader
   Dim Contents As String
   sr = New StreamReader(path)
   Contents = sr.ReadToEnd()
   sr.Close()
   Return Contents
End Function
```

The following code reverses the previous example, using the same approach with the **StreamWriter** class to write to a text file.

```
Imports System
Imports System.IO
Sub WriteTextFile(ByVal path As String, ByVal value As String)
Dim sr As StreamWriter
Dim Contents As String
   sr = New StreamWriter(path)
   sr.Write(value)
   sr.Close()
End Sub
```

FileSystemObject

Like the functions and methods provided by the **Microsoft.VisualBasic** namespace, the **FileSystemObject** object will be familiar to users of previous versions of Visual Basic.

The FileSystemObject (FSO) model presents files, directories, and drives as COM objects, each with their own properties and methods. You can create and manipulate these objects and use their properties to discover information such as a directory's contents, a file's size, the time an object was created, and so on. You access the objects representing the system's files, directories, and drives by creating and accessing an instance of the **FileSystemObject** object.

The FSO object model, which is in the Microsoft Scripting Runtime type library (Scrrun.dll), supports the creation and manipulation of text files through the **TextStream** object. For more information, see TextStream Object. The **FileStream** class exposes a stream around a file, allowing both synchronous and asynchronous read and write operations on a file system as well as buffering input and output for better performance. Information is written to the buffer and the contents of the buffer are written to the file only when the buffer is full or a **Close** method is called. For more information, see FileStream Class.

Advantages and Disadvantages

The primary advantage of the FSO is that it gathers a number of file I/O functions into a single object. By creating an instance of the object, you can access its methods and properties.

The FSO object model is also well suited to threading. It provides a constructor for asynchronous I/O, such as when using the **BeginRead** and **BeginWrite** methods, which enables the main thread to continue, allowing the user to process the data later. Multiple I/O requests can be pending at the same time.

However, there are several disadvantages to the FSO. As noted earlier, it works only with text files. To manipulate binary files, you must use pointers to an address in memory, or byte arrays, which are not supported by the object.

Moreover, when you read or write a large amount of content, the information stored in the buffer may create a big memory hit. Finally, you cannot manage permissions or file and folder attributes.

Code Example

The following example uses an instance of the **FileSystemObject** object to read a file and write its contents.

```
Public Class UsesScripting
   Public Shared Sub Main()
      Dim fileSystem As New Scripting.FileSystemObject()
      Dim file As Scripting.TextStream
      file = fileSystem.OpenTextFile("c:\test.txt", _
      Scripting.IOMode.ForReading, False, _
      Scripting.Tristate.TristateUseDefault)
      Dim contents As String = file.ReadAll()
      Console.WriteLine(contents)
      file.Close()
   End Sub
End Class
```

See Also

File Access with Visual Basic .NET | File Access with Visual Basic Run-Time Functions | Accessing Files with FileSystemObject

File Access with Visual Basic .NET

The .NET **System.IO** model provides an object-based tool for working with folders and files. Like **FileSystemObject** (as described in Accessing Files with FileSystemObject), it allows you to use standard *object.method* syntax — with its own set of properties, methods, and events — to process text and data, giving your applications the ability to read from and write to files easily.

The .NET **System.IO** namespace includes a class library that facilitates string, character, and file manipulation. These classes contain properties, methods, and events for creating, copying, moving, and deleting files. And, since both strings and numeric data types are supported, they also allow you to incorporate data types in files. The most commonly used classes are **FileStream**, **BinaryReader, BinaryWriter**, **StreamReader**, and **StreamWriter**.

In This Section

File Access Through the FileStream Class
Provides information about general file-access issues.
File Access Through BinaryReader and BinaryWriter Classes
Describes how to manipulate binary data types.
File Access Through StreamReader and StreamWriter Classes
Describes how to handle string data types.
Drive and Folder Access
Describes how to work with drives and folders via the **Directory** class.
Walkthrough: Manipulating Files Using .NET Framework Methods
Demonstrates file access using the **StreamReader** and **StreamWriter** classes.

Related Sections

Code: Writing Text to a File (Visual Basic)

Demonstrates how to write to a text file.

Code: Reading from a Text File (Visual Basic)
Demonstrates how to read from a text file.
System.IO Namespace
Summarizes synchronous and asynchronous reading and writing between data streams and files.

File Access Through the FileStream Class

The **FileStream** class provides access to files, including the standard input, output, and error devices. You use it with the **File** class, which aids in creating **FileStream** objects and provides routines for creating, copying, deleting, moving, and opening files. Another class, **Path**, provides a means of manipulating strings that represent folder or file paths.

The **FileAccess**, **FileMode**, and **FileShare** enumerations contain the flags used by the constructors of the **FileStream** class, and affect how the underlying file is created, opened, and shared. The **FileEnumerator** enumeration provides a means of reading the information maintained by the host file system for a specified group of files and directories. The **FileEnumerator** enumeration allows read access to information such as access and creation dates.

When used in conjunction with **FileEnumerator**, the **FileAttributes** enumeration enables the gathering of file-specific information. The **FileAttributes** enumeration returns the file's stored attributes, such as whether it is an archive, compressed, a directory, encrypted, hidden, read-only, a system file, or a temporary file.

See Also

File Access with Visual Basic .NET | File Access Through BinaryReader and BinaryWriter Classes |
File Access Through StreamReader and StreamWriter Classes | Drive and Folder Access | System.IO Namespace |
Code: Reading from a Text File (Visual Basic) | Code: Writing Text to a File (Visual Basic)

File Access Through BinaryReader and BinaryWriter Classes

The **BinaryReader** and **BinaryWriter** classes may be more familiar to Visual Basic users as **DataReader** and **DataWriter**. Although the names have been changed for the System.IO Namespace model, the underlying functionality remains the same. The **BinaryReader** class is used for reading strings and elementary data types, whereas the **BinaryWriter** class writes elementary types in binary data from a sequential text file (sometimes referred to as a *text stream*).

> **Security Note** When writing to files, an application may need to create a file if the file to which it is trying to write does not exist. To do so, it needs permission for the directory in which the file is to be created. However, if the file already exists, the application only needs **Write** permission to the file itself. Wherever possible, it is more secure to create the file during deployment and only grant **Write** permission to that file, rather than to the entire directory. It is also more secure to write data to user directories than to the root directory or the Program Files directory.

The following example demonstrates how to read and write to a newly created file:

```
Imports System.IO
Private Sub MyButton_Click (ByVal sender As System.Object, ByVal e As System.EventArgs) Handl
es MyButton.Click)
' Code throws an exception if the file already exists.
   Dim fs As FileStream = New FileStream("c:\data.txt", _
   FileMode.CreateNew)
   Dim w As BinaryWriter = New BinaryWriter(fs)
   Dim r As BinaryReader = New BinaryReader(fs)
   Dim i As Integer

   ' Takes the series of integers and stores them in a buffer.
   For i = 0 To 11
      w.Write(CStr(i))
   Next

   ' Sets the pointer to the beginning of the file.
   w.Seek(0, SeekOrigin.Begin)

   ' Writes the contents of the buffer to the console.
   For i = 0 To 11
      Console.WriteLine(CStr(i))
   Next

   ' Reads from file and writes it one character at a time.
   For i = 0 To 11
      Dim NewString As String
      NewString = r.read()
      Console.WriteLine(NewString)
   Next
End Sub
```

> **Note** Both **BinaryReader** and **BinaryWriter** read and write information as binary rather than text. To process text information, see File Access Through StreamReader and StreamWriter Classes.

See Also

File Access with Visual Basic .NET | File Access Through the FileStream Class |
File Access Through StreamReader and StreamWriter Classes | Drive and Folder Access | Serialization | System.IO Namespace |
Code: Reading from a Text File (Visual Basic) | Code: Writing Text to a File (Visual Basic)

File Access Through StreamReader and StreamWriter Classes

The System.IO Namespace class allows you to read and write characters to and from files as streams, or contiguous groups of data, using specific encoding to convert characters to and from bytes. It includes a pair of classes, **StreamReader** and **StreamWriter**, which enable you to read or write a sequential stream of characters to or from a file. The **StreamReader** and **StreamWriter** classes mirror the functionality of the **BinaryReader** and **BinaryWriter** classes, but they read and write information as text rather than binary. For more information, see File Access Through BinaryReader and BinaryWriter Classes.

The **StreamReader** class is derived from an abstract class called **TextReader**, which reads characters from a stream or file. The **StreamWriter** class is derived from an abstract class called **TextWriter**, which writes characters to a stream or file.

> **Security Note** When writing to files, an application may need to create a file if the file to which it is trying to write does not exist. To do so, it needs permission for the directory in which the file is to be created. However, if the file already exists, the application only needs **Write** permission to the file itself. Wherever possible, it is more secure to create the file during deployment and only grant **Write** permission to that file, rather than to the entire directory. It is also more secure to write data to user directories than to the root directory or the Program Files directory.

Reading From and Writing to Files

The following example writes a line of text to a file.

```
' Write text to a file
Sub WriteTextToFile()
    Dim file As New System.IO.StreamWriter("c:\test.txt")
    file.WriteLine("Here is the first line.")
    file.Close()
End Sub
```

The following example reads the text from a file into one string variable and writes the text to the console.

```
Sub ReadTextFromFile()
    Dim file As New System.IO.StreamReader("c:\test.txt")
    Dim words As String = file.ReadToEnd()
    Console.WriteLine(words)
    file.Close()
End Sub
```

The following example adds text to an existing file.

```
Sub AppendTextToFile()
    Dim file As New System.IO.StreamWriter("c:\test.txt", True)
    file.WriteLine("Here is another line.")
    file.Close()
End Sub
```

The following example reads one line at a time from a file and prints each line to the console.

```
Sub ReadTextLinesFromFile()
    Dim file As New System.IO.StreamReader("c:\test.txt")
    Dim oneLine As String
    oneLine = file.ReadLine()
    While (oneLine <> "")
        Console.WriteLine(oneLine)
        oneLine = file.ReadLine()
    End While
    file.Close()
End Sub
```

File Encoding

Both the **StreamReader** and **StreamWriter** classes use UTF-8 encoding by default. UTF-8 encoding handles Unicode characters correctly and ensures that localized versions of the operating system are consistent with each other.

You can detect a file's encoding automatically with **StreamReader**, or you can specify it as a parameter on a constructor.

StreamWriter takes an encoding parameter on its constructor. If encoding is specified, Visual Basic writes to the file to indicate the encoding used.

See Also

File Access with Visual Basic .NET | File Access Through BinaryReader and BinaryWriter Classes |
File Access Through the FileStream Class | Drive and Folder Access | System.IO Namespace |
Code: Reading from a Text File (Visual Basic) | Code: Writing Text to a File (Visual Basic)

Drive and Folder Access

Visual Basic .NET allows you to work with folders and drives via the **Directory** class. This class gives you the ability to create, edit, and delete folders, as well as maintain the drives on your system. Frequently used methods of the **Directory** class are listed in the following table:

Method	Description
CreateDirectory	Creates all directories specified by *path*.
Delete	Overloaded. Deletes a directory and its contents.
Exists	Determines whether the given path refers to an existing **DirectoryInfo** object on dis k.
GetDirectories	Overloaded. Gets an array of directories in the current directory.
GetDirectoryRoot	Returns the root portion of the specified path.
GetLogicalDrives	Retrieves names of logical drives on current machine.
GetParent	Retrieves parent directory of the specified path.
Move	Moves a **DirectoryInfo** object and its contents to a new path.
SetCurrentDirectory	Sets the current directory.

See Also

File Access with Visual Basic .NET | System.IO Namespace | File Access with Visual Basic Run-Time Functions | File Access Through StreamReader and StreamWriter Classes

Walkthrough: Manipulating Files Using .NET Framework Methods

After completing this walkthrough, you will be able to open and read a file using the **StreamReader** class, check to see if a file is being accessed, search for a string within a file read with an instance of the **StreamReader** class, and write to a file using the **StreamWriter** class.

Creating the Application

Start Visual Studio and begin the project by creating a form that the user can use to write to the designated file.

To create the project

1. On the **File** menu, point to **New**, and then click **Project**.

   The **New Project** dialog box appears.

2. In the **Project Types** pane, click **Visual Basic Projects**, and then click **Windows Application** in the **Templates** pane.
3. In the **Name** box type MyDiary and click **OK**.

   Visual Studio adds the project to **Solution Explorer**, and the **Windows Forms Designer** opens.

4. Add the controls in the following table to the form and set the corresponding values for their properties.

Object	Properties	Value
Button	Name Text	Submit Submit Entry
Button	Name Text	Clear Clear Entry
TextBox	Name Text Multiline	Entry Please enter something . False

Writing to the File

To add the ability to write to a file via the application, use the **StreamWriter** class. **StreamWriter** is designed for character output in a particular encoding, whereas the **Stream** class is designed for byte input and output. Use **StreamWriter** for writing lines of information to a standard text file. For more information on the **StreamWriter** class, see StreamWriter Class. To add writing functionality:

1. To open the **Code Editor**, select **Code** from the **View** menu.
2. Because the application references the **System.IO** namespace, add the following statements at the very beginning of your code, before the class declaration for the form, which begins `Public Class Form1`.

   ```
   Imports System
   Imports System.IO
   ```

3. Before writing to the file, you must create an instance of a **StreamWriter** class. Return to the **Windows Forms Designer** by selecting **Designer** from the **View** menu.
4. Double-click the **Submit** button to create a click event handler for the button and then add the following code to it:

   ```
   Dim fw As StreamWriter
   ```

   Note The Visual Studio Integrated Development Environment (IDE) returns to the **Code Editor** after the double-click and positions the cursor within the event handler where you should add the code.

5. To write to the file, use the **Write** method of the **StreamWriter** class. Add the following code directly after `Dim fw As`

`StreamWriter`. You do not need to worry that an exception will be thrown if the file is not found, because it will be created if it doesn't already exist.

```
    Dim ReadString As String
    Try
       'Pass the file path and name to the StreamWriter constructor.
       'Indicate that Append is True, so file will not be overwritten.
       fw = New StreamWriter("C:\MyDiary.txt", True)
       ReadString = Entry.Text
       fw.WriteLine(ReadString)
    Finally
       'Close the file.
       fw.Close()
    End Try
```

6. To make sure that the user cannot submit a blank entry, add the following code directly aft`er Dim ReadString As String`.

```
If (Entry.Text = "" Or Entry.Text = "Please enter something.") Then
   Entry.Text = "Please enter something."
   Return
End If
```

7. 8Because this is a diary, the user will want to assign a date to each entry. Insert the following code after `fw = New StreamWriter("C:\MyDiary.txt", True)` to set the variable `Today` to the current date.

```
Dim Today As DateTime
Today = Now.ToString
fw.Write(CStr(today))
fw.Write(ControlChars.CrLf)
```

8. Finally, attach code to clear the TextBox. Add the following code to the Clear button's Click event.

```
Entry.Text = ""
```

To display the diary, additional functionality will display the latest entry in the **DisplayEntry** TextBox. Further refinement adds a ComboBox that displays various entries and from which a user can select an entry to display in the **DisplayEntry** TextBox. An instance of the **StreamReader** class will read from MyDiary.txt. Like the **StreamWriter** class, **StreamReader** is intended for use with text files.

For this section of the walkthrough, add the controls in the following table to the form and set the corresponding values for their properties.

Object	**Properties**	**Value**
TextBox	Name	DisplayEntry
	Visible	False
	Size	120,60
	Multiline	True
Button	Name	Display
	Text	Display
Button	Name	GetEntries
	Text	Get Entries

ComboBox	Name Text Enabled	PickEntries Select an Entry False

To populate the ComboBox

1. The **PickEntries** ComboBox will be used to display the dates on which a user submitted each entry, so the user can select an entry from a specific date. Create a Click event handler to the **GetEntries** button and add the following code to it:

```
Dim fr As StreamReader
Dim FileString As String
FileString = ""
Try
   fr = New StreamReader("C:\MyDiary.txt")
   PickEntries.Items.Clear()
   PickEntries.Enabled = True
   Do
      FileString = fr.ReadLine
      If IsDate(FileString) Then
         PickEntries.Items.Add(FileString)
      End If
   Loop Until (FileString Is Nothing)
Finally
   fr.Close()
End Try
PickEntries.Enabled = True
```

2. To test your code, press **F5** to compile the application and click **Show Entries**. Click the dropdown arrow in the ComboBox to display the entry dates.

To choose and display individual entries

1. Create a Click event handler for the `Display` button and add the following code to it:

```
Dim fr As StreamReader
Dim ReadString As String
'Make sure ReadString begins empty.
ReadString = ""
Dim FileString As String
fr = New StreamReader("C:\MyDiary.txt")
'If no entry has been selected, show the whole file.
If PickEntries.Enabled = False Or PickEntries.SelectedText Is Nothing Then
   Do
      'Read a line from the file into FileString.
      FileString = fr.ReadLine
      'add it to ReadString
      ReadString = ReadString & ControlChars.CrLf & FileString
   Loop Until (FileString = Nothing)
Else
   'An entry has been selected, find the line that matches.
   Do

      FileString = fr.ReadLine
   Loop Until FileString = CStr(PickEntries.SelectedItem)
   FileString = CStr(PickEntries.SelectedItem) & ControlChars.CrLf
   ReadString = FileString & fr.ReadLine
```

```
    'Read from the file until EOF or another Date is found.
    Do Until ((fr.Peek < 0) Or (IsDate(fr.ReadLine)))
        ReadString = ReadString & fr.Readline
    Loop
End If
fr.Close()
DisplayEntry.Visible = True
DisplayEntry.Text = ReadString
```

2. To test your code, press **F5** to compile the application, submit an entry. Click **Show Entries**, select an entry from the ComboBox, and then click **Display**. The contents of the selected entry appears in the `DisplayEntry` TextBox.

Finally, additional functionality enables users to delete or modify an entry by using Delete Entry and Edit Entry buttons. Both buttons remain disabled unless an entry is being displayed.

1. Add the controls in the following table to the form and set the corresponding values for their properties.

Object	Properties	Value
Button	Name	DeleteEntry
	Text	Delete Entry
	Enabled	False
Button	Name	EditEntry
	Text	Edit Entry
	Enabled	False
Button	Name	SubmitEdit
	Text	Submit Edit
	Enabled	False

2. Add the following code to the `Display` button's click event, after `DisplayEntry.Text = ReadString`.

```
DeleteEntry.enabled = True
```

3. Create a Click event handler for the `DeleteEntry` button and add the following code:

```
Dim fr As StreamReader
Dim ReadString As String
Dim WriteString As String
Dim ConfirmDelete As MsgBoxResult
fr = New StreamReader("C:\MyDiary.txt")
ReadString = fr.ReadLine
' Read through the textfile
Do Until (fr.Peek < 0)
      ReadString = ReadString & vbCrLf & fr.ReadLine
Loop
WriteString = Replace(ReadString, DisplayEntry.Text, "")
fr.Close()
' Check to make sure the user wishes to delete the entry
ConfirmDelete = MsgBox("Do you really wish to delete this entry?", _ MsgBoxStyle.OKCancel
)
If ConfirmDelete = MsgBoxResult.OK Then
    File.Delete("C:\MyDiary.txt")
    Dim fw As StreamWriter = File.CreateText("C:\MyDiary.txt")
    fw.WriteLine(WriteString)
    fw.Close()
    ' Reset controls on the form
```

```
    DisplayEntry.Text = ""
    PickEntries.Text = ""
    PickEntries.Items.Clear()
    PickEntries.Enabled = False
    DeleteEntry.Enabled = False
End If
```

4. When a user displays an entry, the `EditEntry` button becomes enabled. Add the following code to the click event of the `Display` button after `DisplayEntry.Text = ReadString`:

```
EditEntry.Enabled = True
```

5. Create a Click event handler for the `EditEntry` button and add the following code to it:

```
Entry.Text = DisplayEntry.Text
SubmitEdit.Enabled = True
```

6. Create a Click event handler for the `SubmitEdit` button and add the following code to it:

```
Dim fr As StreamReader
Dim ReadString As String
Dim WriteString As String
If Entry.Text = "" Then
    MsgBox("Use Delete to Delete an Entry")
    Return
End If
fr = New StreamReader("C:\MyDiary.txt")
ReadString = fr.ReadLine
Do Until (fr.Peek < 0)
    ReadString = ReadString & vbCrLf & fr.ReadLine
Loop
WriteString = Replace(ReadString, DisplayEntry.Text, Entry.Text)
fr.Close()
File.Delete("C:\MyDiary.txt")
Dim fw As StreamWriter = File.CreateText("C:\MyDiary.txt")
fw.WriteLine(WriteString)
fw.Close()
DisplayEntry.Text = Entry.Text
Entry.Text = ""
EditEntry.Enabled = False
SubmitEdit.Enabled = False
```

To test your code, press **F5** to compile the application. Click **Show Entries**, select an entry, and then click **Display**. The entry displays in the `DisplayEntry` TextBox. Click **Edit Entry**. The entry displays in the `Entry` TextBox. Edit the entry in the `Entry` TextBox and click **Submit Edit**. Open the MyDiary.txt file to confirm your correction. Now select an entry and click **Delete Entry**. When the MsgBox requests confirmation, click **OK**. Close the application and open MyDiary.txt to confirm the deletion.

See Also

Choosing Among File I/O Options in Visual Basic .NET | StreamReader Class | StreamWriter Class | Accessing Files with FileSystemObject | Visual Basic Language Walkthroughs

File Access with Visual Basic Run-Time Functions

In earlier versions of Visual Basic, files were processed using the Open statement and other related statements and functions. Visual Basic .NET includes compatible functions as part of its library, specifically in the **System.IO.File** namespace (for details, see System.IO Namespace). The topics in this section show how to process files directly to create, manipulate, and store text and other data using available file I/O functions.

In This Section

File Access Types and Functions
Explains statements and functions used with the three types of file access.
Sequential File Access
Covers how to open files for sequential access and edit their data.
Random File Access
Provides an overview of creating and reading random-access files.
Binary File Access
Discusses creating and accessing binary files.
Data Access Functions
Describes statements typically used to write data to and read data from files.
Walkthrough: Manipulating Files and Folders in Visual Basic .NET
Demonstrates how to read and write to a text file using Visual Basic Run-Time functions.

Related Sections

Accessing Files with FileSystemObject
Discusses creating objects, accessing their properties, and using the appropriate methods.
System.IO Namespace
Summarizes synchronous and asynchronous reading and writing between data streams and files.

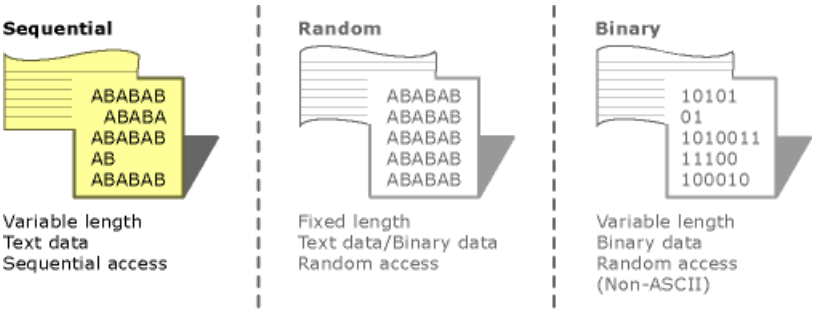
File Access Types and Functions

By itself, a file is nothing more than a series of related bytes of data on a disk. When your application accesses a file, it must assume whether the bytes represent characters, data records, integers, strings, and so on. You tell your application what to assume by specifying the file's access type.

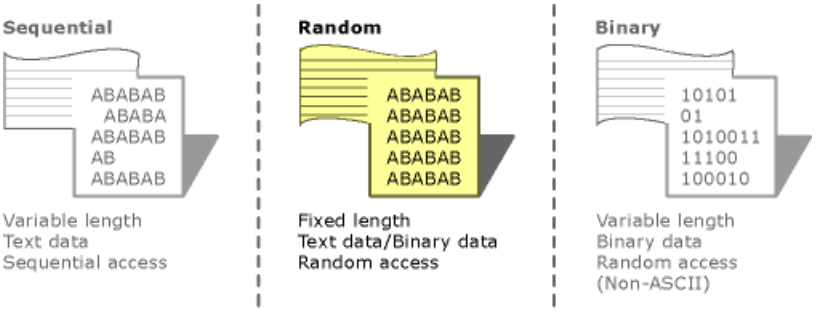
The file access type you use depends on what kind of data the file contains. Visual Basic provides three types of file access:

- Sequential, for reading and writing text files in continuous blocks.
- Random, for reading and writing text or binary files structured as fixed-length records.
- Binary, for reading and writing arbitrarily structured files.

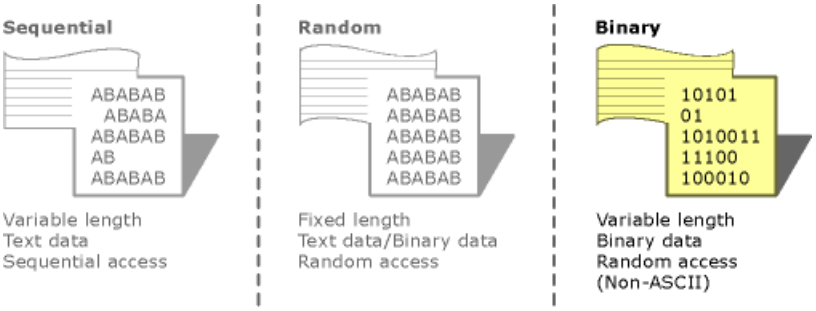
Sequential access is designed for use with plain text files. Each character in the file is assumed to represent either a text character or a text-formatting sequence, such as a newline character. Data is stored as ANSI characters.



A file opened for *random access* is assumed to be composed of a set of identical-length records, or sets of fields containing information. You can employ user-defined types to create records made up of numerous fields, each of which can have different data types. Data is stored as binary information.



Binary access allows you to use files to store data in any form that suits your needs, whether it is numeric, string, or both. Binary access is similar to random access except that no assumptions are made about data type or record length. However, you must know precisely how the data was written to the file to retrieve it correctly. For example, if you store a series of names and phone numbers, you need to remember that the first field (the name) is text and the second field (the phone number) is numeric.



File Access Functions

The following functions are used with sequential, random, and binary file access.

Dir Function	EOF Function	FileCopy Function
FileDateTime Function	FileLen Function	FreeFile Function
GetAttr Function	Loc Function	LOF Function
Seek Function	SetAttr Function	

The following table lists all file access functions available for each of the three types of direct file access.

Functions	Sequential	Random	Binary
FileClose Function	X	X	X
FileOpen Function	X	X	X
Input Function	X		X
InputString Function	X		
LineInput Function	X		
Print, PrintLine Functions	X		
Write, WriteLine Functions	X		

Note For additional information on file access functions, look up the function topic in the index.

See Also

Data Access Functions | Sequential File Access | Random File Access | Binary File Access

Sequential File Access

Using **System.IO** is the recommended way to create text files, but should you need to use older text file-creation methods, this section shows you how.

Sequential access works best when you want to process files consisting only of text, such as files created with a typical text editor, not files in which data is divided into a series of records. Sequential access is not well suited for storing long series of numbers, because each number is stored as a character string. Thus, a four-digit number would require four bytes of storage instead of the two bytes required to store the same number as an integer.

In This Section

Opening Files for Sequential Access
Describes how to gain access for input, output, and appending characters within a file.

Editing Files Opened for Sequential Access
Explains using **Line Input**, **InputString**, and **Input** to edit files.

Writing Strings to Sequential-Access Files
Discusses how to use **Print** and **Write** functions to store strings in text files.

Related Sections

Accessing Files with FileSystemObject
Discusses creating objects, accessing their properties, and using the appropriate methods.

Opening Files for Sequential Access

When you open a file for sequential access, you must specify whether to input characters from the file (**Input** mode), output characters to the file (**Output** mode), or append characters to the file (**Append** mode).

To open a file for sequential access

- Use the FileOpen Function with the following syntax:

```
FileOpen(FileNumber, FileName, OpenMode.Input)
```

When you open a sequential file for **Input**, the file must already exist; otherwise an error occurs. When you try to open a nonexistent file for **Output** or **Append**, however, the **FileOpen** statement creates the file first and then opens it.

After opening a file for an **Input**, **Output**, or **Append** operation, you must close it with the FileClose Function statement before reopening it for another type of operation.

> **Security Note** When writing to files, an application may need to create a file if the file to which it is trying to write does not exist. To do so, it needs permission for the directory in which the file is to be created. However, if the file already exists, the application only needs **Write** permission to the file itself. Wherever possible, it is more secure to create the file during deployment and only grant **Write** permission to that file, rather than to the entire directory. It is also more secure to write data to user directories than to the root directory or the Program Files directory.

See Also

Sequential File Access | Editing Files Opened for Sequential Access | File Access with Visual Basic Run-Time Functions | FileOpen Function | Random File Access | Binary File Access | Writing Strings to Sequential-Access Files

Editing Files Opened for Sequential Access

To edit a file, you must first read its contents to program variables, then change the variables, and finally, write the variables back to the file. The following sections discuss how to edit records opened for sequential access.

To read strings from files

1. Retrieve the contents of a text file by opening it for **Input**.
2. Use the LineInput, InputString, or Input functions to copy the file into program variables.

Visual Basic provides statements and functions that read and write sequential files one character at a time or one line at a time. For example, the following code fragment reads a file line by line:

```
Dim LinesFromFile, NextLine As String, FileNum As Integer
Do Until EOF(FileNum)
   Nextline = LineInput(FileNum)
   LinesFromFile = LinesFromFile & NextLine & Chr(13) & Chr(10)
Loop
```

Although the **LineInput** function recognizes the end of a line when it comes to the carriage return/line feed sequence, it does not include that sequence when it reads the line into the variable. If you want to retain the carriage return/line feed, your code must add it.

You can use the **InputString** function to copy a specified number of characters from a file to a variable, provided the variable is large enough. For example, the following code uses **InputString** to copy `CharCount` number of characters to a variable:

```
LinesFromFile = InputString(FileNum, CharCount)
```

You can also use the **Input** function, which reads a list of numbers and/or string expressions from a file. For example, to read in a line from a mailing list file, you might use the following statements:

```
Input(FileNum, fname)
Input(FileNum, lname)
Input(FileNum, street)
Input(FileNum, city)
Input(FileNum, state)
Input(FileNum, zip)
```

See Also

Sequential File Access | Opening Files for Sequential Access | File Access with Visual Basic Run-Time Functions | FileOpen Function | Random File Access | Binary File Access | Writing Strings to Sequential-Access Files

Writing Strings to Sequential-Access Files

You can add data in the form of strings to existing files via the Print Function or in the form of numbers and string expressions through the Write Function.

To write strings to files

1. Use the FileOpen Function to open a text file for **Output** or **Append**.
2. Use the **Print** function to write the strings to the file as in the following example, which a text editor might use to copy the contents of a text box into a file:

```
Print(FileNum, TheBox.Text)
```

Visual Basic also supports the **Write** function, which writes a list of numbers and/or string expressions to a file. It automatically separates each expression with a comma and puts quotation marks around **String** expressions, as illustrated in the following code:

```
Dim AnyString As String, AnyNumber, FileNum As Integer
AnyString = "AnyCharacters"
AnyNumber = 23445
Write(FileNum, AnyString, AnyNumber)
```

The previous code segment writes two expressions to the file specified by `FileNum`. The first contains a string, and the second contains the number 23445. Therefore, Visual Basic writes the following characters (including all punctuation) to the file:

```
"AnyCharacters",23445
```

> **Note** If you are maintaining a database using **Write** and **Input** with sequential access, consider using **FilePutObject** and **FileGetObject** with random or binary access instead; both types of access are better suited to record-oriented data.

One of the primary differences between **Print** and **Write** is that **Print** formats the string, while **Write** does not. For this reason, **Print** does not include quotation marks around the string, which is formatted as the user would like to see it, while **Write** does.

Use a function that writes each string to a new line, such as **PrintLine** or **WriteLine** if you need strings separated by line.

See Also

Sequential File Access | Opening Files for Sequential Access | Editing Files Opened for Sequential Access | File Access with Visual Basic Run-Time Functions | Random File Access | Binary File Access | Print, PrintLine Functions | Write, WriteLine Functions | LineInput Function

Random File Access

The bytes in random-access files form records of uniform length, each containing one or more fields. A record with one field corresponds to any standard type, such as an integer or fixed-length string. A record with more than one field corresponds to a user-defined type.

In this section you declare, open, edit, and write to a random-access file, using a hypothetical Employee Records database as an example. Each file-access aspect of the example is discussed separately in the topics listed immediately below.

In This Section

Declaring Variables for Random File Access
Discusses creating variables and user-defined types for random access.
Opening Files for Random Access
Explains how to use the **FileOpen** statement to work on existing files.
Editing Files Opened for Random Access
Explains how to move records from a random file into variables.
Writing Variables to Records
Covers how to write variables back to records after editing their values.

Related Sections

File Access with Visual Basic Run-Time Functions
Covers processing files directly to create, manipulate, and store text and other data.
Accessing Files with FileSystemObject
Discusses creating objects, accessing their properties, and using the appropriate methods.

Declaring Variables for Random File Access

Before your application opens a file for random access, it should declare all variables required to handle data from the file. This includes user-defined types, which correspond to records in the file, as well as standard types for other variables that hold data related to processing a random-access file.

Before opening a file for random access, define a structure that corresponds to the records the file contains or will contain. For example, a hypothetical Employee Records database might contain a user-defined data type called `Person` as follows:

```
Structure Person
   Public ID As Integer
   Public MonthlySalary As Decimal
   Public LastReviewDate As Long
   <VBFixedString(15)> Public FirstName As String
   <VBFixedString(15)> Public LastName As String
   <VBFixedString(15)> Public Title As String
   <VBFixedString(150)> Public ReviewComments As String
End Structure
```

Declaring Field Variables in a Structure Definition

Because all records in a random-access file must have the same length, it is often useful for string elements in a user-defined type to have a fixed length also. In the `Person` type declaration above, for example, `FirstName` and `LastName` have a fixed length of 15 characters. To declare a fixed-length string, set its length with the **VBFixedString** attribute.

If the actual string contains fewer characters than the fixed length of the string element to which it is written, Visual Basic fills the trailing spaces in the record with blanks (character code 32). If the string is longer than the field size, Visual Basic truncates it. (If you use variable-length strings, the total size of any record stored with the FilePut Function or retrieved with the FileGet Function must not exceed the record length specified in the FileOpen Function).

After defining a structure that corresponds to a typical record, declare any other variables that your application needs to process a file opened for random access. The Employee Records database, for instance, declares `Employee`, `Position`, and `LastRecord` variables, as follows:

```
' Declare a record variable.
Public Employee As Person
' Track the current record.
Public Position As Long
' Get the number of the last record in the file.
Public LastRecord As Long
```

See Also

Opening Files for Random Access | Editing Files Opened for Random Access | Writing Variables to Records |
File Access with Visual Basic Run-Time Functions | Sequential File Access | Binary File Access

Opening Files for Random Access

After you create the file, you need to open it to make any modifications.

To open a file for random access

- Use the FileOpen Function with the following syntax:

```
FileOpen(FileNumber, FileName, OpenMode.Random, , , RecordLength)
```

FileNumber and *FileName* specify the file number and name of the file to be opened, respectively. *RecordLength* specifies the size of each record in bytes. If *RecordLength* is less than the actual length of the record written to the file, an error is generated. If *RecordLength* is greater than the actual length of the record, the record is written, although some disk space may be wasted. Note that every **String** variable in Visual Basic stores an ASCII string, and that you must specify the byte length of that ASCII string.

> **Security Note** When writing to files, an application may need to create a file if the file to which it is trying to write does not exist. To do so, it needs permission for the directory in which the file is to be created. However, if the file already exists, the application only needs **Write** permission to the file itself. Wherever possible, it is more secure to create the file during deployment and only grant **Write** permission to that file, rather than to the entire directory. It is also more secure to write data to user directories than to the root directory or the Program Files directory.

In Declaring Variables for Random File Access, a hypothetical Employee Records database first defined a `Person` data type and then declared a record variable of that type as well as two other variables for processing records. The following code continues the Employee Records example, demonstrating how to open a random-access file to accept `Employee` data of user-defined type `Person`:

```
Sub Main()
   Dim FileNum As Integer, RecLength As Long, Employee As Person
   ' Calculate the record length.
   RecLength = Len(Employee)
   ' Get the next available file number.
   FileNum = FreeFile
   ' Open the new file with the FileOpen statement.
   FileOpen(FileNum, "MYFILE.DAT", OpenMode.Random, , , RecLength)
End Sub
```

See Also

Declaring Variables for Random File Access | Editing Files Opened for Random Access | Writing Variables to Records | File Access with Visual Basic Run-Time Functions | Sequential File Access | Binary File Access

Editing Files Opened for Random Access

To edit a random-access file, first read records from the file into program variables, and then change the values in the variables. In Declaring Variables for Random File Access and Opening Files for Random Access, a hypothetical Employee Records database defined variables for reading and writing employee records. The following step describes how to read and copy employee records using those variables.

To read and copy records into variables

- Use the FileGetObject Function, which copies a record from the Employee Records file into the `Employee` variable:

  ```
  FileGet(FileNum, Employee, Position)
  ```

  In this line of code, `FileNum` contains the number that the FileOpen Function used to open the file, `Position` contains the number of the record to copy, and `Employee`, declared as the user-defined type `Person`, receives the contents of the record.

See Also

Declaring Variables for Random File Access | Opening Files for Random Access | Writing Variables to Records | File Access with Visual Basic Run-Time Functions | Sequential File Access | Binary File Access | FilePutObject Function | FileGetObject Function

Writing Variables to Records

After editing records read into program variables from a random-access file, use the FilePut Function to replace or add records. In Declaring Variables for Random File Access and Opening Files for Random Access, a hypothetical Employee Records database defined variables for reading and writing employee records. The following steps describe how to replace and add employee records using those variables.

To replace a record

- Use **FilePut** to specify the position of the record you want to replace. The Random File Access example illustrates this in the following code:

  ```
  FilePut(FileNum, Employee, Position)
  ```

  This code replaces the record number specified by `Position` with the data in the `Employee` variable.

To add a record

- Using **FilePut**, set the value of the `Position` variable equal to one more than the number of records in the file. To add a record to a file that contains five records, for example, set `Position` equal to 6.

  In the case of the Employee Records example, substituting the following statements for the **FilePut** statement above adds a record to the end of the file instead of overwriting the one specified by `Position`:

  ```
  LastRecord = LastRecord + 1
  FilePut(FileNum, Employee, LastRecord)
  ```

Deleting Records

You can delete a record's contents by clearing its fields, but the record will still exist in the file. In most cases, you don't want empty records in your file because they waste space. To avoid this, you can copy the remaining records to a new file, and then delete the old one; or, you can make a note of the location of the empty record and use that slot on the next insert operation.

To delete a record

- Clear the record's fields.

To remove a deleted record

1. Create a new file.
2. Copy all the valid records from the original file into the new file.
3. Close the original file and use the **Kill** function to delete it.
4. Use the **Rename** function to rename the new file with the name of the original file.

See Also

Declaring Variables for Random File Access | Opening Files for Random Access | Editing Files Opened for Random Access | File Access with Visual Basic Run-Time Functions | Sequential File Access | Binary File Access | Kill Function | Rename Function | FilePut Function

Binary File Access

Use binary access when it is important to keep file size small. Because binary access doesn't require fixed-length fields, the type declaration can omit the string length parameters. This enables you to conserve disk space by building variable-length records.

In This Section

Opening and Closing Files for Binary Access
Describes how to use the **FileOpen** and **FileClose** functions to access a file.

Reading from and Writing to a File Opened for Binary Access
Describes how to read from and write to a file using the **FilePut**, **FilePutObject**, **FileGet** and **FileGetObject** functions.

Related Sections

File Access with Visual Basic Run-Time Functions
Covers processing files directly to create, manipulate, and store text and other data.

Accessing Files with FileSystemObject
Covers using **FileSystemObject** to create and manipulate files.

Opening and Closing Files for Binary Access

Use binary access when it is important to keep file size small. Because binary access doesn't require fixed-length fields, you can conserve disk space by building variable-length records.

To open a file for binary access

- Use the FileOpen Function with the following syntax:

```
FileOpen(FileNumber, FileName, OpenMode.Binary)
```

To close a file for binary access

- Use the FileClose Function with the following syntax:

```
FileClose(FileNumber)
```

As you can see, opening files for binary access differs from Opening Files for Random Access in that the *RecordLength* expression is not specified. (If you do include a record length in a binary-access **FileOpen** function, it is ignored.)

To appreciate binary access, consider the Employee Records example discussed in the Random File Access section. This random-access file uses fixed-length records and fields to store information about employees. As a result, every record in the Employee Records database takes the same number of bytes, regardless of the actual contents of the fields.

You can minimize the size of Employee Records by using binary access. Because you are not required to use fixed-length fields, you can omit the string-length parameters from the type declaration, as in the following code:

```
Structure Person
   ID As Integer
   MonthlySalary As Decimal
   LastReviewDate As Long
   FirstName As String
   LastName As String
   Title As String
   ReviewComments As String
End Structure

Public Employee As Person   ' Defines a record.
```

Each employee record in the Employee Records file now occupies only the exact number of bytes required, because the fields are of variable lengths.

The drawback to binary input/output with variable-length fields is that you cannot randomly access records; you must instead access them sequentially to learn the length of each one. You can still go directly to a specified byte position in a file, but if the fields are of variable length, there is no direct way to know which record is at which byte position.

> **Security Note** When writing to files, an application may need to create a file if the file to which it is trying to write does not exist. To do so, it needs permission for the directory in which the file is to be created. However, if the file already exists, the application only needs **Write** permission to the file itself. Wherever possible, it is more secure to create the file during deployment and only grant **Write** permission to that file, rather than to the entire directory. It is also more secure to write data to user directories than to the root directory or the Program Files directory.

See Also

File Access with Visual Basic Run-Time Functions | Sequential File Access | Random File Access | FileOpen Function |FileGet Function |FilePut Function

Reading from and Writing to a File Opened for Binary Access

Binary access provides a number of advantages for certain tasks. For example, you can store information in a specific format, such as .zip, .mpeg or .xls. Binary access is similar to random access except that no assumptions are made about data type or record length. Because binary access does not require fixed-length fields, you can write variable-length fields to such a file. Binary access can thus keep file size smaller than random access because each field can occupy only the space it requires.

To open a file for binary access, specify **OpenMode.Binary** with the **FileOpen** statement. After it has been opened, you can use the same functions to write to the file, such as **FilePut** and **FilePutObject**, that you would with a file opened for sequential or random access. To read from the file, use **FileGet** and **FileGetObject** functions. For more information, see File Access Types and Functions.

> **Security Note** When reading from files, do not make decisions about the contents of a file based on the file name extension. For example, a file named Form1.vb may not be a Visual Basic .NET source file.

The following example uses the **FilePut** function to write a string to a file opened for binary access. It assumes that there is a file named test.txt in the current directory.

```
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As_
System.EventArgs) Handles Button1.Click
   Dim MyString As String
   Dim ReadString As String
   Dim iFr As Short
   iFr = FreeFile()
   ReadString = CurDir() & "\test.txt"
   FileOpen(iFr, 1, OpenMode.Binary)
   MyString = "new information"
   FilePut(MyString)
   FileClose(iFr)
End Sub
```

The following example uses the **FileGet** function to display the first 15 characters contained in test.txt. Again, this example presupposes that test.txt exists.

```
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As_
System.EventArgs) Handles Button1.Click
     Dim MyString As String
     Dim ReadString As String
     Dim iFr As Short
     iFr = FreeFile()
     ReadString = CurDir() & "\test.txt"
     FileOpen(iFr, ReadString, OpenMode.Binary)
     MyString = New String(" "c, 15)
     FileGet(iFr, MyString)
     FileClose(iFr)
     MsgBox(MyString)
End Sub
```

You may wish to use information gathered from a file in your code. For example, the following code sample writes the contents of a textfile, test.txt, to a listbox, myListBox. It assumes that the file, test.txt, exists and contains several items separated by carriage returns.

```
Dim myListbox As ListBox
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As _
System.EventArgs) Handles Button1.Click
   Dim ReadString As String
   Dim iFr As Short
   Dim sTmp As String
   ReadString = CurDir() & "\test.txt"
   iFr = FreeFile()
   FileOpen(iFr, ReadString, OpenMode.Binary)
   sTmp = Space(FileLen(ReadString))
   FileGet(iFr, sTmp)
```

```
        FileClose(iFr)

        Dim strRecords() As String : strRecords = Split(sTmp, vbCrLf)
        'Clear the listbox
        MyListbox.Items.Clear()
        Dim lngRecord As Long
        For lngRecord = LBound(strRecords) To UBound(strRecords)
           myListbox.Items.Add(strRecords(lngRecord))
        Next lngRecord
    End Sub
```

See Also

Opening and Closing Files for Binary Access | Binary File Access | Random File Access | Sequential File Access | File Access with Visual Basic Run-Time Functions | FilePut Function | FilePutObject Function | FileGet Function | FileGetObject Function | File Access Types and Functions

Data Access Functions

Working with large amounts of data often requires that data be written to or read from a file. Before either operation can take place, however, the file must first be opened. The FileOpen Function lets you create and access files with one of three types of file access:

- Sequential access (**Input**, **Output**, and **Append** modes) is used for writing text files, such as error logs and reports.
- Random access (**Random** mode) is used to read and write data to a file without closing it. Random-access files keep data in records, which makes it easy to locate information quickly.
- Binary access (**Binary** mode) is used to read or write to any byte position in a file, such as storing or displaying a bitmap image.

  Do not use the **FileOpen** function to manipulate files that are usually edited by a specific application, such as Microsoft Office documents or registry files. Doing so will cause file corruption and loss of file integrity.

The following table shows the statements typically used when writing data to and reading it from files.

Access type	Writing data	Reading data
Sequential	Print, PrintLine Functions Write, WriteLine Functions	InputString Function LineInput Function
Random	FilePut Function	FileGet Function
Binary	FilePut Function	FileGet Function

See Also

File Access Types and Functions | Sequential File Access | Random File Access | Binary File Access

Walkthrough: Manipulating Files and Folders in Visual Basic .NET

This walkthrough provides an introduction to the fundamentals of file I/O in Microsoft Visual Basic .NET. To illustrate the features, it creates a small application, FileExplorer, which examines text files within a directory and provides information such as attributes, last access time, and the first 80 characters of the file. It also includes an option that writes the information to a log file.

Creating the Application

To begin the project, create a form from which users can select a directory, choose a file from the directory, and choose the information they want to retrieve about that file.

To create the project

1. On the **File** menu, point to **New**, and then click **Project**.

   The **New Project** dialog box appears.

2. In the **Project Types** pane, click **Visual Basic Projects**, and then click **Windows Application** in the **Templates** pane.
3. In the **Name** box, type FileExplorer to set the project name.

   Visual Studio adds the project to Solution Explorer, and the Windows Forms Designer opens.

4. Add the controls in the following table to the form, and set the corresponding values for their properties.

Object	Properties	Value
TextBox	Name Text	txtDirectory Directory
Button	Name Text	btnSubmit Submit
Button	Name Text	btnExamine Examine
ComboBox	Name Text	LstFilePick Select A File
CheckBox	Name Text Checked	chkAttributes Attributes True
CheckBox	Name Text Checked	chkFileLength File Length True
CheckBox	Name Text Checked	chkLastAccess Last Access Time True
CheckBox	Name Text Checked	chkFirstLine Show First Line True

CheckBox	Name	chkSave
	Text	Save Results
	Checked	False

Displaying the Current Directory

The FileExplorer application needs a starting point. Accordingly, the `txtDirectory` TextBox uses the **CurDir** function to return and display a string representing the current path.

To return the current directory

1. Create an event handler for `Form1_Load` by double-clicking the form.

   The Code Editor opens.

2. Add the following code, so the `txtDirectory` TextBox displays the current location.

```
txtDirectory.Text = CurDir()
```

3. Run the program to verify that the **CurDir** function returns the correct path.

   The `txtDirectory` TextBox displays the current directory.

Changing Directories

Since a user may want to select files in a different directory, the application uses the **ChDir** function to switch directories. To change to a different directory, a user enters a new path in the `txtDirectory` TextBox.

To change directories

1. Create a click event handler for `btnSubmit` by double-clicking the control on the form.

   The Code Editor opens.

2. Add the following code to the click event handler.

```
Dim NewPath As String
' NewPath holds the path the user has entered.
NewPath = txtDirectory.Text
' Change the location to NewPath.
ChDir(NewPath)
```

Verifying That a Valid Path Has Been Entered

The **ChDir** function requires the Path parameter, which may or may not include the drive letter. If the path does not specify the drive, **ChDir** uses the current drive. To avoid this potential source of confusion, the application incorporates a **Try...Catch** statement that catches such exceptions as well as exceptions that arise from the submission of a blank path.

To ensure valid paths

1. In the `btnSubmit_Click` event, after the code line `Dim NewPath As String`, add `Dim ErrorMessage As String` on a new line.
2. Before the code line `ChDir(NewPath)`, add a **Try** statement on its own line as follows. If you hit return, the Code Editor will automatically insert `Catch ex As Exception` and `End Try` statements. Remove these; you will add your own in the next step.

```
        Try
```

3. After the code line `ChDir(NewPath)`, add the following:

```
        ' This checks to make sure the path is not blank.
        Catch ex As Exception When NewPath = ""
            ErrorMessage = "You must enter a path."
```

```
        ' This catches errors caused by a path that is not valid.
        Catch
            ErrorMessage = "You must enter a valid path.  If trying" & _
            "to access a different drive, remember to include the drive" & _
            "letter."
        Finally
            ' Display the error message only if one exists.
            If ErrorMessage <> "" Then
               MsgBox(ErrorMessage)
            End If
         End Try
```

Displaying the Contents of the Directory in a ComboBox

To enable the application to display the contents of the current directory, you can use the **Dir** function, which returns a string representing the name of a file, directory, or folder matching a specified pattern. Since the **Dir** function returns only the first matching item, the application includes a **Do Until...Loop** to check each item. The following code added to the Submit button also restricts the types of files listed to .txt files.

> **Note** This process takes advantage of the fact that calling the **Dir** function without arguments leaves the previous matching pattern in place.

To display directory contents

1. Insert `Dim ContentItem As String` at the beginning of the `btnSubmit_Click` event.
2. After the line `End Try` insert:

```
LstFilePick.Items.Clear()
' Only look at the .txt files.
ContentItem = Dir("*.txt")
If ContentItem = "" Then
   ErrorMessage = "No text files found."
   MsgBox(ErrorMessage)
End If
' Iterate through the directory contents.
Do Until ContentItem = ""
   ' Add it to the list.
   lstFilePick.Items.Add(ContentItem)
   ' Move to the next textfile.
   ContentItem = Dir()
Loop
```

The gathered information displays in the `lstFilePick` ComboBox from which you can pick a specific file to examine.

Test the application by running it first on a directory that contains no .txt files and then on one that contains more than one .txt file. In the first instance, the application displays the appropriate error **MsgBox**. In the second, the application creates a list in the ComboBox of all .txt files within the directory specified in the `txtDirectory` TextBox.

Enabling a User to Select a File to Examine

Although the `lstFilePick` ComboBox displays all files in a directory, a user probably wants to select and examine a specific file.

- Create a click event handler for `btnExamine_Click` and add the following code to confirm a file's selection.

```
Dim thisFile As Object
thisFile = lstFilePick.SelectedItem
```

Enabling a user to determine what information to gather

Now that files display in the `lstFilePick` ComboBox, additional code enables a user to specify the reported information. For example, one user might want to merely know the date when the file was last accessed. Another user might want to know a file's size, too. A user checks or unchecks a series of CheckBoxes (`chkAttributes`, `chkLastAccess`, `chkFileLength`, `chkFirstLine`) to customize the results.

To display specific information

1. Declare these variables at the beginning of the `btnExamine_Click` event after **`thisFile = lstFilePick.SelectedItems`**:

```
Dim Attributes As FileAttribute
Dim AttributeResult As String
Dim StringAttributes As String
Dim LastAccess As Date
Dim Length As Long
Dim FirstLine As String
Dim FinalString As String
Dim ErrorMessage As String
Dim NewName As String
NewName = Ctype(thisfile, String)
If thisFile = Nothing Then
    MsgBox("You must select a file to examine.")
    Exit Sub
End If
```

   The **GetAttr** function returns a member of the **FileAttribute** enumeration representing the attributes of a file, directory, or folder. The returned value is a sum of its enumeration values. To determine which attributes are set, use the **And** operator to perform a bitwise comparison between the value returned by **GetAttr** and the value of the file attribute you want.

   > **Note** Because the application inspects only text files, it is reasonable to ignore some values possibly returned by **GetAttr**, such as **Directory**, **Hidden**, and **System**. It should be noted that this example code will be incorrect for files with such attributes.

2. Add the following code to the `btnExamine_Click` event after `NewName = CType(thisFile, String)` to enable a user to check the file's attributes.

```
' Check Attributes.
Dim MultiAttribute as FileAttribute
MultiAttribute = vbReadOnly Or vbArchive
If chkAttributes.Checked = True Then
    Attributes = GetAttr(thisFile)
    If (Attributes And MultiAttribute) = MultiAttribute Then
        AttributeResult = "ReadOnly and Archive"
    ElseIf (Attributes and vbReadOnly) = vbReadOnly Then
        AttributeResult = "ReadOnly"
    ElseIf (Attributes and vbArchive) = vbArchive Then
        AttributeResult = "Archive"
    Else
        AttributeResult = "Normal"
    End If
End If
```

3. Add the following code to the end of the `btnExamine_Click` event.

```
' Check last access time.
If chkLastAccess.Checked = True Then
     LastAccess = FileDateTime(NewName)
End If
```

   The **FileDateTime** function determines the last time the file was accessed. The returned **Date** value indicates the date and time when the file was created or last modified.

4. Add the following code to end of the `btnExamine_Click` event.

```
' Check Length.
If chkFileLength.Checked = True Then
     Length = FileLen(NewName)
End If
```

The **FileLen** function, which determines the length of the file, returns a long value specifying the length of the file in bytes.

5. Add the following code to the end of the `btnExamine_Click` event to enable a user to get the first line of the file.

```
' Check the first line.
If chkFirstLine.Checked = True Then
     Try
          FileOpen(1, NewName, OpenMode.Input)
          FirstLine = (InputString(1, 80))
     Catch ex As Exception
          ErrorMessage = "Something is wrong with the file you " & _
          "are trying to examine. Make sure it is a text " & _
          " file that is not empty or less than 80 characters."
          MsgBox (ErrorMessage)
          Exit Sub
     Finally
          FileClose(1)
     End Try

End If
```

Displaying the Results

To complete the application's functionality, a **MsgBox** reports the gathered information.

To display the results

1. At the beginning of the `btnExamine_Click` event, declare and initialize the following variables after `Dim stringAttributes As String`:

```
stringAttributes = "The attributes of the file are: "
Dim stringLastAccess As String
stringLastAccess = "The file was last accessed on: "
Dim stringLength As String
stringLength = "The file's length, in bytes, is: "
Dim strFirstLine As String
strFirstLine = "The first 80 characters are: "
```

2. At the end of the `If` statement that determines whether or not the `chkAttributes` CheckBox has been checked, add the following before the final `End If`.

```
' Add this to the results messagebox.
FinalString = stringAttributes & AttributeResult & "." & vbCrLf
```

3. At the end of the `If` statement that determines whether or not the `chkLastAccess` CheckBox has been checked, add the following before the final `End If`.

```
' Add to the messagebox.
FinalString = FinalString & stringLastAccess & LastAccess & "." _
& vbCrLf
```

4. At the end of the `If` statement that determines whether or not the `chkFileLength` CheckBox has been checked, add the

following before the final `End If`.

```
' Add to the messagebox.
FinalString = FinalString & stringLength & CStr(Length) & "." _
& vbCrLf
```

5. At the end of the `If` statement that determines whether or not the `chkFirstLine` CheckBox has been checked, add the following before the final `End If`.

```
' Add to the messagebox.
FinalString = FinalString & strFirstLine & Firstline & vbCrLf
```

6. At the very end of the function, add the following before the final `End Sub`.

```
If FinalString = "" Then
     MsgBox("No file attribute checkboxes selected.")
Else
     MsgBox(FinalString)
End If
```

Saving the Results

A user may want to save the results of an examination of a file. Accordingly, add code that checks to see if a log file exists, creates one if necessary, and then writes the results to the log file.

To create a log file

- Add the following to the end of the `btnExamine_Click` event.

```
' Check to see if results should be saved.
If chkSave.Checked = True and finalstring <> "" Then
     ' FileOpen will create the file if it doesn't exist.
     FileOpen(1, "log.txt", OpenMode.Append)
     Writeline (1, finalString)
     FileClose(1)
End If
```

To Test Your Application

1. In the directory of your choice, create a text file named test.txt with the following first line: "This is the first line of the first file. The FileExplorer application examines only text files." In the same directory, create a second text file named test2.txt with the following first line: "This is the first line of the second file. The FileExplorer application examines only text files." The FileExplorer requires a text file whose length is at least 80 characters.
2. Start the application.
3. Type an invalid path and click **Submit**.

   The following message appears: "You must enter a valid path. If trying to access a different drive, remember to include the drive letter."

4. Type the path to the directory that stores test.txt and click **Submit**.

   The `lstFilePick` ComboBox displays the text files.

5. Select test.txt in the `lstFilePick` ComboBox. Make sure all of the checkboxes are selected, and click **Examine**.

   The results form includes the file attributes, the last access date, the length, and the first 80 characters of the file.

6. Select test2.txt in the `lstFilePick` ComboBox, deselect all checkboxes and click **Examine**.

   The following error message appears: "No file attribute checkboxes selected."

7. Select **Attributes** and **Save Results** and click **Examine**.

The results form displays only the file's attributes.

8. Close **FileExplorer**.

   Since you checked the **Save Results** option, FileExplorer generated a log file named log.txt in the same directory as the text files.

To check the log

- In the current directory, open log.txt and confirm that FileExplorer recorded the correct information.

See Also

File Access with Visual Basic Run-Time Functions | Accessing Files with FileSystemObject | File Access Through the FileStream Class | System.IO Namespace | File Access Types and Functions | Visual Basic Language Walkthroughs | Choosing Among File I/O Options in Visual Basic .NET

Accessing Files with FileSystemObject

The File System Object (FSO) model provides an object-based tool for working with folders and files. It allows you to use the familiar *object.method* syntax with a rich set of properties, methods, and events to process folders and files. You can also employ the traditional Visual Basic statements and commands.

The FSO model gives your application the ability to create, alter, move, and delete folders, or to determine if and where particular folders exist. It also enables you to get information about folders, such as their names and the date they were created or last modified.

The FSO model makes processing files much easier as well. When processing files, your primary goal is to store data in an efficient, easy-to-access format. You need to be able to create files, insert and change the data, and output (read) the data. Although you can store data in a database, doing so adds a significant amount of overhead to your application. You may not want to have such overhead, or your data access requirements may not call for the extra functionality associated with a full-featured database. In this case, storing your data in a text file or binary file is the most efficient solution.

The FSO model, contained in the Scripting type library (Scrrun.dll), supports the creation and manipulation of text files through the **TextStream** object; however, the FSO model does not support binary files. To manipulate binary files, use the FileOpen Function with the **Binary** keyword.

The following objects make up the FSO model:

Object	Description
FileSystemObject	Allows you to create, delete, gain information about, and generally manipulate drives, folders, and files. Many of the methods associated with this object duplicate those in the other objects.
Drive	Allows you to gather information about a drive attached to the system, such as how much room is available and what its share name is. Note that a "drive" under the FSO model isn't necessarily a h ard disk: it can be a CD-ROM drive, a RAM disk, and so forth. A drive also isn't required to be phys ically attached to the system; it can also be logically connected through a local area network (LAN) .
Folder	Allows you to create, delete, or move folders, as well as query the system about their names, paths , and other information.
File	Allows you to create, delete, or move files, as well as query the system about their names, paths, a nd other information.
TextStream	Enables you to read and write text files.

For information about the various properties, methods, and events in the FSO model, use the Object Browser in Visual Basic by pressing CTRL+ALT+J and looking at the Scripting type library. If the Scripting type library does not appear in the list, create a reference to it as shown in the next section.

Programming in the FSO Model

Programming in the FSO model entails three main tasks:

- Creating a **FileSystemObject** object by using the **CreateObject** method, or by dimensioning a variable as a **FileSystemObject** object.
- Using the appropriate method on the newly created object.
- Accessing the object's properties.

The FSO model is contained in the Scripting type library, which is located in the file Scrrun.dll. If you don't already have a reference to it, you can create one.

To create a reference to the Scripting type library (Scrrun.dll)

1. On the **Project** menu, click **Add Reference**, and then click the **COM** tab.
2. Choose **Microsoft Scripting Runtime** from the **Component Name** list, and then click **Select**.

   You can now use the Object Browser to view the FSO model's objects, collections, properties, methods, events, and constants.

To create a FileSystemObject object

- Dimension a variable as type **FileSystemObject**, as in the following code:

```
Dim fso As New FileSystemObject
```

- or -

- Use the **CreateObject** method to create a **FileSystemObject** object, as in the following code:

```
fso = CreateObject("Scripting.FileSystemObject")
```

In the second example, `Scripting` is the name of the type library, and **FileSystemObject** is the name of the object of which you want to create an instance.

> **Note** The first method works only in Visual Basic, while the second method works either in Visual Basic or VBScript.

> **Security Note** Do not make decisions about the contents of a file based on the file name extension. For example, a file named Form1.vb may not be a Visual Basic .NET source file.

FSO Methods

The following table shows FSO methods and the tasks that they perform:

Task	Command
Create a new object	**CreateFolder** or **CreateTextFile**
Delete a file or folder	**DeleteFile** or **File.Delete**; **DeleteFolder** or **Folder.Delete**
Copy an object	**CopyFile** or **File.Copy**; **CopyFolder** or **Folder.Copy**
Move an object	**MoveFile** or **File.Move**; **MoveFolder** or **Folder.Move**
Access an existing drive, folder, or file	**GetDrive**, **GetFolder**, or **GetFile**

> **Note** The FSO model does not support the creation or deletion of drives.

As you can see, some functionality in the **FileSystemObject** object model is redundant. For example, you can copy a file using either the **CopyFile** method of the **FileSystemObject** object, or you can use the **Copy** method of the **File** object. The methods work the same way; both versions are included to give you maximum programming flexibility.

Note, however, that you don't need to use the **Get** methods for a newly created object, since the **Create** functions already return a handle to it. For example, if you create a new folder using the **CreateFolder** method, you don't need to use the **GetFolder** method to access its properties, such as **Name**, **Path**, or **Size**. Just set a variable to the **CreateFolder** function to gain a handle to the newly created folder, and then access its properties, methods, and events.

Drive Information

The **Drive** object allows you to get information about the various drives attached to a system, either physically or over a network. Its properties contain the following information:

Property	Description
TotalSize	Total size of the drive, in bytes
AvailableSpace, **FreeSpace**	How much space is available on the drive, in bytes
DriveLetter	Letter assigned to the drive
DriveType	Type of drive (removable, fixed, network, CD-ROM, or RAM disk)
SerialNumber	Drive's serial number
FileSystem	Type of file system the drive uses (FAT, FAT32, or NTFS)
IsReady	Whether a drive is available for use
ShareName, **VolumeName**	Name of the share and/or volume
Path, **RootFolder**	Path or root folder of the drive

App.Path, ChDir, ChDrive, and CurDir

If you use the **Path** property (**App.Path**), the **ChDrive** and **ChDir** statements, or the **CurDir** function, be aware that they may return a Universal Naming Convention (UNC) path (that is, one beginning with *Server**Share*...) rather than a drive path (such as e:*Folder*), depending on how you run your program or project.

App.Path returns a UNC path when:

- You run a project after loading it from a network share, even if the network share is mapped to a drive letter.
- You run a compiled executable file from a network share (but only if it is run using a UNC path).

ChDrive cannot handle UNC paths, and it raises an error when **App.Path** returns one. You can handle this error by adding `On Error Resume Next` before the **ChDrive** statement, or by testing the first two characters of **App.Path** to see if they are backslashes.

The command prompt always has a drive path for the current directory, so **CurDir** is set to a drive path. **ChDir** does not raise an error, but it fails to change the directory from a drive path to a UNC path. The only workaround for this situation is to locate a local drive that is mapped to the share specified in the UNC path, or to use network commands to create such a mapping.

If the project is loaded into the Visual Basic development environment from a network share — either a UNC path or a mapped drive path — **App.Path** returns a UNC path when the project is run and **ChDrive** fails and raises an error. **ChDir** does not raise an error, but it fails to change the directory. The only workaround for this situation is to manually set the drive and directory.

If more than one person can open the project on the network share, you can use an MS-DOS environment variable to give each person a customized mapping for the share.

Folder Information

The following table shows the methods for carrying out common folder tasks:

Method	Task
FileSystemObject.CreateFolder	Create a folder.
Folder.Delete or **FileSystemObject.DeleteFolder**	Delete a folder.
Folder.Move or **FileSystemObject.MoveFolder**	Move a folder.
Folder.Copy or **FileSystemObject.CopyFolder**	Copy a folder.
Folder.Name	Retrieve the name of a folder.
FileSystemObject.FolderExists	Find out if a folder exists on a drive.
FileSystemObject.GetFolder	Get an instance of an existing **Folder** object.
FileSystemObject.GetParentFolderName	Find out the name of a folder's parent folder.
FileSystemObject.GetSpecialFolder	Find out the path of a system folder.

Sequential Text Files and FSO

Sequential text files (sometimes referred to as a *text stream*) are useful when you want to read the contents of a file within a **FileSystemObject** object. You can access sequential text files with these methods:

Method	Task
CreateTextFile, **OpenTextFile**, or **OpenAsTextStream**	Create a sequential text file.
Write or **WriteLine**	Add data to a text file.
Read, **ReadLine**, **or ReadAll**	Read data from a text file.
File.Move or **MoveFile**	Move a file.
File.Copy or **CopyFile**	Copy a file.
File.Delete or **DeleteFile**	Delete a file.

See Also

File Access with Visual Basic Run-Time Functions | File Access with Visual Basic .NET

Exception Handling

Visual Basic supports structured exception (error) handling, which allows the program to detect and possibly recover from errors during execution. Visual Basic uses an enhanced version of the **Try...Catch...Finally** syntax already supported by other languages such as C++. Structured exception handling combines a modern control structure (similar to **Select Case** or **While**) with exceptions, protected blocks of code, and filters.

Structured exception handling, which is the recommended method of error handling in Visual Basic, makes it easy to create and maintain programs with robust, comprehensive error handlers. Unstructured exception handling using **On Error** can degrade application performance and result in code that is difficult to debug and maintain.

In This Section

Introduction to Exception Handling
Summarizes how to handle exceptions in your programs.
Types of Errors
Provides an overview of syntax errors, run-time errors, and logic errors.
Structured Exception Handling
Discusses structured exception handling in Visual Basic .NET, with code examples.
Unstructured Exception Handling
Discusses unstructured exception handling in Visual Basic .NET, with code examples.

Related Sections

Language Changes in Visual Basic
Summarizes language element changes in Visual Basic .NET.

Introduction to Exception Handling

Visual Basic supports both *structured* and *unstructured* exception (error) handling. By placing specific code in your application, you can handle most of the errors users may encounter and enable the application to continue running. Structured and unstructured error handling allow you to plan for potential errors, preventing them from interfering with the intended purpose of the application.

Consider using exception handling in any method that uses operators that may generate an exception, or that calls into or accesses other procedures that may generate an exception.

If an exception occurs in a method that is not equipped to handle it, the exception is propagated back to the calling method , or the previous method. If the previous method also has no exception handler, the exception is propagated back to that method 's caller, and so on. The search for a handler continues up the call stack, which is the series of procedures called within the application. If it fails to find a handler for the exception, an error message is displayed and the application is terminated.

> **Note** A single method can contain either structured or unstructured exception handling, but not both.

Structured Exception Handling

In structured exception handling, blocks of code are encapsulated, with each block having one or more associated handlers. Each handler specifies some form of filter condition on the type of exception it handles. When an exception is raised by code in a protected block, the set of corresponding handlers is searched in order, and the first one with a matching filter condition is executed. A single method can have multiple structured exception handling blocks, and the blocks can also be nested within each other.

The **Try...Catch...Finally** statement is used specifically for structured exception handling. For more information, see Structured Exception Handling.

Unstructured Exception Handling

The **On Error** statement is used specifically for unstructured exception handling. In unstructured exception handling, **On Error** is placed at the beginning of a block of code. It then has "scope" over that block; it handles any errors occurring within the block. If the program encounters another **On Error** statement, that statement becomes valid and the first statement becomes invalid.

See Also

Types of Errors | Structured Exception Handling | Unstructured Exception Handling | On Error Statement | Try...Catch...Finally Statements

Choosing When to Use Structured and Unstructured Exception Handling

Structured exception handling is simply that — using a control structure containing exceptions, isolated blocks of code, and filters to create an exception handling mechanism. This allows your code to differentiate between different types of errors and react in accordance with circumstances. In unstructured exception handling, an **On Error** statement at the beginning of the code handles all exceptions.

Structured exception handling is significantly more versatile, robust, and flexible than unstructured. If possible, use structured exception handling. However, you might use unstructured exception handling under these circumstances:

- You are upgrading an application written in an earlier version of Visual Basic.
- You are developing a preliminary or draft version of an application and you don't mind if the program fails to shut down gracefully.
- You know in advance exactly what will cause the exception.
- A deadline is pressing and you need to take shortcuts.
- Code is trivial or so short that you only need to test the branch of code generating the exception.
- You need to use the **Resume Next** statement, which is not supported in structured exception handling.

You cannot combine structured and unstructured exception handling in the same function. If you use an **On Error** statement, you cannot use a **Try...Catch** statement in the same function.

Regardless of which you choose to handle exceptions within your code, you must take a step back and examine what assumptions that code makes. For example, when your application asks the user to input a telephone number, the following assumptions come into play:

- The user will input a number rather than characters.
- The number will have a certain format.
- The user will not input a null string.
- The user has a single telephone number.

User input might violate any or all of these assumptions. Robust code requires adequate exception handling, which allows your application to recover gracefully from such a violation.

Unless you can guarantee that a method will never throw an exception under any circumstances, allow for informative exception handling. Exception handling should be meaningful. Beyond stating that something went wrong, messages resulting from exception handling should indicate why and where it went wrong. An uninformative message along the lines of "An error has occurred" only frustrates the user.

See Also

Types of Errors | Structured Exception Handling | Unstructured Exception Handling

Types of Errors

In Visual Basic, errors (also called *exceptions*) fall into one of three categories: syntax errors, run-time errors, and logic errors.

Syntax Errors

Syntax errors are those that appear while you write your code. Visual Basic checks your code as you type and alerts you if you make a mistake, such as misspelling a word or using a language element improperly. Syntax errors are the most common type of errors. You can fix them easily in the coding environment as soon as they occur.

> **Note** The **Option Explicit** statement is one means of avoiding syntax errors. It forces the programmer to declare, in advance, all the variables to be used in the application. Therefore, when those variables are used in the code, any typographic errors are caught immediately and can be fixed.

Run-Time Errors

Run-time errors are those that appear only after you compile and run your code. These involve code that may appear to be correct in that it has no syntax errors, but that will not execute. For example, you might correctly write a line of code to open a file. But if the file is corrupted, the application cannot carry out the **Open** function, and it stops running. You can fix most run-time errors by rewriting the faulty code, and then recompiling and rerunning it.

Logic Errors

Logic errors are those that appear once the application is in use. They most often take the form of unwanted or unexpected results in response to user actions. For example, a mistyped key or other outside influence might cause your application to stop working within expected parameters, or altogether. Logic errors are generally the hardest type to fix, since it is not always clear where they originate.

See Also

Introduction to Exception Handling | Structured Exception Handling | Unstructured Exception Handling | On Error Statement | Option Explicit Statement | Try...Catch...Finally Statements

Structured Exception Handling

Visual Basic supports structured exception handling, which helps you create and maintain programs with robust, comprehensive error handlers. Structured exception handling is code designed to detect and respond to errors during execution by combining a control structure (similar to **Select Case** or **While**) with exceptions, protected blocks of code, and filters.

Using the **Try...Catch...Finally** statement, you can protect blocks of code that have the potential to raise errors. You can nest exception handlers, and the variables declared in each block will have local scope.

The Try...Catch...Finally Statement

The following code shows the structure of a **Try...Catch...Finally** statement:

```
Try
   ' Starts a structured exception handler.
   ' Place executable statements that may generate
   ' an exception in this block.
Catch [optional filters]
   ' This code runs if the statements listed in
   ' the Try block fail and the filter on the Catch statement is true.
[Additional Catch blocks]
Finally
   ' This code always runs immediately before
   ' the Try statement exits.
End Try
   ' Ends a structured exception handler.
```

The **Try** block of a **Try...Catch...Finally** exception handler contains the section of code you want your error handler to monitor. If an error occurs during execution of any of the code in this section, Visual Basic examines each **Catch** statement within the **Try...Catch...Finally** until it finds one whose condition matches that error. If one is found, control transfers to the first line of code in the **Catch** block. If no matching **Catch** statement is found, the search proceeds to the **Catch** statements of the outer **Try...Catch...Finally** block that contains the block in which the exception occurred. This process continues through the entire stack until a matching **Catch** block is found in the current procedure. If none is found, an error is produced.

The code in the **Finally** section always executes last, just before the error-handling block loses scope, regardless of whether the code in the **Catch** blocks has executed. Place cleanup code, such as that for closing files and releasing objects, in the **Finally** section.

Error Filtering in the Catch Block

Catch blocks allow three options for specific error filtering. In one, errors are filtered based on the class of the exception (in this case `ClassLoadException`), as shown in the following code:

```
Try
   ' "Try" block.
Catch e as ClassLoadException
   ' "Catch" block.
Finally
   ' "Finally" block.
End Try
```

If a `ClassLoadException` error occurs, the code within the specified **Catch** block is executed.

In the second error-filtering option, the **Catch** section can filter on any conditional expression. One common use of this form of **Catch** filter is to test for specific error numbers, as shown in the following code:

```
Try
   ' "Try" block.
Catch When ErrNum = 5 'Type mismatch.
   ' "Catch" block.
Finally
   ' "Finally" block.
End Try
```

When Visual Basic finds the matching error handler, it executes the code within that handler, and then passes control to the **Finally** block.

> **Note** When trying to find a **Catch** block to handle an exception, each block's handler is evaluated until a match is found. Since these handlers can be calls to functions, unexpected side effects may result; for example, such a call might change a public variable that is then used in the code of a different **Catch** block that ends up handling the exception.

As a third alternative, you can combine options one and two and use both for exception handling. Your **Catch** statements should move from most specific to least specific. A **Catch** block by itself will catch all exceptions derived from **System.Exception**, and therefore should always be placed as the last block before **Finally**.

Structured Exception Handler Example

The following example shows another simple error handler based on the **Try...Catch...Finally** statement:

```
Function GetStringsFromFile(ByVal FileName As String) As Collection
   Dim Strings As New Collection
   Dim Stream As System.IO.StreamReader = System.IO.File.OpenText(FileName)   'Open the file.

   Try
      While True   ' Loop terminates with an EndOfStreamException
                   ' error when end of stream is reached.
         Strings.Add(Stream.ReadLine())
       End While
   Catch eos As System.IO.EndOfStreamException
      ' No action is necessary; end of stream has been reached.
   Catch IOExcep As System.IO.IOException
      ' Some kind of error occurred. Report error and clear collection.
      MsgBox(IOExcep.Message)
      Strings = Nothing
   Finally
      Stream.Close()   ' Close the file.
   End Try

   Return Strings
End Function
```

The **Finally** block is always run, regardless of any actions occurring in preceding **Catch** blocks. You cannot use **Resume** or **Resume Next** in structured exception handling.

> **Note** Any exception other than the **IOException** class or the **EndOfStreamException** class is propagated back to the caller unhandled.

See Also

The Exception Class | Introduction to Exception Handling | Types of Errors | Unstructured Exception Handling | Try...Catch...Finally Statements | IOException Class | EndOfStreamException Class

The Exception Class

To facilitate the use of structured exception handling, Visual Basic provides the ability to separate standard code from exception handling code. Exception handling code accesses an instance of the **Exception** class, which allows you to retrieve information about any exception you encounter.

Whenever an exception is thrown, the global **Err** object is set, and a new instance of an **Exception** class is created.

The properties of the **Exception** class aid in identifying the code location, type, and cause of exceptions. For example, the **StackTrace** property lists the called methods that led up to the exception, helping you find where the error occurs in the code. The **Message** property returns a text message describing the error; you can alter it to make a cryptic message easier to understand. If you do not supply an error message text string, the default is used. **HelpLink** gets or sets a link to an associated help file. **Source** gets or sets a string containing the name of the object causing the error or the name of the assembly where the exception originated.

> **Note** Avoid writing code that tests for specific error messages, since messages can change between versions of the class that originally produced them. Instead, test for the Exception type or use the exception number from the global **Err** object.

See Also

Introduction to Exception Handling | Types of Errors | Structured Exception Handling | Unstructured Exception Handling | The Err Object | Exception Class | Source Property

Creating Your Own Exceptions

There are two defined subclasses of exceptions in the Exception base class: System.Exception and Application.Exception.

System.Exception is the class from which the .NET Framework derives the pre-defined common language runtime exception classes. It is thrown by the common language runtime when nonfatal errors occur. System.Exception does not provide information about the cause of the exception.

You can create your own application exception classes by inheriting them from the Application.Exception class. Follow good coding practices by ending the class name of your exception with the word "Exception" — for example, OutOfMoneyException or TooMuchRainException.

The following example defines an exception class and defines three constructors for it, each of which takes different parameters.

```
Imports System
Public Class GardenException
   Inherits System.ApplicationException
   Public Sub New()
   End Sub
   ' Creates a Sub New for the exception that allows you to set the
   ' message property when thrown.
   Public Sub New(Message As String)
      MyBase.New(Message)
   End Sub
   ' Creates a Sub New that can be used when you also want to include
   ' the inner exception.
   Public Sub New(Message As String, Inner As Exception)
      MyBase.New(Message)
   End Sub
End Class
```

Note When using remoting in combination with user-defined exceptions, you must ensure that the metadata for your user-defined exceptions is available to code executing remotely, including exceptions that occur across application domains.

See Also

Structured Exception Handling | SystemException Class | ApplicationException Class

Walkthrough: Structured Exception Handling

While you can still employ the **On Error** statement to handle exceptions in your code, Visual Basic .NET also supports structured exception handling, which enables you to create and maintain programs with comprehensive error handling. In structured exception handling, blocks of code test for specific circumstances and react accordingly.

This walkthrough uses the creation of a simple application to demonstrate how to add structured exception handling to a program. It explains how to use the **Try...Catch...Finally** statement to handle exceptions, and how to perform error filtering within **Catch** blocks.

> **Note** You cannot combine both structured and unstructured exception handling within a single procedure.

Creating the Application

The application in this walkthrough is a customer order form for a company that sells teddy bears. The user interface consists of the following controls: a **TextBox** for the customer's name, two **ComboBox** controls to select the color and size of the bear, and an **Order** button. When the user enters the requested information and clicks the **Order** button, the application displays a summary of the order.

To create the application

1. From the **File** menu, select **New**, and then select **Project**. The **New Project** dialog box appears.
2. In the **Project Types** window, select **Visual Basic Projects** if it is not already selected, and then choose **Windows Application** from the **Templates** window.
3. In the **Name** box, set the project name to `TeddyBearProject`. The project is added to Solution Explorer, and the Windows Forms Designer opens.
4. Add the controls in the following table to the form, and set their properties as specified.

Control	Properties	Property Values
Label	Name Text	customerLabel Bear Order Form
TextBox	Name Text	customerName Customer Name
Label	Name Text	bearColorLabel Available Colors
ComboBox	Name Items Text	BearColor Black, Brown, Spotted Bear Color
Label	Name Text	bearSizeLabel Available Sizes
ComboBox	Name Items Text	BearSize Small, Normal, Large Bear Size
Button	Name Text	Order Order

Adding Functionality

Now that the controls and their properties are added and set, you must place code behind them to make the application functional. The following code handles the **Click** event for the **Order** button and displays a message to the customer.

To add functionality to the form and its controls

- Add the following code to the **Order** button's **Click** event:

```
Private Sub order_Click(ByVal sender As System.Object, _
   ByVal e As System.EventArgs) Handles Order.Click
   Dim bearOrder As String
   bearOrder = _
     String.Format("You have ordered a {0} {1} bear.", _
     bearSize.SelectedItem, bearColor.SelectedItem)
   MessageBox.Show(bearOrder)
```

```
End Sub
```

The application is now ready for you to add structured exception handling code.

Adding a Try...Catch Block

To ensure that the customer has specified a color for the bear, add a **Try...Catch** statement. Keep the following considerations in mind:

- A **Catch** clause with no identifier catches all exceptions.
- A **Catch** clause with a **When** clause catches exceptions only when the expression evaluates to **True**; the type of the expression must be implicitly convertible to **Boolean**.

To add a simple Try...Catch block

- Add the following code to the **Order** button's **Click** event after the section that tests the size and color values, that is, after `String.Format(("You have ordered a {0} {1} bear.", BearSize.SelectedItem, BearColor.SelectedItem)`. This code throws an exception if it encounters an invalid color value.

```
Try
   If ((BearColor.SelectedIndex < 0) or _
         (BearColor.SelectedIndex > 2)) Then
      Throw New System.Exception()
   End If
' The Catch statement handles errors caused by a lack of bear color.
Catch ex As Exception When BearColor.SelectedIndex < 0
   bearOrder = String.Format("You must select a bear color!")
Finally
   Beep () ' Beep at the end.
End Try
```

To add an additional Catch clause

1. Add a new item, "Purple", to the `BearColor` **ComboBox** control.
2. 8Add the following code after the code line `bearOrder = String.Format("You must select a bear color!")`:

```
Catch ex As Exception When BearColor.SelectedIndex = 3
   bearOrder = String.Format("There are no bears of that color.")
```

You can add as many **Catch** clauses as necessary to your code.

Testing

You can now test the application to make sure it works correctly.

To build and run the application

1. From the **Build** menu, select **Build TeddyBearProject**.
2. Press F5 to run the application. The main form appears.

To test the application

1. Enter a name in the **Customer name** text box, and then select a color and size for the bear from the **Available Colors** and **Available Sizes** combo boxes.
2. Click the **Order** button. If you specified either **Black**, **Brown**, or **Spotted** as the color in the previous step, a message appears stating the size and color of the bear you ordered.

   If you do not specify a color, a message appears prompting you to do so.

3. Click **OK** to cancel the message.
4. In the **Available Colors** text box, select **Purple**, and then click the **Order** button. A message appears stating that purple bears are not available.

See Also

On Error Statement | Try...Catch...Finally Statements | Structured Exception Handling | Throw Statement | Throwing Exceptions from Components

Unstructured Exception Handling

In *unstructured exception handling*, you place an **On Error** statement at the beginning of a block of code, and it handles any errors occurring within that block. When an exception is raised in a procedure after the **On Error** statement executes, the program branches to the line argument specified in the **On Error** statement. The line argument, which is a line number or line label, indicates the exception handler location.

Sometimes a call is made from the original procedure to another procedure, and an exception occurs in the called procedure. In such cases, if the called procedure does not handle the exception, the exception propagates back to the calling procedure, and execution branches to the line argument.

> **Note** Unstructured error handling using **On Error** can degrade application performance and result in code that is difficult to debug and maintain. Structured error handling is the recommended method.

On Error GoTo Line

The **On Error GoTo Line** statement assumes that error-handling code starts at the line specified in the required *line* argument. If a run-time error occurs, control branches to the line label or line number specified in the argument, activating the error handler. The specified line must be in the same procedure as the **On Error GoTo Line** statement; otherwise, Visual Basic generates a compiler error. The following example illustrates the use of an error handler with a line label:

```
Sub TestSub
   On Error GoTo ErrorHandler
      ' Code that may or may not contain errors.
   Exit Sub

   ErrorHandler:
      ' Code that handles errors.
      Resume
End Sub
```

The example contains an error handler named `ErrorHandler`. If any code in the `TestSub` subroutine generates an error, Visual Basic immediately executes the code that follows the `ErrorHandler` label. At the end of the error-handling block, the **Resume** statement passes control back to the line of code where the error first occurred. The rest of the subroutine then continues executing as if the error did not occur.

> **Note** You must place an **Exit Sub** statement immediately before the error-handling block. Otherwise, Visual Basic runs the error-handling code when it reaches the end of the subroutine, causing unwanted or unexpected results.

On Error Resume Next

The **On Error Resume Next** statement specifies that in the event of a run-time error, control passes to the statement immediately following the one in which the error occurred. At that point, execution continues. **On Error Resume Next** enables you to put error-handling routines where errors will occur, rather than transferring control to another location in the procedure.

> **Note** If your procedure calls another procedure, the **On Error Resume Next** statement becomes inactive during the execution of the called procedure. Therefore, you should place an **On Error Resume Next** statement in each called procedure that needs one. This is necessary because the **Resume Next** behavior applies only to the procedure containing the **On Error Resume Next** statement. If an unhandled error occurs in a called procedure, the exception propagates back to the calling procedure, and execution resumes on the statement following the call. In such cases, the error is not handled.

Resume also can be used on its own, outside the **On Error** statement. When **Resume** is used this way, Visual Basic returns control to the statement that caused the error. You generally use **Resume** after an error handler corrects the error.

Visual Basic also provides the **Resume Next** statement, which directs control to the line immediately following the line of code that caused the error. You might use **Resume Next** for cases in which an error will not cause your application to stop working. You might also use it if an error will not change the expected results of your subroutine.

Another variation on the **Resume** statement is **Resume Line**, which is similar to **On Error GoTo Line**. **Resume Line** passes control to a line you specify in the *line* argument. You can use **Resume Line** only within an error handler.

> **Note** When debugging your code, you must disable the **On Error Resume Next** statement.

On Error GoTo 0

The **On Error GoTo 0** statement disables any error handler in the current procedure. If you do not include an **On Error GoTo 0** statement, the error handler is still disabled when the procedure containing the exception handler ends.

> **Note** The **On Error GoTo 0** statement is not meant to specify line 0 as the start of the error-handling code, even if the procedure contains a line numbered 0.

On Error GoTo -1

The **On Error GoTo -1** statement disables any exception handlers in the current procedure. If you do not include an **On Error GoTo -1** statement, the exception is automatically disabled when its procedure ends.

> **Note** The **On Error GoTo -1** statement is not meant to specify line -1 as the start of the error-handling code, even if the procedure contains a line numbered -1.

Unstructured Exception Handler Example

In the following code, the exception handler is named `DivideByZero` and handles a specific error — that of attempting to divide by zero. If a different error occurs, Visual Basic raises a run-time error and stops the application.

```
Sub ErrorTest ()
' Declare variables.
   Dim x As Integer, y As Integer, z As Integer
   ' The exception handler is named "DivideByZero".
   On Error GoTo DivideByZero
   ' The main part of the code, which might cause an error.
   x = 2
   y = 0
   z = x \ y

   ' This line disables the exception handler.
   On Error GoTo 0
      Console.WriteLine(x & "/" & y & " = " & z)

   ' Exit the subroutine before the error-handling code.
   ' Failure to do so can create unexpected results.
   Exit Sub

   ' This is the exception handler, which deals with the error.
   DivideByZero:
   ' Include a friendly message to let the user know what is happening.
   Console.WriteLine("You have attempted to divide by zero!")

   ' Provide a solution to the error.
   y = 2

   ' The Resume statement returns to the point at which
   ' the error first occurred, so the application
   ' can continue to run.
   Resume

End Sub
```

See Also

The Err Object | Introduction to Exception Handling | Types of Errors | Structured Exception Handling | End Statement | Err Object (Language Reference) | Exit Statement | LastDllError Property | On Error Statement | Resume Statement

The Err Object

To write error-handling code, you need to know which error occurred. You can find this out by using the **Err** Object.

The **Err** object contains properties whose values are determined by the error that just occurred. The **Number** property contains the cause of the error, and the **Description** property contains a text message that describes the error in detail. **Helpfile** and **HelpContext** allow you to display a related help file when a user presses the **Help** button or the F1 key. **LastDLLError** displays the last DLL called, and whether or not the call was successful. **Source** specifies a string expression representing the object or application that generated the error.

Your error handler should test or save the relevant property values before any other error can occur. Your code can handle only a single error at a time, so you must deal with one before moving on to another.

> **Note** The **Err** Object can be used only with exceptions caught with the **On Error GoTo** statement.

Application-Specific Errors

In addition to handling errors generated by Visual Basic, you might also want to handle errors specific to your application.

If you create an object that accesses other objects, you should write code to deal with any unhandled errors the other objects pass back. If necessary, you can map the error codes in **Err.Number** to one of your own application-specific errors, and then pass it back to the caller of your object.

See Also

Introduction to Exception Handling | Types of Errors | Structured Exception Handling | Unstructured Exception Handling | The Exception Class | Err Object | On Error Statement | Description Property | Number Property | HelpFile Property | HelpContext Property | LastDLLError Property | Source Property

Reading from and Writing to the Registry

When programming in Visual Basic .NET, you can choose to access the registry via either the functions provided by Visual Basic .NET or the registry classes of the .NET Framework. The registry hosts information from the operating system as well as information from applications hosted on the machine. Working with the registry may compromise security by allowing inappropriate access to system resources or protected information.

In This Section

Reading from and Writing to the Registry Using the Microsoft.Win32 Namespace
: Explains how to use the **Registry** and **RegistryKey** classes in the Microsoft.Win32 namespace of the .NET Framework to read from and write to a registry key.

Reading from and Writing to the Registry Using Visual Basic Run-Time Functions
: Explains how to use the Visual Basic .NET functions, **DeleteSetting**, **GetSetting**, **GetAllSettings**, and **SaveSetting**, to access the registry.

Related Sections

Registry Access Changes in Visual Basic .NET
: Provides an explanation of differences between registry access in Visual Basic 6.0 and Visual Basic .NET.

Registry Class
: Presents an overview of the Registry class along with links to individual keys and members.

Reading from and Writing to the Registry Using the Microsoft.Win32 Namespace

Because you can use the GetSetting and **SaveSetting** functions to access keys only under **HKEY_CURRENT_USER\Software\VB and VBA Program Settings**, you may find them limiting. When that happens, you can use the **Registry** and **RegistryKey** classes in the Microsoft.Win32 namespace of the .NET Framework. For more information, see Registry Class and RegistryKey Class.

> **Security Note** It is more secure to write data to the current user (Microsoft.Win32.Registry.CurrentUser) than to the local computer (Microsoft.Win32.Registry.LocalMachine). A condition, that's typically referred to as "squatting," occurs when the key you are creating was previously created by another, possibly malicious, process. To prevent this from occurring, use the **RegistryKey.GetValue** method. It returns **Nothing** if the key does not already exist.

The **Registry** class supplies the base registry keys that can be used to access subkeys and their values. The base keys themselves are read-only. The following table lists and describes the seven keys exposed by the **Registry** class.

Key	Description
ClassesRoot	Defines the types of documents and the properties associated with those types.
CurrentConfig	Contains hardware configuration information that is not user-specific.
CurrentUser	Contains information about the current user preferences, such as environmental variables.
DynData	Contains dynamic registry data, such as that used by Virtual Device Drivers.
LocalMachine	Contains five subkeys (Hardware, SAM, Security, Software, and System) that hold the configuration data for the local machine.
PerformanceData	Contains performance information for software components.
Users	Contains information about the default user preferences.

The following example shows how to read a DWORD value from HKEY_CURRENT_USER:

```
Imports Microsoft.Win32
Dim regVersion As RegistryKey
Dim keyValue As String
keyValue = "Software\\Microsoft\\TestApp\\1.0"
regVersion = Registry.CurrentUser.OpenSubKey(keyValue, False)
Dim intVersion As Integer = 0
If (Not regVersion Is Nothing) Then
    intVersion = regVersion.GetValue("Version", 0)
    regVersion.Close()
End If
```

The following example reads, increments, and then writes a DWORD value to HKEY_CURRENT_USER:

```
Imports Microsoft.Win32
Dim regVersion As RegistryKey
regVersion =
Registry.CurrentUser.OpenSubKey("SOFTWARE\\Microsoft\\TestApp\\1.0", True)
If regVersion Is Nothing Then
    ' Key doesn't exist; create it.
    regVersion =
Registry.CurrentUser.CreateSubKey("SOFTWARE\\Microsoft\\TestApp\\1.0")
End If

Dim intVersion As Integer = 0
If (Not regVersion Is Nothing) Then
    intVersion = regVersion.GetValue("Version", 0)
    intVersion = intVersion + 1
    regVersion.SetValue("Version", intVersion)
    regVersion.Close()
End If
```

See Also

Structured Exception Handling | SystemException Class | ApplicationException Class

Reading from and Writing to the Registry Using Visual Basic Run-Time Functions

Visual Basic .NET offers four functions to access the registry. To use them, you must have Read and Write permission from the RegistryPermissionAccess Enumeration enumeration. Any code running with full trust (under the default security policy, this is any code installed on the user's local hard drive) has the necessary permissions to access the registry. For more information, see RegistryPermission Class.

Visual Basic .NET Function	Description
DeleteSetting	Deletes a section or key setting from an application's entry in th e registry.
GetSetting	Returns a key setting value from an application's entry in the re gistry.
GetAllSettings	Returns a list of key settings and their values from an applicatio n's entry in the registry.
SaveSetting	Creates or saves an application entry in the registry.

Note Section names cannot be returned via the **GetSetting** function.

If the key setting cannot be saved, an **ArgumentException** object is thrown. For more information on this exception, see ArgumentException Class.

The following example creates a registry key and two subkeys. It prints the first key's value and then prints the values of the first key as well as its subkeys. It then deletes the second subkey and prints the values of the first key and its subkeys to confirm the deletion of the second subkey.

```
' Create the first key.
SaveSetting("TestApp","Startup","FirstKey","First")
' Create the first subkey.
SaveSetting("TestApp","FirstKey","FirstSubKey","FirstSub")
' Create the second subkey.
SaveSetting("TestApp","FirstKey","SecondSubKey","SecondSub")
Try
   ' Write the first key's value.
   Console.WriteLine ((GetSetting("TestApp","Startup","FirstKey"))
   ' Write the first key as well as its two subkeys.
   Console.WriteLine (GetAllSettings("TestApp","Startup"))
   Catch e As ArgumentException
   Catch e As Exception
   Console.WriteLine (e.GetType.ToString)
   Finally
End Try
DeleteSetting("TestApp","FirstKey","SecondSubKey")
Try
   Console.WriteLine (GetSetting("TestApp","Startup","FirstKey"))
   Console.WriteLine (GetAllSettings("TestApp","Startup"))
   Catch e As ArgumentException
   Catch e As Exception
   Console.WriteLine (e.GetType.ToString)
   Finally
End Try
```

There is a limitation to using the built-in registry access functions: You can access only the registry keys under **HKEY_CURRENT_USER\Software\VB and VBA Program Settings**. To do so, you must be logged onto the system because the **HKEY_CURRENT_USER** registry key is not active until you log on to the system.

Registry settings that are to be accessed from a noninteractive process (such as mtx.exe) should be stored under either the **HKEY_LOCAL_MACHINE\Software** or the **HKEY_USER\DEFAULT\Software** registry keys.

See Also

Reading from and Writing to the Registry Using the Microsoft.Win32 Namespace | DeleteSetting Function | GetSetting Function |

GetAllSettings Function | SaveSetting Function | Code: Determining Whether a Registry Key Exists (Visual Basic) | Code: Adding Data to the Registry (Visual Basic) | Code: Creating a Registry Key and Setting Its Values (Visual Basic) | Code: Deleting Data from the Registry (Visual Basic) | Code: Deleting a Registry Key (Visual Basic) | Code: Reading Data from the Registry (Visual Basic)

Visual Basic Language and Run-Time Reference

In This Section

Typographic and Code Conventions
Summarizes the way that keywords, placeholders and other elements of the language are formatted in the Visual Basic documentation.
Visual Basic Language Keywords
Lists the keywords of the Visual Basic language.
Visual Basic Run-Time Library Members
Lists the functions, methods, and properties you can use in your code, grouped by functionality.
Keywords and Members by Task
Summarizes Visual Basic language keywords and run-time library members by purpose and use.
Attributes
Documents the attributes available in Visual Basic .NET.
Constants and Enumerations
Documents the constants and enumerations available in Visual Basic .NET.
Data Types
Documents the data types available in Visual Basic .NET.
Directives
Documents the directives available in Visual Basic .NET.
Functions
Documents the functions available in Visual Basic .NET.
Keywords
Documents the keywords available in Visual Basic .NET.
Methods
Documents the methods available in Visual Basic .NET.
Objects
Documents the objects available in Visual Basic .NET.
Operators
Documents the operators available in Visual Basic .NET.
Properties
Documents the properties available in Visual Basic .NET.
Statements
Documents the statements available in Visual Basic .NET.
Error Messages
Provides a brief overview of error messages and how to handle them in Visual Basic .NET.

Related Sections

Visual Basic Language
Provides comprehensive help on all areas of the Visual Basic language.
Visual Basic Compiler Options
Describes how to use the command-line compiler as an alternative to compiling programs from within the Visual Studio .NET integrated development environment (IDE).
Visual Basic .NET Language Specification
Provides a complete high-level description of the language.

Typographic and Code Conventions

Visual Basic documentation uses the following typographic and code conventions.

Typographic Conventions

Convention	Description
Sub, **If**, **ChDir**, **Print**, **True**, **Debug**	Words in bold with initial letter capitalized indicate language-specific keywords.
Setup	Words you are instructed to type appear in bold.
object, *varname*, *arglist*	Italic letters indicate placeholders for information you supply.
[*expressionlist*]	In syntax, items inside brackets are optional.
{ **Public** | **Private** }	In syntax, braces and a vertical bar indicate a mandatory choice between two or more items. You must choose one, and only one, of the items.
[**Protected** | **Friend**]	In syntax, a vertical bar separates alternative options. Within brackets, the alternatives are not exclusive. Within braces, the alternatives are exclusive.
[{ **ByRef** | **ByVal** }]	In syntax, brackets surrounding items within braces indicate that no more than one item may be included, and the items may also be omitted entirely.
$membname_1$, $membname_2$, $membname_3$	Subscripts differentiate multiple instances of the same placeholder.
$membname_1$ ... $membname_n$	In syntax, ellipses indicate an indefinite number of items of the kind immediately preceding the ellipses. In code, ellipses signify code omitted for the sake of clarity.
ESC, ENTER	Words in capital letters indicate key names and key sequences.
ALT+F1, CTRL+R	A plus sign (+) between key names indicates a combination of keys. For example, ALT+F1 means hold down the ALT key while pressing the F1 key.

Code Conventions

Sample code	Description
`MyString = "Hello, world!"`	This font is used for code, variables, and error message text.
`' This is a comment.`	An apostrophe (') introduces code comments.

`MyVar = "This is an " _`
`& "example" _`
`& " of how to continue code."`
	A space followed by an underscore (_) at the end of line continues a line of code.

See Also

Visual Basic Language and Run-Time Reference

Visual Basic Language Keywords

The following table contains a listing of the Visual Basic .NET language keywords. All language keywords are reserved.

AddHandler	AddressOf	Alias	And
AndAlso	Ansi	As	Assembly
Auto	Boolean	ByRef	Byte
ByVal	Call	Case	Catch
CBool	CByte	CChar	CDate
CDec	CDbl	Char	CInt
Class	CLng	CObj	Const
CShort	CSng	CStr	CType
Date	Decimal	Declare	Default
Delegate	Dim	DirectCast	Do
Double	Each	Else	ElseIf
End	Enum	Erase	Error
Event	Exit	False	Finally
For	Friend	Function	Get
GetType	**GoSub**	GoTo	Handles
If	Implements	Imports	In
Inherits	Integer	Interface	Is
Let	Lib	Like	Long
Loop	Me	Mod	Module
MustInherit	MustOverride	MyBase	MyClass
Namespace	New	Next	Not
Nothing	NotInheritable	NotOverridable	Object
On	Option	Optional	Or
OrElse	Overloads	Overridable	Overrides
ParamArray	Preserve	Private	Property
Protected	Public	RaiseEvent	ReadOnly
ReDim	REM	RemoveHandler	Resume
Return	Select	Set	Shadows
Shared	Short	Single	Static
Step	Stop	String	Structure
Sub	SyncLock	Then	Throw
To	True	Try	TypeOf
Unicode	Until	**Variant**	When
While	With	WithEvents	WriteOnly
Xor	#Const	#ExternalSource	#If...Then...#Else
#Region	-	&	&=
*	*=	/	/=
\	\=	^	^=
+	+=	=	-=

Note **GoSub**, **Let**, and **Variant** are retained as keywords, although they are no longer used in Visual Basic .NET.

See Also

Visual Basic Run-Time Library Members | Keywords and Members By Task

Visual Basic Run-Time Library Members

The **Microsoft.VisualBasic** namespace contains the classes and modules that constitute the Visual Basic .NET run-time library. The library provides functions, methods, and properties you can use in your code. Each module and class represents a particular category of functionality.

Microsoft.VisualBasic.Collection Class

Add	Count	GetEnumerator	Item
Remove			

Microsoft.VisualBasic.ComClassAttribute Class

ClassID	EventID	InterfaceID	InterfaceShadows

Microsoft.VisualBasic.Conversion Module

ErrorToString	Fix	Hex	Int
Oct	Str	Val	

Microsoft.VisualBasic.DateAndTime Module

DateAdd	DateDiff	DatePart	DateSerial
DateString	DateValue	Day	Hour
Minute	Month	MonthName	Now
Second	TimeOfDay	Timer	TimeSerial
TimeString	TimeValue	Today	WeekDay
WeekDayName	Year		

Microsoft.VisualBasic.ErrObject Class

Clear	Description	Erl	GetException
HelpContext	HelpFile	LastDLLError	Number
Raise	Source		

Microsoft.VisualBasic.FileSystem Module

ChDir	ChDrive	CurDir	Dir
EOF	FileAttr	FileClose	FileCopy
FileDateTime	FileGet	FileGetObject	FileLen
FileOpen	FilePut	FilePutObject	FileWidth
FreeFile	GetAttr	Input	InputString
Kill	LineInput	Loc	Lock
LOF	MkDir	Print	PrintLine
Rename	Reset	RmDir	Seek
SetAttr	SPC	TAB	Unlock
Write	WriteLine		

Microsoft.VisualBasic.Financial Module

DDB	FV	IPmt	IRR
MIRR	NPer	NPV	Pmt
PPmt	PV	Rate	SLN
SYD			

Microsoft.VisualBasic.Globals Module

ScriptEngine	ScriptEngineBuildVersion	ScriptEngineMajorVersion	ScriptEngineMinorVersion

Microsoft.VisualBasic.Information Module

Err	IsArray	IsDate	IsDBNull
IsError	IsNothing	IsNumeric	IsReference
LBound	QBColor	RGB	SystemTypeName
TypeName	UBound	VarType	VbTypeName

Microsoft.VisualBasic.Interaction Module

AppActivate	Beep	CallByName	Choose
Command	CreateObject	DeleteSetting	Environ
GetAllSettings	GetObject	GetSetting	IIf
InputBox	MsgBox	Partition	SaveSetting
Shell	Switch		

Microsoft.VisualBasic.Strings Module

Asc	AscW	Chr	ChrW
Filter	Format	FormatCurrency	FormatDateTime
FormatNumber	FormatPercent	GetChar	InStr
InStrRev	Join	LCase	Left
Len	LSet	LTrim	Mid
Replace	Right	RSet	RTrim
Space	Split	StrComp	StrConv
StrDup	StrReverse	Trim	UCase

Microsoft.VisualBasic.VBFixedArrayAttribute Class

Bounds	Length		

Microsoft.VisualBasic.VBFixedStringAttribute Class

Length			

Microsoft.VisualBasic.VbMath Module

Randomize	Rnd		

The Visual Basic .NET run-time library also provides constants and enumeration members you can use in your code. Each enumeration represents a particular category of functionality. See Constants and Enumerations for more information.

See Also

Visual Basic Runtime Namespace in Visual Studio | Constants and Enumerations | Visual Basic Language Keywords | Keywords and Members By Task

Keywords and Members by Task

Visual Basic language keywords and run-time library members are organized by purpose and use.

Category	Description
Arrays	Creating, defining, and using arrays.
Collections	Using collections.
Compiler Directives	Controlling compiler behavior.
Control Flow	Looping and controlling procedure flow.
Conversion	Converting numbers and data types.
Data Types	Data types and object subtypes.
Dates and Times	Converting and using date and time expressions.
Directories and Files	Controlling the file system and processing files.
Errors	Trapping and returning error values.
Financial	Performing financial calculations.
Input and Output	Receiving input and displaying or printing output.
Information and Interaction	Starting other applications and processing events.
Math	Performing trigonometric and other mathematical calculations.
Operators	Comparing expressions and other operations.
Registry	Deleting, reading, and saving program settings.
String Manipulation	Manipulating strings and string type data.
Declarations and Constants	Declaring programming elements and defining constants.

See Also

Visual Basic Language Keywords | Visual Basic Run-Time Library Members

Arrays Summary

Visual Basic language keywords and run-time library members are organized by purpose and use.

Action	Language element
Verify an array.	IsArray
Declare and initialize an array.	Dim, Private, Public, ReDim
Find the limits of an array.	LBound, UBound
Reinitialize an array	Erase, ReDim

See Also

Keywords and Members by Task | Visual Basic Language Keywords | Visual Basic Run-Time Library Members

Collection Object Summary

Visual Basic language keywords and run-time library members are organized by purpose and use.

Action	Language element
Create a **Collection** object.	Collection
Add an item to a collection.	Add
Remove an object from a collection.	Remove
Reference an item in a collection.	Item
Return a reference to an **IEnumerator** interface.	GetEnumerator

See Also

Keywords and Members by Task | Visual Basic Language Keywords | Visual Basic Run-Time Library Members

Compiler Directive Summary

Visual Basic language keywords and run-time library members are organized by purpose and use.

Action	Language element
Define a compiler constant.	#Const
Compile selected blocks of code.	#If...Then...#Else
Collapse and hide sections of code.	#Region
Indicate a mapping between source lines and text external to the source.	#ExternalSource

See Also

Keywords and Members by Task | Visual Basic Language Keywords | Visual Basic Run-Time Library Members

Control Flow Summary

Visual Basic language keywords and run-time library members are organized by purpose and use.

Action	Language element
Branch.	GoTo, On Error
Exit or pause the program.	End, Exit, Stop
Loop.	Do...Loop, For...Next, For Each...Next, While...End While, With
Make decisions.	Choose, If...Then...Else, Select Case, Switch
Use procedures.	Call, Function, Property, Sub

See Also

Keywords and Members by Task | Visual Basic Language Keywords | Visual Basic Run-Time Library Members

Conversion Summary

Visual Basic language keywords and run-time library members are organized by purpose and use.

Action	Language element
ANSI value to string.	Chr, ChrW
String to lowercase or uppercase.	Format, LCase, UCase
Date to serial number.	DateSerial, DateValue
Decimal number to other bases.	Hex, Oct
Number to string.	Format, Str
One data type to another.	CBool, CByte, CDate, CDbl, CDec, CInt, CLng, CSng, CShort, CStr, CType, Fix, Int
Date to day, month, weekday, or year .	Day, Month, Weekday, Year
Time to hour, minute, or second.	Hour, Minute, Second
String to ASCII value.	Asc, AscW
String to number.	Val
Time to serial number.	TimeSerial, TimeValue

See Also

Keywords and Members by Task | Visual Basic Language Keywords | Visual Basic Run-Time Library Members

Data Types Summary

Visual Basic language keywords and run-time library members are organized by purpose and use.

Action	Language element
Convert between data types.	CBool, CByte, CChar, CDate, CDbl, CDec, CInt, CLng, CObj, CShort, CSng, CStr, Fix, Int
Set intrinsic data types.	Boolean, Byte, Char, Date, Decimal, Double, Integer, Long, Object(default), Short, Single, String
Verify data types.	IsArray, IsDate, IsDbNull, IsError, IsNothing, IsNumeric, IsReference

See Also

Keywords and Members by Task | Visual Basic Language Keywords | Visual Basic Run-Time Library Members

Dates and Times Summary

Visual Basic language keywords and run-time library members are organized by purpose and use.

Action	Language element
Get the current date or time.	Now, Today, TimeOfDay
Perform date calculations.	DateAdd, DateDiff, DatePart
Return a date.	DateSerial, DateValue, MonthName, WeekDayName
Return a time.	TimeSerial, TimeValue
Set the date or time.	DateString, TimeOfDay, TimeString, Today
Time a process.	Timer

See Also

Keywords and Members by Task | Visual Basic Language Keywords | Visual Basic Run-Time Library Members

Declarations and Constants Summary

Visual Basic language keywords and run-time library members are organized by purpose and use.

Action	Language element
Assign a value.	Get, Property
Declare variables or constants.	Const, Dim, Private, Protected, Public, Shadows, Shared, Static
Declare a class, delegate, enumeration, module, namespace, or structure.	Class, Delegate, Enum, Module, Namespace, Structure
Create objects.	CreateObject, GetObject, New
Get information about an object.	GetType, IsArray, IsDate, IsDbNull, IsError, IsNothing, IsNumeric, IsReference, SystemTypeName, TypeName, VarType, VbTypeName
Refer to the current object.	Me
Require explicit variable declarations.	Option Explicit, Option Strict
Handle events.	AddHandler, Event, RaiseEvent, RemoveHandler
Implement inheritance.	Inherits, MustInherit, MustOverride, MyBase, MyClass, New, NotInheritable, NotOverridable, Overloads, Overridable, Overrides

See Also

Keywords and Members by Task | Visual Basic Language Keywords | Visual Basic Run-Time Library Members

Directories and Files Summary

Visual Basic language keywords and run-time library members are organized by purpose and use.

Action	Language element
Change a directory or folder.	ChDir
Change the drive.	ChDrive
Copy a file.	FileCopy
Make a directory or folder.	MkDir
Remove a directory or folder.	RmDir
Rename a file, directory, or folder.	Rename
Return the current path.	CurDir
Return a file's date/time stamp.	FileDateTime
Return file, directory, or label attributes.	GetAttr
Return a file's length.	FileLen
Return a file's name or volume label.	Dir
Set attribute information for a file.	SetAttr

See Also

Keywords and Members by Task | Visual Basic Language Keywords | Visual Basic Run-Time Library Members

Errors Summary

Visual Basic language keywords and run-time library members are organized by purpose and use.

Action	Language element
Generate run-time errors.	Clear, Error, Raise
Get exceptions.	GetException
Provide error information.	Err
Trap errors during run time.	On Error, Resume, Try...Catch...Finally
Provide line number of error.	Erl
Provide system error code.	LastDLLError

See Also

Keywords and Members by Task | Visual Basic Language Keywords | Visual Basic Run-Time Library Members

Financial Summary

Visual Basic language keywords and run-time library members are organized by purpose and use.

Action	Language element
Calculate depreciation.	DDB, SLN, SYD
Calculate future value.	FV
Calculate interest rate.	Rate
Calculate internal rate of return.	IRR, MIRR
Calculate number of periods.	NPer
Calculate payments.	IPmt, Pmt, PPmt
Calculate present value.	NPV, PV

See Also

Keywords and Members by Task | Visual Basic Language Keywords | Visual Basic Run-Time Library Members

Input and Output Summary

Visual Basic language keywords and run-time library members are organized by purpose and use.

Action	Language element
Access or create a file.	FileOpen
Close files.	FileClose, Reset
Control output appearance.	Format, Print, Spc, Tab, FileWidth
Copy a file.	FileCopy
Get information about a file.	EOF, FileAttr, FileDateTime, FileLen, FreeFile, GetAttr, Loc, LOF, Seek
Manage files.	Dir, Kill, Lock, Unlock
Read from a file.	FileGet, FileGetObject, Input, InputString, LineInput
Return length of a file.	FileLen
Set or get file attributes.	FileAttr, GetAttr, SetAttr
Set read-write position in a file.	Seek
Write to a file.	FilePut, FilePutObject, Print, Write

See Also

Keywords and Members by Task | Visual Basic Language Keywords | Visual Basic Run-Time Library Members

Math Summary

Visual Basic language keywords and run-time library members are organized by purpose and use.

Action	Language element
Derive trigonometric functions.	Atn, Cos, Sin, Tan
General calculations.	Exp, Log, Sqr
Generate random numbers.	Randomize, Rnd
Get absolute value.	Abs
Get the sign of an expression.	Sgn
Perform numeric conversions.	Fix, Int

See Also

Keywords and Members by Task | Visual Basic Language Keywords | Visual Basic Run-Time Library Members

Derived Math Functions

The following is a list of non-intrinsic math functions that can be derived from the intrinsic math functions:

Function	Derived equivalents
Secant (Sec(x))	1 / **Cos(**x**)**
Cosecant (Csc(x))	1 / **Sin(**x**)**
Cotangent (Ctan(x))	1 / **Tan(**x**)**
Inverse sine (Asin(x))	**Atan(**x / **Sqrt(**-x * x + 1**))**
Inverse cosine (Acos(x))	**Atan(**-x / **Sqrt(**-x * x + 1**))** + 2 * **Atan(**1**)**
Inverse secant (Asec(x))	2 * **Atan(**1**) – Atan(Sign(**x**)** / **Sqrt(**x * x **–** 1**))**
Inverse cosecant (Acsc(x))	**Atan(Sign(**x**)** / **Sqrt(**x * x **–** 1**))**
Inverse cotangent (Acot(x))	2 * **Atan(**1**)** - **Atan(**x**)**
Hyperbolic sine (Sinh(x))	(**Exp(**x**) – Exp(**-x**)**) / 2
Hyperbolic cosine (Cosh(x))	(**Exp(**x**)** + **Exp(**-x**)**) / 2
Hyperbolic tangent (Tanh(x))	(**Exp(**x**) – Exp(**-x**)**) / (**Exp(**x**)** + **Exp(**-x**)**)
Hyperbolic secant (Sech(x))	2 / (**Exp(**x**)** + **Exp(**-x**)**)
Hyperbolic cosecant (Csch(x))	2 / (**Exp(**x**) – Exp(**-x**)**)
Hyperbolic cotangent (Coth(x))	(**Exp(**x**)** + **Exp(**-x**)**) / (**Exp(**x**) – Exp(**-x**)**)
Inverse hyperbolic sine (Asinh(x))	**Log(**x + **Sqrt(**x * x + 1**))**
Inverse hyperbolic cosine (Acosh(x))	**Log(**x + **Sqrt(**x * x **–** 1**))**
Inverse hyperbolic tangent (Atanh(x))	**Log(**(1 + x) / (1 – x)**)** / 2
Inverse hyperbolic secant (AsecH(x))	**Log(**(**Sqrt(**-x * x + 1**)** + 1) / x**)**
Inverse hyperbolic cosecant (Acsch(x))	**Log(**(**Sign(**x**)** * **Sqrt(**x * x + 1**)** + 1) / x**)**
Inverse hyperbolic cotangent (Acoth(x))	**Log(**(x + 1) / (x – 1)**)** / 2

Requirements

Namespace: **System.Math**

Assembly: mscorlib (in mscorlib.dll)

See Also

Math Functions

Information and Interaction Summary

Visual Basic language keywords and run-time library members are organized by purpose and use.

Action	Language element
Run other programs.	AppActivate, Shell
Call a method or property.	CallByName
Sound a beep from computer.	Beep
Provide a command-line string.	Command
Manipulate COM objects.	CreateObject, GetObject
Retrieve color information.	QBColor, RGB
Control dialog boxes	InputBox, MsgBox

See Also

Keywords and Members by Task | Visual Basic Language Keywords | Visual Basic Run-Time Library Members

Operators Summary

Visual Basic language keywords and run-time library members are organized by purpose and use.

Action	Language element
Arithmetic	^, –, *, /, \, Mod, +, =
Assignment	=, ^=, *=, /=, \=, +=, -=, &=
Comparison	=, <>, <, >, <=, >=, Like, Is
Concatenation	&, +
Logical/bitwise operations	Not, And, Or, Xor, AndAlso, OrElse
Miscellaneous operations	AddressOf, GetType

See Also

Keywords and Members by Task | Visual Basic Language Keywords | Visual Basic Run-Time Library Members

Registry Summary

Visual Basic language keywords and run-time library members are organized by purpose and use.

Action	Language element
Delete program settings.	DeleteSetting
Read program settings.	GetSetting, GetAllSettings
Save program settings.	SaveSetting

See Also

Keywords and Members by Task | Visual Basic Language Keywords | Visual Basic Run-Time Library Members

String Manipulation Summary

Visual Basic language keywords and run-time library members are organized by purpose and use.

Action	Language element
Compare two strings.	StrComp
Convert strings.	StrConv
Reverse a string.	InStrRev, StrReverse
Convert to lowercase or uppercase.	Format, LCase, UCase
Create a string of repeating characters.	Space, StrDup
Find the length of a string.	Len
Format a string.	Format, FormatCurrency, FormatDateTime, FormatNumber, FormatPercent
Manipulate strings.	InStr, Left, LTrim, Mid, Right, RTrim, Trim
Set string comparison rules.	Option Compare
Work with ASCII and ANSI values.	Asc, AscW, Chr, ChrW
Replace a specified substring.	Replace
Return a filter-based string array.	Filter
Return a specified number of substrings.	Split, Join

See Also

Keywords and Members by Task | Visual Basic Language Keywords | Visual Basic Run-Time Library Members

Visual Basic .NET Attributes

In This Section

COMClassAttribute Class

VBFixedArrayAttribute Class

VBFixedStringAttribute Class

Related Sections

Attributes (Conceptual)

ComClassAttribute Class

The **ComClassAttribute** attribute instructs the compiler to add metadata that allows a class to be exposed as a COM object.

```
<System.AttributeUsage(System.AttributeTargets.Class, _
   Inherited := False, AllowMultiple := False)> _
Public NotInheritable Class ComClassAttribute
   Inherits System.Attribute
```

Remarks

Use **ComClassAttribute** to simplify the process of creating COM components from Visual Basic .NET. COM objects are considerably different from .NET assemblies, and without the **ComClassAttribute**, you need to follow a number of steps to generate a COM object from Visual Basic .NET. For classes marked with **ComClassAttribute**, the compiler performs many of these steps automatically.

Example

To run this example, create a new **Class Library** application and add the following code to a class module.

```
<ComClass(ComClass1.ClassId, ComClass1.InterfaceId, ComClass1.EventsId)> _
Public Class ComClass1
   ' Use the Region directive to define a section named COM Guids.
#Region "COM GUIDs"
   ' These  GUIDs provide the COM identity for this class
   ' and its COM interfaces. You can generate
   ' these guids using guidgen.exe
    Public Const ClassId As String = "7666AC25-855F-4534-BC55-27BF09D49D46"
    Public Const InterfaceId As String = "54388137-8A76-491e-AA3A-853E23AC1217"
    Public Const EventsId As String = "EA329A13-16A0-478d-B41F-47583A761FF2"
#End Region

   Public Sub New()
      MyBase.New()
   End Sub

   Function AddNumbers(ByVal X As Integer, ByVal Y As Integer)
      AddNumbers = X + Y
   End Function
End Class
```

> **Note** This attribute simplifies the creation of COM objects. To create a COM object, you must compile the project with the **Register for COM Interop** option selected in the **Build** section of the **Configuration Properties** dialog.

Requirements

Namespace: Microsoft.VisualBasic

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Attributes | Applying Interop Attributes | ComClassAttribute Class Members | Walkthrough: Creating COM Objects with Visual Basic .NET | VBFixedArrayAttribute Class | VBFixedStringAttribute Class | Attributes Used in Visual Basic .NET | System.Runtime.InteropServices Namespace | Declaring Variables for Random File Access

ComClassAttribute Class Members

Properties

ClassID Property	Contains a class ID used to uniquely identify the class.
EventID Property	Contains an event ID used to uniquely identify the event.
InterfaceID Property	Contains an interface ID used to uniquely identify the interface.
InterfaceShadows Property	Indicates that a generated COM interface name shadows another member of the class or base class.

Constructor

ComClassAttribute Constructor	Initializes a new instance of the **ComClassAttribute** class.

Requirements

Namespace: Microsoft.VisualBasic

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Attributes | Walkthrough: Creating COM Objects with Visual Basic .NET | Applying Interop Attributes | VBFixedArrayAttribute Class | VBFixedStringAttribute Class | ComClassAttribute Class | Shadows | Attributes Used in Visual Basic .NET | System.Runtime.InteropServices Namespace

ComClassAttribute Constructor

Initializes a new instance of the **ComClassAttribute** class.

```
Public Sub New()
```

-or-

```
Public Sub New( _
   ByVal _ClassID As String _
)
```

-or-

```
Public Sub New( _
   ByVal _ClassID As String, _
   ByVal _InterfaceID As String _
)
```

-or-

```
Public Sub New( _
   ByVal _ClassID As String, _
   ByVal _InterfaceID As String, _
   ByVal _EventID As String _
)
```

Parameters

_ClassID
Initializes the value of the **ClassID** property that is used to uniquely identify a class.
_InterfaceID
Initializes the value of the **InterfaceID** property that is used to uniquely identify an interface.
_EventID
Initializes the value of the **EventID** property that is used to uniquely identify an event.

Remarks

The constructor for the **ComClassAttribute** class allows you to set the **ClassID**, **InterfaceID**, or **EventID** properties when applying the **ComClassAttribute** to a class.

Requirements

Namespace: Microsoft.VisualBasic

Class: ComClassAttribute

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Attributes | ComClassAttribute Class | Attributes Used in Visual Basic .NET |
Walkthrough: Creating COM Objects with Visual Basic .NET | System.Runtime.InteropServices Namespace |
Object Lifetime: How Objects Are Created and Destroyed

ComClassAttribute.ClassID Property

Gets a class ID used to uniquely identify a class.

```
Public ReadOnly Property ClassID As String
```

Property Value

Read only. A string that can be used by the compiler to uniquely identify the class when a COM object is created.

Remarks

The constructor sets this property when the **ComClassAttribute** is applied to a class.

Requirements

Namespace: Microsoft.VisualBasic

Class: ComClassAttribute

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Attributes | ComClassAttribute Class | Attributes Used in Visual Basic .NET | System.Runtime.InteropServices Namespace

ComClassAttribute.EventID Property

Gets an event ID used to uniquely identify an event.

```
Public ReadOnly Property EventID As String
```

Property Value

Read only. A string that can be used by the compiler to uniquely identify an event for the class when a COM object is created.

Remarks

The constructor sets this property when the **ComClassAttribute** is applied to a class.

Requirements

Namespace: Microsoft.VisualBasic

Class: ComClassAttribute

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Attributes | ComClassAttribute Class | Attributes Used in Visual Basic .NET | System.Runtime.InteropServices Namespace

ComClassAttribute.InterfaceID Property

Gets an interface ID used to uniquely identify an interface.

```
Public ReadOnly Property InterfaceID As String
```

Property Value

Read only. A string that can be used by the compiler to uniquely identify an interface for the class when a COM object is created.

Remarks

The constructor sets this property when the **ComClassAttribute** is applied to a class.

Requirements

Namespace: Microsoft.VisualBasic

Class: ComClassAttribute

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Attributes | ComClassAttribute Class | Attributes Used in Visual Basic .NET | System.Runtime.InteropServices Namespace

ComClassAttribute.InterfaceShadows Property

Indicates that the COM interface name shadows another member of the class or base class.

```
Public Property InterfaceShadows As Boolean
```

Property Value

A **Boolean** value that indicates that the COM interface name shadows another member of the class or base class.

Remarks

Shadowing is when a member uses the same name as another member.

Requirements

Namespace: Microsoft.VisualBasic

Class: ComClassAttribute

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Attributes | Shadowing | ComClassAttribute Class | Attributes Used in Visual Basic .NET | System.Runtime.InteropServices Namespace

VBFixedArrayAttribute Class

Indicates that an array in a structure or non-local variable should be treated as a fixed length array.

```
<System.AttributeUsage(System.AttributeTargets.Field, _
   Inherited := False, AllowMultiple := False)> _
Public NotInheritable Class VBFixedArrayAttribute
   Inherits System.Attribute
```

Remarks

Visual Basic .NET arrays are of variable length by default. This attribute is useful when using Visual Basic .NET file input and output functions, such as **FileGet** and **FilePut**, and API calls that require fixed length arrays.

Example

```
    Structure Book
        <VBFixedArray(4)> Public Chapter() As Integer
    End Structure

    Sub WriteData()
        Dim FileNum As Integer = FreeFile()
        Dim MyBook As Book
        ReDim MyBook.Chapter(4)
        ' Add code to populate the array.
        MyBook.Chapter(0) = 1
        MyBook.Chapter(1) = 2
        MyBook.Chapter(2) = 3
        MyBook.Chapter(3) = 4
        MyBook.Chapter(4) = 5
        ' Write the array to a file.
        FileOpen(FileNum, "C:\testfile", OpenMode.Binary, _
                 OpenAccess.Write, OpenShare.Default)
        FilePut(FileNum, MyBook) ' Write data.
        FileClose(FileNum)
    End Sub
```

Note The **VBFixedArrayAttribute** is informational and does not allocate any storage. The purpose of this attribute is to modify how arrays in structures and non-local variables are used by methods or API calls that recognize the **VBFixedArrayAttribute**. Keep in mind that this attribute does not convert a variable length array to a fixed array and that you must still allocate array storage using **Dim** or **ReDim** statements.

Requirements

Namespace: Microsoft.VisualBasic

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Attributes | VBFixedArrayAttribute Class Members | VBFixedStringAttribute Class | MarshalAsAttribute Class | ComClassAttribute Class | Declaring Variables for Random File Access | XML and SOAP Serialization | FileGet Function | FilePut Function | FileOpen Function

VBFixedArrayAttribute Class Members

Properties

Bounds	Returns the bounds of the array.
Length	Returns the size of the array.

Constructor

VBFixedArrayAttribute Constructor	Initializes the bounds of a fixed-sized array.

Requirements

Namespace: Microsoft.VisualBasic

Class: VBFixedArrayAttribute

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Attributes | VBFixedStringAttribute Class | ComClassAttribute Class | MarshalAsAttribute Class | Attributes Used in Visual Basic .NET | Declaring Variables for Random File Access

VBFixedArrayAttribute Constructor

Initializes the value of the **Bounds** property.

```
Public Sub New( _
   ByVal UpperBound1 As Integer, _
   )
```

-or-

```
Public Sub New( _
   ByVal UpperBound1 As Integer, _
   ByVal UpperBound2 As Integer, _
   )
```

Parameters

UpperBound1
Initializes the value of upper field, which represents the size of the first dimension of the array.
UpperBound2
Initializes the value of upper field, which represents the size of the second dimension of the array.

Remarks

The constructor runs when the **VBFixedArrayAttribute** attribute is applied to an array.

Requirements

Namespace: Microsoft.VisualBasic

Class: VBFixedArrayAttribute

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Attributes | VBFixedStringAttribute Class | ComClassAttribute Class | Attributes Used in Visual Basic .NET | MarshalAsAttribute Class | Declaring Variables for Random File Access | Object Lifetime: How Objects Are Created and Destroyed

VBFixedArrayAttribute.Bounds Property

Returns the bounds of the array.

```
Public ReadOnly Property Bounds() As Integer()
```

Property Value

Contains an integer array that represents the bounds of the array.

Remarks

The constructor sets this property when the **VBFixedArrayAttribute** is applied to an array.

Requirements

Namespace: Microsoft.VisualBasic

Class: VBFixedArrayAttribute

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Attributes | VBFixedStringAttribute Class | ComClassAttribute Class | MarshalAsAttribute Class | Attributes Used in Visual Basic .NET | Declaring Variables for Random File Access

VBFixedArrayAttribute.Length Property

Returns the size of the array.

```
Public ReadOnly Property Length As Integer
```

Property Value

Contains an integer that represents the number of elements in the array.

Remarks

The constructor sets this property when the **VBFixedArrayAttribute** is applied to an array.

Requirements

Namespace: Microsoft.VisualBasic

Class: VBFixedArrayAttribute

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Attributes | VBFixedStringAttribute Class | ComClassAttribute Class | MarshalAsAttribute Class | Attributes Used in Visual Basic .NET | Declaring Variables for Random File Access

VBFixedStringAttribute Class

Indicates that a string should be treated as if it were fixed length.

```
<System.AttributeUsage(System.AttributeTargets.Field, _
   Inherited := False, AllowMultiple := False)> _
Public NotInheritable Class VBFixedStringAttribute
   Inherits System.Attribute
```

Remarks

Visual Basic .NET strings are of variable length by default. This attribute is useful when using Visual Basic .NET file input and output functions, such as **FileGet** and **FilePut** that require fixed length strings.

Example

```
Structure Person
   Public ID As Integer
   Public MonthlySalary As Decimal
   Public LastReviewDate As Long
   <VBFixedString(15)> Public FirstName As String
   <VBFixedString(15)> Public LastName As String
   <VBFixedString(15)> Public Title As String
   <VBFixedString(150)> Public ReviewComments As String
End Structure
```

> **Note** The **VBFixedStringAttribute** is informational and cannot be used to convert a variable length string to a fixed string. The purpose of this attribute is to modify how strings in structures and non-local variables are used by methods or API calls that recognize the **VBFixedStringAttribute**. Keep in mind that this attribute does not change the actual length of the string itself.

Smart Device Developer Notes

This class is not supported.

Requirements

Namespace: Microsoft.VisualBasic

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Attributes | VBFixedStringAttribute Class Members | VBFixedArrayAttribute Class | StringBuilder Class | MarshalAsAttribute Class | ComClassAttribute Class | Attributes Used in Visual Basic .NET | Declaring Variables for Random File Access | FileGet Function | FilePut Function | FileOpen Function

VBFixedStringAttribute Class Members

Fields

VBFixedStringAttribute.Length	Gets the length of the string.

Constructor

VBFixedStringAttribute Constructor	Initializes the value of the **SizeConst** field.

Smart Device Developer Notes

This class is not supported.

Requirements

Namespace: Microsoft.VisualBasic

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Attributes | VBFixedArrayAttribute Class | VBFixedStringAttribute Class | StringBuilder Class | MarshalAsAttribute Class | ComClassAttribute Class | Attributes Used in Visual Basic .NET | Declaring Variables for Random File Access

VBFixedStringAttribute Constructor

Initializes the value of the **SizeConst** field.

```
Public Sub New( _
   ByVal Length As Integer _
)
```

Parameters

Length
The length of the fixed string.

Remarks

The constructor runs when the **VBFixedStringAttribute** attribute is applied to a string.

Smart Device Developer Notes

This constructor is not supported.

Requirements

Namespace: Microsoft.VisualBasic

Class: VBFixedStringAttribute

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Attributes | VBFixedStringAttribute Class | StringBuilder Class | MarshalAsAttribute Class | Attributes Used in Visual Basic .NET | Object Lifetime: How Objects Are Created and Destroyed | Declaring Variables for Random File Access

VBFixedStringAttribute.Length Property

Gets the length of the string.

```
Public ReadOnly Property Length As Integer
```

Property Value

Returns the length of the string.

Remarks

The constructor for this attribute sets this property when the **VBFixedStringAttribute** is applied to a string.

Smart Device Developer Notes

This property is not supported.

Requirements

Namespace: Microsoft.VisualBasic

Class: VBFixedStringAttribute

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Attributes | VBFixedStringAttribute Class | Attributes Used in Visual Basic .NET | Declaring Variables for Random File Access

Constants and Enumerations

In This Section

AppWinStyle Enumeration

CallType Enumeration

CompareMethod Enumeration

DateFormat Enumeration

DateInterval Enumeration

DueDate Enumeration

FileAttribute Enumeration

FirstDayOfWeek Enumeration

FirstWeekOfYear Enumeration

MsgBoxResult Enumeration

MsgBoxStyle Enumeration

OpenAccess Enumeration

OpenMode Enumeration

OpenShare Enumeration

Print and Display Constants

Tristate Enumeration

VariantType Enumeration

VBStrConv Enumeration

Related Sections

Visual Basic Language and Run-Time Reference

Visual Basic Language

AppWinStyle Enumeration

When you issue **Shell** commands, you can use the following enumeration members in your code in place of the actual values.

AppWinStyle Enumeration Members

The *Style* argument takes the following **AppWinStyle** enumeration members:

Member	Constant	Description
Hide	**vbHide**	Window is hidden and focus is passed to the hidden window.
NormalFocus	**vbNormalFocus**	Window has focus and is restored to its original size and position.
MinimizedFocus	**vbMinimizedFocus**	Window is displayed as an icon with focus.
MaximizedFocus	**vbMaximizedFocus**	Window is maximized with focus.
NormalNoFocus	**vbNormalNoFocus**	Window is restored to its most recent size and position. The currently active window remains active.
MinimizedNoFocus	**vbMinimizedNoFocus**	Window is displayed as an icon. The currently active window remains active.

Requirements

Namespace: Microsoft.VisualBasic

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Intrinsic Constants and Enumerations | Shell Function

CallType Enumeration

When you call the **CallByName** function, you can use the following **CallType** enumeration members in your code in place of the actual values:

CallType Enumeration Members

The *UseCallType* argument takes the following **CallType** values:

Member	Constant	Description
Method	**vbMethod**	A method is being invoked.
Get	**vbGet**	A property value is being retrieved.
Set	**vbSet**	A property value is being determined.
Let	**vbLet**	An Object property value is being determined.

Smart Device Developer Notes

This enumeration is not supported.

Requirements

Namespace: Microsoft.VisualBasic

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Intrinsic Constants and Enumerations | CallByName Function | CallType Constant Changes in Visual Basic .NET

CompareMethod Enumeration

When you call comparison functions, you can use the **CompareMethod** enumeration in your code in place of the actual values.

CompareMethod Enumeration Members

The *Compare* argument takes the following **CompareMethod** enumeration members:

Member	Constant	Description
Binary	**vbBinaryCompare**	Performs a binary comparison.
Text	**vbTextCompare**	Performs a textual comparison.

Requirements

Namespace: Microsoft.VisualBasic

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Intrinsic Constants and Enumerations | Filter Function | InStr Function | InStrRev Function | Replace Function | Split Function | StrComp Function

DateFormat Enumeration

When you call the **DateValue** function, you can use the following enumeration members in your code in place of the actual values.

Member	Constant	Description
GeneralDate	**vbGeneralDate**	For real numbers, displays a date and time. If the number has no fractional part, displays only a date. If the number has no integer part, displays time only. Date and time display is determined by your computer's regional settings.
LongDate	**vbLongDate**	Displays a date using the long-date format specified in your computer's regional settings.
ShortDate	**vbShortDate**	Displays a date using the short-date format specified in your computer's regional settings.
LongTime	**vbLongTime**	Displays a time using the long-time format specified in your computer's regional settings.
ShortTime	**vbShortTime**	Displays a time using the short-time format specified in your computer's regional settings.

Note You can access your computer's regional settings by double-clicking the Regional Options icon, found in the Control Panel.

Requirements

Namespace: Microsoft.VisualBasic

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

DateValue Function

DateInterval Enumeration

When you call date-related functions, you can use enumeration members in your code in place of the actual values.

The **DateInterval** enumeration defines constants used with date-related functions to identify how date intervals are determined and formatted. The following table lists the **DateInterval** enumeration members.

Member	Description
DateInterval.Day	Day of month (1 through 31)
DateInterval.DayOfYear	Day of year (1 through 366)
DateInterval.Hour	Hour (1 through 24)
DateInterval.Minute	Minute (1 through 60)
DateInterval.Month	Month (1 through 12)
DateInterval.Quarter	Quarter of year (1 through 4)
DateInterval.Second	Second (1 through 60)
DateInterval.Weekday	Day of week (1 through 7)
DateInterval.WeekOfYear	Week of year (1 through 53)
DateInterval.Year	Year

Requirements

Namespace: Microsoft.VisualBasic

Assembly: Microsoft.VisualBasic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Intrinsic Constants and Enumerations | DateFormat Enumeration| DueDate Enumeration | DatePart Function | DateAdd Function | DateDiff Function

DueDate Enumeration

When you call date-related functions, you can use enumeration members in your code in place of the actual values.

The **DueDate** enumeration defines constants used to identify where a due date falls within a date interval. The following table lists the **DueDate** enumeration members.

Member	Description
DueDate.BegOfPeriod	Falls at the beginning of the date interval
DueDate.EndOfPeriod	Falls at the end of the date interval

Requirements

Namespace: Microsoft.VisualBasic

Assembly: Microsoft.VisualBasic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Intrinsic Constants and Enumerations | DateInterval Enumeration | DatePart Function| DateAdd Function | DateDiff Function

FileAttribute Enumeration

When you call the **Dir**, **GetAttr**, or **SetAttr** functions, you can use the **FileAttribute** enumeration in your code in place of the actual values.

FileAttribute Enumeration Members

The *Attributes* argument takes the following **FileAttribute** enumeration members:

Member	Constant	Description
Normal	**vbNormal**	Normal (default for **Dir** and **SetAttr**). No special characteristics apply to this file.
ReadOnly	**vbReadOnly**	Read only.
Hidden	**vbHidden**	Hidden.
System	**vbSystem**	System file.
Volume	**vbVolume**	Volume label.
Directory	**vbDirectory**	Directory or folder.
Archive	**vbArchive**	File has changed since last backup.

Smart Device Developer Notes

This enumeration is not supported.

Requirements

Namespace: Microsoft.VisualBasic

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Intrinsic Constants and Enumerations | Dir Function | GetAttr Function | SetAttr Function

FirstDayOfWeek Enumeration

When you call date-related functions, you can use the following enumeration members in your code in place of the actual values.

Some date-related functions take a *DayOfWeek* argument, a *WeekOfYear* argument, or both. The **FirstDayOfWeek** enumeration specifies the valid values for the *DayOfWeek* arguments, as well as the possible return values from the functions.

FirstDayOfWeek Enumeration Members

The *DayOfWeek* argument takes the following **FirstDayOfWeek** enumeration members:

Member	Constant	Description
System	**vbUseSystemDayOfWeek**	The first day of the week as specified in your system settings
Sunday	**vbSunday**	Sunday (default)
Monday	**vbMonday**	Monday
Tuesday	**vbTuesday**	Tuesday
Wednesday	**vbWednesday**	Wednesday
Thursday	**vbThursday**	Thursday
Friday	**vbFriday**	Friday
Saturday	**vbSaturday**	Saturday

Date-related functions also return **FirstDayOfWeek** values (except **System**).

Requirements

Namespace: Microsoft.VisualBasic

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Intrinsic Constants and Enumerations | FirstWeekOfYear Enumeration | DateFormat Enumeration | DateDiff Function | DatePart Function | Format Function | Weekday Function

FirstWeekOfYear Enumeration

When you call date-related functions, you can use the following enumeration members in your code in place of the actual values.

Some date-related functions take a *DayOfWeek* argument, a *WeekOfYear* argument, or both. The **FirstWeekOfYear** enumeration specifies the valid values for the *WeekOfYear* arguments, as well as the possible return values from the functions.

FirstWeekOfYear Enumeration Members

The *WeekOfYear* argument takes the following **FirstWeekOfYear** enumeration members:

Member	Constant	Description
System	**vbUseSystem**	The day of the week specified in your system settings as the first day of the week
Jan1	**vbFirstJan1**	The week in which January 1 occurs (default)
FirstFourDays	**vbFirstFourDays**	The first week that has at least four days in the new year
FirstFullWeek	**vbFirstFullWeek**	The first full week of the year

Requirements

Namespace: Microsoft.VisualBasic

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Intrinsic Constants and Enumerations | FirstDayOfWeek Enumeration | DateFormat Enumeration | DateDiff Function | DatePart Function | Format Function

MsgBoxResult Enumeration

When you call the **MsgBox** function, you can use the **MsgBoxResult** enumeration in your code in place of the actual values.

MsgBoxResult Enumeration Members

The **MsgBox** function returns the following **MsgBoxResult** enumeration values:

Member	Constant	Description
OK	**vbOK**	**OK** button was pressed.
Cancel	**vbCancel**	**Cancel** button was pressed.
Abort	**vbAbort**	**Abort** button was pressed.
Retry	**vbRetry**	**Retry** button was pressed.
Ignore	**vbIgnore**	**Ignore** button was pressed.
Yes	**vbYes**	**Yes** button was pressed.
No	**vbNo**	**No** button was pressed.

Requirements

Namespace: Microsoft.VisualBasic

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Intrinsic Constants and Enumerations | MsgBoxStyle Enumeration | MsgBox Function

MsgBoxStyle Enumeration

When you call the **MsgBox** function, you can use the **MsgBoxStyle** enumeration in your code in place of the actual values.

MsgBoxStyle Enumeration Members

The *Buttons* argument takes the following **MsgBoxStyle** enumeration members:

Member	Constant	Description
OKOnly	**vbOKOnly**	**OK** button only (default).
OKCancel	**vbOKCancel**	**OK** and **Cancel** buttons.
AbortRetryIgnore	**vbAbortRetryIgnore**	**Abort**, **Retry**, and **Ignore** buttons.
YesNoCancel	**vbYesNoCancel**	**Yes**, **No**, and **Cancel** buttons.
YesNo	**vbYesNo**	**Yes** and **No** buttons.
RetryCancel	**vbRetryCancel**	**Retry** and **Cancel** buttons.
Critical	**vbCritical**	Critical message.
Question	**vbQuestion**	Warning query.
Exclamation	**vbExclamation**	Warning message.
Information	**vbInformation**	Information message.
DefaultButton1	**vbDefaultButton1**	First button is default (default).
DefaultButton2	**vbDefaultButton2**	Second button is default.
DefaultButton3	**vbDefaultButton3**	Third button is default.
ApplicationModal	**vbApplicationModal**	Application modal message box (default).
SystemModal	**vbSystemModal**	System modal message box.
MsgBoxSetForeground	**vbMsgBoxSetForeground**	Foreground message box window.
MsgBoxRight	**vbMsgBoxRight**	Right-aligned text.
MsgBoxRtlReading	**vbMsgBoxRtlReading**	Right-to-left reading text (Hebrew and Arabic systems) .
MsgBoxHelp	**vbMsgBoxHelp**	Help text.

Requirements

Namespace: Microsoft.VisualBasic

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Intrinsic Constants and Enumerations | MsgBoxResult Enumeration | MsgBox Function

OpenAccess Enumeration

When you call file access-related functions, you can use enumeration members in your code in place of the actual values.

The **OpenAccess** enumeration defines constants used to identify how a file can be accessed. The following table lists the **OpenAccess** enumeration members.

Member	Description
OpenAccess.Default	Read and write permitted. This is the default.
OpenAccess.Read	Read permitted.
OpenAccess.ReadWrite	Read and write permitted.
OpenAccess.Write	Write permitted.

Requirements

Namespace: Microsoft.VisualBasic

Assembly: Microsoft.VisualBasic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Intrinsic Constants and Enumerations | OpenAccess (Object Browser) | OpenMode Enumeration | OpenShare Enumeration

OpenMode Enumeration

When you call file access-related functions, you can use enumeration members in your code in place of the actual values.

The **OpenMode** enumeration defines constants used to set file access modes. The following table lists the **OpenMode** enumeration members.

Member	Description
OpenMode.Append	File opened to append to it
OpenMode.Binary	File opened for binary access
OpenMode.Input	File opened for write access
OpenMode.Output	File opened for read access
OpenMode.Random	File opened for random access

Requirements

Namespace: Microsoft.VisualBasic

Assembly: Microsoft.VisualBasic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Intrinsic Constants and Enumerations | OpenShare Enumeration | OpenMode (Object Browser) | OpenAccess Enumeration | Binary File Access | Random File Access

OpenShare Enumeration

When you call file access-related functions, you can use enumeration members in your code in place of the actual values.

The **OpenShare** enumeration defines constants that identify a file's sharing level. The following table lists the **OpenShare** enumeration members.

Member	Description
OpenShare.Default	Shared. This is the default.
OpenShare.LockRead	Other processes cannot read the file.
OpenShare.LockReadWrite	Other processes cannot read or write to the file.
OpenShare.LockWrite	Other processes cannot write to the file.
OpenShare.Shared	Any process can read or write to the file.

Requirements

Namespace: Microsoft.VisualBasic

Assembly: Microsoft.VisualBasic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Intrinsic Constants and Enumerations | OpenShare (Object Browser) | OpenAccess Enumeration | OpenMode Enumeration

Print and Display Constants

When you call print and display functions, you can use the following constants in your code in place of the actual values.

ControlChars Module Members

Member	Constant	Equivalent	Description
CrLf	**vbCrLf**	**Chr(**13**) + Chr(**10**)**	Carriage return/linefeed character combination.
Cr	**vbCr**	**Chr(**13**)**	Carriage return character.
Lf	**vbLf**	**Chr(**10**)**	Linefeed character.
NewLine	**vbNewLine**	**Chr(**13**) + Chr(**10**)**	New line character.
NullChar	**vbNullChar**	**Chr(**0**)**	Character having value 0.
na	**vbNullString**	String having value 0	Not the same as a zero-length string (""); used for calling external procedures.
na	**vbObjectError**	-2147221504	Error number. User-defined error numbers should be greater than this value. For example: `Err.Raise(Number) = vbObjectError + 1000`
Tab	**vbTab**	**Chr(**9**)**	Tab character.
Back	**vbBack**	**Chr(**8**)**	Backspace character.
FormFeed	**vbFormFeed**	**Chr(**12**)**	Not useful in Microsoft Windows.
VerticalTab	**vbVerticalTab**	**Chr(**11**)**	Not useful in Microsoft Windows.
Quote	na	**Chr(**34**)**	Quotation mark character (" or ') used to enclose values.

Requirements

Namespace: Microsoft.VisualBasic

Module: Constants

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Intrinsic Constants and Enumerations | Print, Printline Functions | Write, WriteLine Functions

TriState Enumeration

When you call number-formatting functions, you can use the following enumeration members in your code in place of the actual values.

Constant	Value	Description
True	–1	True
False	0	False
UseDefault	–2	Default setting

Requirements

Namespace: Microsoft.VisualBasic

Module: Strings

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

FormatNumber Function | FormatCurrency Function | FormatPercent Function

VariantType Enumeration

When you call the **VarType** function, you can use enumeration members in your code in place of the actual values.

The **VariantType** enumeration defines constants used to identify possible **Variant** types. The following table lists the **VariantType** enumeration members.

Member	Constant	Description
VariantType.Array	**vbArray**	Array.
VariantType.Boolean	**vbBoolean**	**Boolean**. (**True** or **False**.)
VariantType.Byte	**vbByte**	**Byte**. (0 through 255.)
VariantType.Char	**vbChar**	**Char**. (0 through 65535.)
VariantType.Currency	**vbCurrency**	Currency.
VariantType.DataObject		DataObject.
VariantType.Date	**vbDate**	**Date**. (0:00:00 on January 1, 0001 through 11:59:59 PM on December 31, 9999.)
VariantType.Decimal	**vbDecimal**	**Decimal**. (0 through +/-79,228,162,514,264,337,593,543,950,335 with no decimal point; 0 through +/-7.9228162514264337593543950335 with 28 places to the right of the decimal; smallest non-zero number is +/-0.0000000000000000000000000001.)
VariantType.Double	**vbDouble**	**Double**. (-1.79769313486231E+308 through -4.94065645841247E-324 for negative values; 4.94065645841247E-324 through 1.79769313486231E+308 for positive values.)
VariantType.Empty	**vbEmpty**	Null reference.
VariantType.Error		
VariantType.Integer	**vbInteger**	**Integer**. (-2,147,483,648 through 2,147,483,647.)
VariantType.Long	**vbLong**	**Long**. (-9,223,372,036,854,775,808 through 9,223,372,036,854,775,807.)
VariantType.Null	**vbNull**	Null object.
VariantType.Object	**vbObject**	Any type can be stored in a variable of type **Object**.
VariantType.Short		**Short**. (-32,768 through 32,767.)
VariantType.Single	**vbSingle**	**Single**. (-3.402823E+38 through -1.401298E-45 for negative values; 1.401298E-45 through 3.402823E+38 for positive values.)
VariantType.String	**vbString**	**String**. (0 to approximately 2 billion Unicode characters.)
VariantType.UserDefinedType	**vbUserDefinedType**	User-defined type. Each member of the structure has a range determined by its data type and independent of the ranges of the other members.
VariantType.Variant	**vbVariant**	**Variant**.

Smart Device Developer Notes

This enumeration is not supported.

Requirements

Namespace: Microsoft.VisualBasic

Assembly: Microsoft.VisualBasic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Intrinsic Constants and Enumerations | VarType Function

VbStrConv Enumeration

When you call the **StrConv** function, you can use the following enumeration members in your code in place of the actual values.

VbStrConv Enumeration Members

The *Conversion* argument takes the following **VbStrConv** enumeration members:

Member	Constant	Description
UpperCase	**vbUpperCase**	Converts the string to uppercase characters.
LowerCase	**vbLowerCase**	Converts the string to lowercase characters.
ProperCase	**vbProperCase**	Converts the first letter of every word in the string to uppercase.
Wide	**vbWide**	Converts narrow (single-byte) characters in the string to wide (double-byte) characters. Applies to Asian locales.
Narrow	**vbNarrow**	Converts wide (double-byte) characters in the string to narrow (single-byte) characters. Applies to Asian locales.
Katakana	**vbKatakana**	Converts Hiragana characters in the string to Katakana characters. Applies to Japanese locale only.
Hiragana	**vbHiragana**	Converts Katakana characters in the string to Hiragana characters. Applies to Japanese locale only.
SimplifiedChinese	**vbSimplifiedChinese**	Converts the string to Simplified Chinese characters.
TraditionalChinese	**vbTraditionalChinese**	Converts the string to Traditional Chinese characters.
LinguisticCasing	**vbLinguisticCasing**	Converts the string from file system rules for casing to linguistic rules.

Smart Device Developer Notes

This enumeration is not supported.

Requirements

Namespace: Microsoft.VisualBasic

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Intrinsic Constants and Enumerations | StrConv Function

Data Types

In This Section

Data Type Summary

Boolean Data Type

Byte Data Type

Char Data Type

Date Data Type

Decimal Data Type

Double Data Type

Integer Data Type

Long Data Type

Object Data Type

Short Data Type

Single Data Type

String Data Type

User-Defined Data Type

Related Sections

Visual Basic Language and Run-Time Reference

Visual Basic Language

Data Type Summary

The following table shows the Visual Basic .NET data types, their supporting common language runtime types, their nominal storage allocation, and their value ranges.

Visual Basic type	Common language runtime type structure	Nominal storage allocation	Value range
Boolean	**System.Boolean**	2 bytes	**True** or **False.**
Byte	**System.Byte**	1 byte	0 through 255 (unsigned).
Char	**System.Char**	2 bytes	0 through 65535 (unsigned).
Date	**System.DateTime**	8 bytes	0:00:00 on January 1, 0001 through 11:59:59 PM on December 31, 9999.
Decimal	**System.Decimal**	16 bytes	0 through +/-79,228,162,514,264,337,593,543,950,335 with no decimal point; 0 through +/-7.9228162514264337593543950335 with 28 places to the right of the decimal; smallest nonzero number is +/-0.0000000000000000000000000001 (+/-1E-28).
Double (double-precision floating-point)	**System.Double**	8 bytes	-1.79769313486231570E+308 through -4.94065645841246544E-324 for negative values; 4.94065645841246544E-324 through 1.79769313486231570E+308 for positive values.
Integer	**System.Int32**	4 bytes	-2,147,483,648 through 2,147,483,647.
Long (long integer)	**System.Int64**	8 bytes	-9,223,372,036,854,775,808 through 9,223,372,036,854,775,807.
Object	**System.Object** (class)	4 bytes	Any type can be stored in a variable of type **Object**.
Short	**System.Int16**	2 bytes	-32,768 through 32,767.
Single (single-precision floating-point)	**System.Single**	4 bytes	-3.4028235E+38 through -1.401298E-45 for negative values; 1.401298E-45 through 3.4028235E+38 for positive values.
String (variable-length)	**System.String** (class)	Depends on implementing platform	0 to approximately 2 billion Unicode characters.
User-Defined Type (structure)	(inherits from **System.ValueType**)	Depends on implementing platform	Each member of the structure has a range determined by its data type and independent of the ranges of the other members.

Note For strings containing text, use the **StrConv** function to convert from one text format to another.

Memory Consumption

When you declare an elementary data type, it is not safe to assume that its memory consumption is the same as its nominal storage allocation. The common language runtime assigns storage based on the characteristics of the platform on which your application is executing. In some circumstances it packs your declared elements as closely together as possible; in other cases it aligns their memory addresses to natural hardware boundaries. Also, storage assignment is different on a 64-bit platform than it is on a 32-bit platform.

The same considerations apply to each member of a composite data type such as a structure or an array. Furthermore, some composite types have additional memory requirements. For example, an array uses extra memory for the array itself and also for each dimension. On a 32-bit platform, this overhead is currently 12 bytes plus 8 bytes for each dimension. On a 64-bit platform the requirement is doubled. You cannot rely on simply adding together the nominal storage allocations of the components.

An **Object** referring to any elementary or composite data type uses 4 bytes in addition to the data contained in the data type.

See Also

Boolean Data Type | Byte Data Type | Char Data Type | Date Data Type | Decimal Data Type | Double Data Type | Integer Data Type | Long Data Type | Object Data Type | Short Data Type | Single Data Type | String Data Type | User-Defined Data Type | Type Conversion Functions | Efficient Use of Data Types | Int, Fix Functions | StrConv Function

Boolean Data Type

Boolean variables are stored as 16-bit (2-byte) numbers, but they can only be **True** or **False**. Use the keywords **True** and **False** to assign one of the two states to **Boolean** variables.

When numeric data types are converted to **Boolean** values, 0 becomes **False** and all other values become **True**. When **Boolean** values are converted to numeric types, **False** becomes 0 and **True** becomes -1.

> **Note** The **ToInt32** method of the **Convert** class in the **System** namespace converts **True** to +1. If you need to convert a **Boolean** value to a numeric data type, take care which conversion method you use.
>
> **Caution** You should never write code that relies on equivalent numeric values for **True** and **False**. Whenever possible, you should restrict usage of **Boolean** variables to the logical values for which they are designed. If it is necessary to mix **Boolean** and numeric values, be sure to use the appropriate conversion keywords.

The equivalent .NET data type is **System.Boolean**.

See Also

Data Type Summary | Boolean Structure | Integer Data Type | Type Conversion Functions | Conversion Summary | Convert.ToInt32 Method (Boolean) | Efficient Use of Data Types

Byte Data Type

Byte variables are stored as unsigned 8-bit (1-byte) numbers ranging in value from 0 through 255.

The **Byte** data type is used for containing binary data.

> **Note** The **Byte** data type can be converted to the **Short**, **Integer**, **Long**, **Single**, **Double**, or **Decimal** data type without encountering a **System.OverflowException** error.

The equivalent .NET data type is **System.Byte**.

See Also

Data Type Summary | Byte Structure | Integer Data Type | Short Data Type | Type Conversion Functions | Conversion Summary | Efficient Use of Data Types

Char Data Type

Char variables are stored as unsigned 16-bit (2-byte) numbers ranging in value from 0 through 65535. Each number represents a single Unicode character. Direct conversions between the **Char** data type and the numeric types are not possible, but you can use the **AscW** and **ChrW** functions for this purpose.

Appending the literal type character **C** to a single-character string literal forces it to the **Char** data type. This is required if the type checking switch (**Option Strict**) is on, as the following example shows:

```
Option Strict On
' ...
Dim CharVar As Char
CharVar = "Z"    ' Cannot convert String to Char with Option Strict On.
CharVar = "Z"C   ' Successfully assigns single character to CharVar.
```

The equivalent .NET data type is **System.Char**.

See Also

Data Type Summary | Char Structure | Integer Data Type | Type Conversion Functions | Conversion Summary | Implicit and Explicit Conversions | Efficient Use of Data Types | Asc, AscW Functions | Chr, ChrW Functions

Date Data Type

Date variables are stored as IEEE 64-bit (8-byte) integers that represent dates ranging from January 1 of the year 1 through December 31 of the year 9999, and times from 0:00:00 (midnight) through 11:59:59 PM.

Date values must be enclosed within number signs (**#**) and be in the format M/d/yyyy, for example `#5/31/1993#`. If you convert a **Date** value to the **String** type, the date is rendered according to the short date format recognized by your computer, and the time is rendered according to the time format (either 12-hour or 24-hour) in effect on your computer.

The equivalent .NET data type is **System.DateTime**.

See Also

Data Type Summary | DateTime Structure | Double Data Type | Type Conversion Functions | Conversion Summary | Data Type Changes in Visual Basic | Efficient Use of Data Types

Decimal Data Type

Decimal variables are stored as signed 128-bit (16-byte) integers scaled by a variable power of 10. The scaling factor specifies the number of digits to the right of the decimal point; it ranges from 0 through 28. With a scale of 0 (no decimal places), the largest possible value is +/-79,228,162,514,264,337,593,543,950,335. With 28 decimal places, the largest value is +/-7.9228162514264337593543950335, and the smallest nonzero value is +/-0.0000000000000000000000000001 (+/-1E-28).

Appending the literal type character **D** to a literal forces it to the **Decimal** data type. Appending the identifier type character **@** to any identifier forces it to **Decimal**. You might need to use the **D** type character to assign a large value to a **Decimal** variable or constant, as the following example shows:

```
Dim BigDec1 As Decimal = 9223372036854775807   ' No overflow.
Dim BigDec2 As Decimal = 9223372036854775808   ' Overflow.
Dim BigDec3 As Decimal = 9223372036854775808D  ' No overflow.
```

This is because without a literal type character the literal is taken as **Long**, and the value to be assigned to `BigDec2` is too large for the **Long** type.

The equivalent .NET data type is **System.Decimal**.

See Also

Data Type Summary | Decimal Structure | Type Conversion Functions | Conversion Summary | Efficient Use of Data Types

Double Data Type

Double variables are stored as signed IEEE 64-bit (8-byte) double-precision floating-point numbers ranging in value from -1.79769313486231570E+308 through -4.94065645841246544E-324 for negative values and from 4.94065645841246544E-324 through 1.79769313486231570E+308 for positive values.

> **Note** The **Double** data type can be converted to the **Decimal** data type without encountering a **System.OverflowException** error.

Appending the literal type character **R** to a literal forces it to the **Double** data type. Appending the identifier type character **#** to any identifier forces it to **Double**.

The equivalent .NET data type is **System.Double**.

See Also

Data Type Summary | Double Structure | Single Data Type | Type Conversion Functions | Conversion Summary | Efficient Use of Data Types

Integer Data Type

Integer variables are stored as signed 32-bit (4-byte) integers ranging in value from -2,147,483,648 through 2,147,483,647.

The **Integer** data type provides optimal performance on a 32-bit processor, as the smaller integral types are slower to load and store from and to memory.

You can convert the **Integer** data type to **Long**, **Single**, **Double**, or **Decimal** without encountering a **System.OverflowException** error.

> **Caution** If you are interfacing with components written in Visual Basic version 6.0, for example Automation or COM objects, keep in mind that **Integer** has a different data width (16 bits) in Visual Basic 6.0. If you are passing a 16-bit argument to such a component, declare it as **Short** instead of **Integer** in Visual Basic .NET.

Appending the literal type character **I** to a literal forces it to the **Integer** data type. Appending the identifier type character **%** to any identifier forces it to **Integer**.

The equivalent .NET data type is **System.Int32**.

See Also

Data Type Summary | Int32 Structure | Long Data Type | Short Data Type | Type Conversion Functions | Conversion Summary | Integer Data Type Changes in Visual Basic | Efficient Use of Data Types

Long Data Type

Long variables are stored as signed 64-bit (8-byte) integers ranging in value from -9,223,372,036,854,775,808 through 9,223,372,036,854,775,807.

You can convert the **Long** data type to **Single**, **Double**, or **Decimal** without encountering a **System.OverflowException** error.

> **Caution** If you are interfacing with components written in Visual Basic version 6.0, for example Automation or COM objects, keep in mind that **Long** has a different data width (32 bits) in Visual Basic 6.0. If you are passing a 32-bit argument to such a component, declare it as **Integer** instead of **Long** in Visual Basic .NET.
>
> Furthermore, Automation does not support 64-bit integers on certain platforms, including Windows 2000 and Windows 98. You cannot pass a Visual Basic .NET **Long** argument to an Automation component on such platforms.

Appending the literal type character **L** to a literal forces it to the **Long** data type. Appending the identifier type character **&** to any identifier forces it to **Long**.

The equivalent .NET data type is **System.Int64**.

See Also

Data Type Summary | Int64 Structure | Integer Data Type | Short Data Type | Type Conversion Functions | Conversion Summary | Integer Data Type Changes in Visual Basic | Efficient Use of Data Types

Object Data Type

Object variables are stored as 32-bit (4-byte) addresses that refer to objects. You can assign any reference type (string, array, class, or interface) to a variable declared as an **Object**.

An **Object** variable can also refer to data of any value type (numeric, **Boolean**, **Char**, **Date**, structure, or enumeration).

> **Note** Although a variable declared with **Object** type is flexible enough to contain a reference to any object, the invocation of a method of an instance using an **Object** variable is always late (run-time) binding. To force early (compile-time) binding, assign the object reference to a variable declared with a specific class name or cast to the specific data type.

The equivalent .NET data type is **System.Object**.

See Also

Data Type Summary | Object Class | Type Conversion Functions | Conversion Summary | Efficient Use of Data Types

Short Data Type

Short variables are stored as signed 16-bit (2-byte) integers ranging in value from -32,768 through 32,767.

> **Note** The **Short** data type can be converted to the **Integer**, **Long**, **Single**, **Double**, or **Decimal** data type without encountering a **System.OverflowException** error.

Appending the literal type character **S** to a literal forces it to the **Short** data type.

The equivalent .NET data type is **System.Int16**.

See Also

Data Type Summary | Int16 Structure | Type Conversion Functions | Conversion Summary | Byte Data Type | Integer Data Type | Long Data Type | Integer Data Type Changes in Visual Basic | Efficient Use of Data Types

Single Data Type

Single variables are stored as signed IEEE 32-bit (4-byte) single-precision floating-point numbers ranging in value from -3.4028235E+38 through -1.401298E-45 for negative values and from 1.401298E-45 through 3.4028235E+38 for positive values. Single-precision numbers store an approximation of a real number.

> **Note** The **Single** data type can be converted to the **Double** or **Decimal** data type without encountering a **System.OverflowException** error.

Appending the literal type character **F** to a literal forces it to the **Single** data type. Appending the identifier type character **!** to any identifier forces it to **Single**.

The equivalent .NET data type is **System.Single**.

See Also

Data Type Summary | Single Structure | Double Data Type | Type Conversion Functions | Conversion Summary | Efficient Use of Data Types

String Data Type

String variables are stored as sequences of unsigned 16-bit (2-byte) numbers ranging in value from 0 through 65535. Each number represents a single Unicode character. A string can contain up to approximately 2 billion (2 ^ 31) Unicode characters.

The first 128 code points (0–127) of Unicode correspond to the letters and symbols on a standard U.S. keyboard. These first 128 code points are the same as those defined by the ASCII character set. The second 128 code points (128–255) represent special characters, such as Latin-based alphabet letters, accents, currency symbols, and fractions. The remaining code points are used for a wide variety of symbols, including worldwide textual characters, diacritics, and mathematical and technical symbols.

Appending the identifier type character **$** to any identifier forces it to the **String** data type.

The equivalent .NET data type is **System.String**.

See Also

Data Type Summary | String Class | Type Conversion Functions | Conversion Summary | Efficient Use of Data Types

User-Defined Data Type

The user-defined type (UDT) supported by previous versions of Visual Basic is generalized to a *structure* in Visual Basic .NET. A structure is a concatenation of one or more *members* of various data types. The structure is treated as a single unit, although its members can also be accessed individually.

A structure declaration starts with the **Structure** statement and ends with the **End Structure** statement. The **Structure** statement supplies the name of the structure, which is also the identifier of the data type being defined by the structure. Other parts of the code can use this identifier to declare the data type of variables, arguments, and function return values.

The following paradigm shows the declaration of a structure.

```
[Public | Protected | Friend | Protected Friend | Private] Structure structname
   {Dim | Public | Friend | Private} member1 As datatype1
   ...
   {Dim | Public | Friend | Private} memberN As datatypeN
End Structure
```

The declarations between the **Structure** and **End Structure** statements define the members of the structure. Every member must be declared using a **Dim** statement or a statement that specifies access, such as **Public**, **Private**, or **Friend**. If a **Dim** statement is used, the access defaults to public.

> **Note** As with all composite data types, you cannot safely calculate the total memory consumption of a structure by adding together the nominal storage allocations of its members. Furthermore, you cannot safely assume that the order of storage in memory is the same as your order of declaration.

All structures inherit from the .NET class **System.ValueType**.

See Also

Data Type Summary | ValueType Class | Structures: Your Own Data Types | Efficient Use of Data Types | Structure Statement

Directives

In This Section

#Const Directive

#ExternalSource Directive

#If...Then...#Else Directives

#Region Directive

Related Sections

Visual Basic Language and Run-Time Reference

Visual Basic Language

#Const Directive

Defines conditional compiler constants for Visual Basic.

```
#Const constname = expression
```

Parts

constname
Required. **String**. Name of the constant being defined.

expression
Required. Literal, other conditional compiler constant, or any combination that includes any or all arithmetic or logical operators except **Is**.

Remarks

Conditional compiler constants are always private to the file in which they appear. You cannot create public compiler constants using the **#Const** directive; you can create them only in the user interface.

You can use only conditional compiler constants and literals in *expression*. Using a standard constant defined with **#Const** causes an error. Conversely, you can use constants defined with the **#Const** keyword only for conditional compilation. Constants also can be undefined, in which case they have a value of **Nothing**.

Example

This example uses the **#Const** directive.

```
#Const MyLocation = "USA"
#Const Version = "8.0.0012"
#Const CustomerNumber = 36
```

See Also

#If...Then...#Else Directives | Const Statement | Conditional Compilation Overview | Declaring Conditional Compilation Constants | If...Then...Else Statements (Conceptual)

#ExternalSource Directive

Indicates a mapping between specific lines of source code and text external to the source.

```
# ExternalSource( StringLiteral , IntLiteral )
   [ LogicalLine+ ]
# End ExternalSource
```

Parts

StringLiteral
The path to the external source.
IntLiteral
The line number of the first line of the external source.
LogicalLine
The line where the error occurs in the external source.
 # End ExternalSource
Terminates the **#ExternalSource** block.

Remarks

This directive is used only by the compiler and the debugger.

A source file may include external source directives, which indicate a mapping between specific lines of code in the source file and text external to the source, such as an .aspx file. If errors are encountered in the designated source code during compilation, they are identified as coming from the external source.

External source directives have no effect on compilation and may not be nested. They are intended for internal use by the application only.

See Also

Conditional Compilation

#If...Then...#Else Directives

Conditionally compiles selected blocks of Visual Basic code.

```
#If expression Then
   statements
[ #ElseIf expression Then
   [ statements ]
...
#ElseIf expression Then
   [ statements ] ]
[ #Else
   [ statements ] ]
#End If
```

Parts

expression
Required for **If** and **ElseIf** statements, optional elsewhere. Any expression, consisting exclusively of one or more conditional compiler constants, literals, and operators, that evaluates to **True** or **False**. Three conditional compilation constants are provided: **Config**, **Debug**, and **Trace**. **Debug** and **Trace** are **Boolean** datatypes and can be set in the Project Properties dialogue. When **Debug** is defined, Debug class methods generate output to the **Output** window. When it is not defined, Debug class methods are not compiled and no Debug output is generated. Similarly, when **Trace** is defined, Trace class methods generate output to the **Output** window. When it is not defined, Trace class methods are not compiled and no Trace output is generated. **Config** is a string datatype, which corresponds to the current setting in the Configuration Manager.

statements
Required for **If** statement block, optional elsewhere. Visual Basic program lines or compiler directives that are compiled if the associated expression evaluates to **True**.

#End If
Terminates the #**If** statement block.

Remarks

On the surface, the behavior of the **#If...Then...#Else** directives appears the same as that of the **If...Then...Else** statements. However, the **#If...Then...#Else** directives evaluate what is compiled by the compiler, whereas the **If...Then...Else** statements evaluate conditions at run time.

Conditional compilation is typically used to compile the same program for different platforms. It is also used to prevent debugging code from appearing in an executable file. Code excluded during conditional compilation is completely omitted from the final executable file, so it has no effect on size or performance.

Regardless of the outcome of any evaluation, all expressions are evaluated using **Option Compare Text**. The **Option Compare** statement does not affect expressions in **#If** and **#ElseIf** statements.

> **Note** No single-line form of the **#If**, **#Else**, **#ElseIf**, and **#End If** directives exists; that is, no other code can appear on the same line as any of the directives.

Example

This example uses the **#If...Then...#Else** construct to determine whether to compile certain statements.

```
#Const CustomerNumber = 36
#If CustomerNumber = 35 Then
    ' Insert code to be compiled for customer # 35.
#ElseIf CustomerNumber = 36 Then
    ' Insert code to be compiled for customer # 36.
#Else
    ' Insert code to be compiled for all other customers.
#End If
```

See Also

#Const Directive | If...Then...Else Statements (Language Reference) | Conditional Compilation Overview | If...Then...Else Statements (Conceptual)

#Region Directive

Collapses and hides sections of code in Visual Basic .NET files.

```
#Region "identifier_string"
#End Region
```

Parts

identifier_string
Required. String that acts as the title of a region when it is collapsed. Regions are collapsed by default.

#End Region
Terminates the #**Region** block.

Remarks

The **#Region** directive lets you specify a block of code that you can expand or collapse when using the outlining feature of the Visual Studio Code Editor. **#Region** statements support block semantics (such as **#If...#End If**), meaning that the start and end must be in the same code block.

Example

This example uses the **#Region** directive.

```
#Region "MathFunctions"
' Insert code for the Math functions here.
#End Region
```

See Also

#If...Then...#Else Directives (Language Reference) | Outlining and Hiding Code | Collapsing and Hiding Sections of Code

Functions

In This Section

A-C Functions

D-G Functions

H-L Functions

M-R Functions

S-Z Functions

Related Sections

Visual Basic Language and Run-Time Reference

Visual Basic Language

A-C Functions

The following table contains a listing of the Visual Basic .NET run-time member functions.

Abs	AppActivate	Asc	Atn
Beep	CallByName	CBool	CByte
CChar	CDate	CDbl	CDec
ChDir	ChDrive	Choose	Chr
ChrW	CInt	CLng	CObj
Command	Conversions	Cos	CreateObject
CShort	CSng	CStr	CType
CurDir			

See Also

D-G Functions | H-L Functions | M-R Functions | S-Z Functions | Visual Basic Language and Run-Time Reference

Math Functions

The math functions in Visual Basic 6 have been replaced by equivalent methods in the **System.Math** class of the .NET Framework.

Remarks

The .NET Framework math methods are functionally identical to their Visual Basic 6 counterparts, although some have slightly different names. For example, the .NET Framework equivalent of the Visual Basic 6 Atn function is **Atan**. The following table lists the Visual Basic 6 math function names and the equivalent .NET Framework methods.

Visual Basic 6 function	Visual Basic .NET method	Description
Abs	Math.Abs Method	Returns the absolute value of a specified number.
Atn	Math.Atan Method	Returns a **Double** value containing the angle whose tangent is the specified number.
Cos	Math.Cos Method	Returns a **Double** value containing the cosine of the specified angle.
Exp	Math.Exp Method	Returns a **Double** value containing e (the base of natural logarithms) raised to the specified power.
Log	Math.Log Method	Returns a **Double** value containing the logarithm of a specified number. This method is overloaded and can return either the natural (base e) logarithm of a specified number or the logarithm of a specified number in a specified base.
Round	Math.Round Method	Returns a **Double** value containing the number nearest the specified value. Additional round functions are available as methods of the intrinsic types such as Decimal.Round Method.
Sgn	Math.Sign Method	Returns an **Integer** value indicating the sign of a number.
Sin	Math.Sin Method	Returns a **Double** value specifying the sine of an angle.
Sqr	Math.Sqrt Method	Returns a **Double** value specifying the square root of a number.
Tan	Math.Tan Method	Returns a **Double** value containing the tangent of an angle.

In addition, the .NET Framework math class provides constants and other static methods for trigonometric, logarithmic, and other common mathematical functions. All of these can be used in a Visual Basic program.

To use these functions without qualification, import the **System.Math** namespace into your project by adding the following code to the top of the source code:

```
Imports System.Math
```

Requirements

Class: Math Class

Abs Example

This example uses the **Abs** method of the **Math** class to compute the absolute value of a number.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim MyNumber As Double
```

```
MyNumber = Abs(50.3)    ' Returns 50.3.
MyNumber = Abs(-50.3)   ' Returns 50.3.
```

Atan Example

This example uses the **Atan** method of the **Math** class to calculate the value of pi.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim pi As Double
pi = 4 * Atan(1)   ' Calculate the value of pi.
```

Cos Example

This example uses the **Cos** method of the **Math** class to return the cosine of an angle.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim MyAngle, MySecant As Double
MyAngle = 1.3   ' Define angle in radians.
MySecant = 1 / Cos(MyAngle)   ' Calculate secant.
```

Exp Example

This example uses the **Exp** method of the **Math** class to return *e* raised to a power.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim MyAngle, MyHSin As Double
' Define angle in radians.
MyAngle = 1.3
' Calculate hyperbolic sine.
MyHSin = (Exp(MyAngle) - Exp(-1 * MyAngle)) / 2
```

Log Example

This example uses the **Log** method of the **Math** class to return the natural logarithm of a number.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim MyAngle, MyLog As Double
' Define angle in radians.
MyAngle = 1.3
' Calculate inverse hyperbolic sine.
MyLog = Log(MyAngle + Sqrt(MyAngle * MyAngle + 1))
```

Round Example

This example uses the **Round** method of the **Math** class to round a number to the nearest integer.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim MyVar1 As Double = 2.8
Dim MyVar2 As Double
MyVar2 =Round(MyVar1)   ' Returns 3.
```

Sign Example

This example uses the **Sign** method of the **Math** class to determine the sign of a number.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim MyVar1, MyVar2, MyVar3 As Double
Dim MySign As Integer
MyVar1 = 12
MyVar2 = -2.4
MyVar3 = 0
MySign = Sign(MyVar1)   ' Returns 1.
MySign = Sign(MyVar2)   ' Returns -1.
MySign = Sign(MyVar3)   ' Returns 0.
```

Sin Example

This example uses the **Sin** method of the **Math** class to return the sine of an angle.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim MyAngle, MyCosecant As Double
MyAngle = 1.3   ' Define angle in radians.
MyCosecant = 1 / Sin(MyAngle)   ' Calculate cosecant.
```

Sqrt Example

This example uses the **Sqrt** method of the **Math** class to calculate the square root of a number.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim MySqr As Double
MySqr = Sqrt(4)    ' Returns 2.
MySqr = Sqrt(23)   ' Returns 4.79583152331272.
MySqr = Sqrt(0)    ' Returns 0.
MySqr = Sqrt(-4)   ' Returns NaN (not a number).
```

Tan Example

This example uses the **Tan** method of the **Math** class to return the tangent of an angle.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim MyAngle, MyCotangent As Double
MyAngle = 1.3   ' Define angle in radians.
MyCotangent = 1 / Tan(MyAngle)   ' Calculate cotangent.
```

Requirements

Namespace: System.Math

Assembly: mscorlib (in mscorlib.dll)

See Also

Rnd Function | Randomize Statement | Derived Math Functions | Double.NaN Field

AppActivate Function

Activates an application window.

```
Public Overloads Sub AppActivate( _
   ByVal { Title As String | ProcessID As Integer } _
)
```

Parameters

Title
String expression specifying the title in the title bar of the application window you want to activate. You can use the task ID returned by the **Shell** function.

ProcessID
Integer specifying the Win32 Process ID number assigned to this process.

Exceptions/Errors

Exception type	Error number	Condition
ArgumentException	5	*ProcessID* is not found .

Remarks

The **AppActivate** function changes the focus to the named application or window, but does not affect whether it is maximized or minimized. Focus moves from the activated application window when the user takes some action to change the focus or close the window. Use the **Shell** function to start an application and set the window style.

You can use **AppActivate** only with processes that own windows. Most console applications do not own windows, which means that they do not appear in the list of processes that **AppActivate** searches. When running from a console application, the system creates a separate process to run the application and returns the output to the console process. Consequently, when the current Process ID is requested, you get the created process's Process ID rather than the console application's Process ID.

At run time the **AppActivate** function activates any running application whose title matches *Title* or whose Process ID matches *ProcessID*. If there is no exact match, it activates any application whose title string begins with *Title*. If there is more than one application named by *Title*, the **AppActivate** function activates one at random.

> **Note** The **AppActivate** function requires UIPermission at the SafeTopLevelWindows level, which may affect its execution in partial trust situations. For more information, see Requesting Permissions and UIPermission Class.

Example

This example illustrates various uses of the **AppActivate** function to activate an application window. If a notepad process is not running, the example throws an exception. The **Shell** procedure assumes the applications are in the paths specified.

```
Dim notepadID As Integer
' Activate a running notepad process.
AppActivate("Untitled - Notepad")
' AppActivate can also use the return value of the Shell function.
' Shell runs a new instance of notepad.
notepadID = Shell("C:\WINNT\NOTEPAD.EXE", AppWinStyle.MinimizedNoFocus)
' Activate the new instance of notepad.
AppActivate(notepadID)
```

Smart Device Developer Notes

This function is not supported.

Requirements

Namespace: Microsoft.VisualBasic

Module: Interaction

Assembly: Microsoft.Visual Basic .NET Runtime (in MicrosoftVisualBasic.dll)

See Also

Shell Function | ArgumentException Class

Asc, AscW Functions

Returns an **Integer** value representing the character code corresponding to a character.

```
Public Overloads Function Asc(ByVal String As Char) As Integer
Public Overloads Function AscW(ByVal String As Char) As Integer
```

-or-

```
Public Overloads Function Asc(ByVal String As String) As Integer
Public Overloads Function AscW(ByVal String As String) As Integer
```

Parameter

String
Required. Any valid **Char** or **String** expression. If *String* is a **String** expression, only the first character of the string is used for input. If *String* is **Nothing** or contains no characters, an **ArgumentException** error occurs.

Exceptions/Errors

Exception type	Error number	Condition
ArgumentException	5	*String* is not specified or is zero length.

Remarks

Asc returns the *code point*, or character code, for the input character. This can be 0 through 255 for single-byte character set (SBCS) values and -32768 through 32767 for double-byte character set (DBCS) values. The returned value depends on the code page for the current thread, which is contained in the **ANSICodePage** property of the **TextInfo** class. **TextInfo.ANSICodePage** can be obtained by specifying **System.Globalization.CultureInfo.CurrentCulture.TextInfo.ANSICodePage**.

AscW returns the Unicode code point for the input character. This can be 0 through 65535. The returned value is independent of the culture and code page settings for the current thread.

> **Note** The **AscB** function of earlier versions of Visual Basic returns a code for a byte, rather than a character. It is used primarily for converting strings in double-byte character set (DBCS) applications. All Visual Basic .NET strings are in Unicode, and **AscB** is no longer supported.

Example

This example uses the **Asc** function to return **Integer** character codes corresponding to the first letter in each string.

```
Dim MyInt As Integer
MyInt = Asc("A")   ' MyInt is set to 65.
MyInt = Asc("a")   ' MyInt is set to 97.
MyInt = Asc("Apple")   ' MyInt is set to 65.
```

Requirements

Namespace: Microsoft.VisualBasic

Module: Strings

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Chr, ChrW Functions | Conversion Functions | Type Conversion Functions | System.Globalization Namespace | CultureInfo Class | TextInfo.ANSICodePage Property | ArgumentException Class

Asc, AscW Functions

Returns an **Integer** value representing the character code corresponding to a character.

```
Public Overloads Function Asc(ByVal String As Char) As Integer
Public Overloads Function AscW(ByVal String As Char) As Integer
```

-or-

```
Public Overloads Function Asc(ByVal String As String) As Integer
Public Overloads Function AscW(ByVal String As String) As Integer
```

Parameter

String
Required. Any valid **Char** or **String** expression. If *String* is a **String** expression, only the first character of the string is used for input. If *String* is **Nothing** or contains no characters, an **ArgumentException** error occurs.

Exceptions/Errors

Exception type	Error number	Condition
ArgumentException	5	*String* is not specified or is zero length.

Remarks

Asc returns the *code point*, or character code, for the input character. This can be 0 through 255 for single-byte character set (SBCS) values and -32768 through 32767 for double-byte character set (DBCS) values. The returned value depends on the code page for the current thread, which is contained in the **ANSICodePage** property of the **TextInfo** class. **TextInfo.ANSICodePage** can be obtained by specifying **System.Globalization.CultureInfo.CurrentCulture.TextInfo.ANSICodePage**.

AscW returns the Unicode code point for the input character. This can be 0 through 65535. The returned value is independent of the culture and code page settings for the current thread.

> **Note** The **AscB** function of earlier versions of Visual Basic returns a code for a byte, rather than a character. It is used primarily for converting strings in double-byte character set (DBCS) applications. All Visual Basic .NET strings are in Unicode, and **AscB** is no longer supported.

Example

This example uses the **Asc** function to return **Integer** character codes corresponding to the first letter in each string.

```
Dim MyInt As Integer
MyInt = Asc("A")   ' MyInt is set to 65.
MyInt = Asc("a")   ' MyInt is set to 97.
MyInt = Asc("Apple")   ' MyInt is set to 65.
```

Requirements

Namespace: Microsoft.VisualBasic

Module: Strings

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Chr, ChrW Functions | Conversion Functions | Type Conversion Functions | System.Globalization Namespace | CultureInfo Class | TextInfo.ANSICodePage Property | ArgumentException Class

Math Functions

The math functions in Visual Basic 6 have been replaced by equivalent methods in the **System.Math** class of the .NET Framework.

Remarks

The .NET Framework math methods are functionally identical to their Visual Basic 6 counterparts, although some have slightly different names. For example, the .NET Framework equivalent of the Visual Basic 6 Atn function is **Atan**. The following table lists the Visual Basic 6 math function names and the equivalent .NET Framework methods.

Visual Basic 6 function	Visual Basic .NET method	Description
Abs	Math.Abs Method	Returns the absolute value of a specified number.
Atn	Math.Atan Method	Returns a **Double** value containing the angle whose tangent is the specified number.
Cos	Math.Cos Method	Returns a **Double** value containing the cosine of the specified angle.
Exp	Math.Exp Method	Returns a **Double** value containing e (the base of natural logarithms) raised to the specified power.
Log	Math.Log Method	Returns a **Double** value containing the logarithm of a specified number. This method is overloaded and can return either the natural (base e) logarithm of a specified number or the logarithm of a specified number in a specified base.
Round	Math.Round Method	Returns a **Double** value containing the number nearest the specified value. Additional round functions are available as methods of the intrinsic types such as Decimal.Round Method.
Sgn	Math.Sign Method	Returns an **Integer** value indicating the sign of a number.
Sin	Math.Sin Method	Returns a **Double** value specifying the sine of an angle.
Sqr	Math.Sqrt Method	Returns a **Double** value specifying the square root of a number.
Tan	Math.Tan Method	Returns a **Double** value containing the tangent of an angle.

In addition, the .NET Framework math class provides constants and other static methods for trigonometric, logarithmic, and other common mathematical functions. All of these can be used in a Visual Basic program.

To use these functions without qualification, import the **System.Math** namespace into your project by adding the following code to the top of the source code:

```
Imports System.Math
```

Requirements

Class: Math Class

Abs Example

This example uses the **Abs** method of the **Math** class to compute the absolute value of a number.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim MyNumber As Double
```

```
MyNumber = Abs(50.3)    ' Returns 50.3.
MyNumber = Abs(-50.3)   ' Returns 50.3.
```

Atan Example

This example uses the **Atan** method of the **Math** class to calculate the value of pi.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim pi As Double
pi = 4 * Atan(1)   ' Calculate the value of pi.
```

Cos Example

This example uses the **Cos** method of the **Math** class to return the cosine of an angle.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim MyAngle, MySecant As Double
MyAngle = 1.3   ' Define angle in radians.
MySecant = 1 / Cos(MyAngle)   ' Calculate secant.
```

Exp Example

This example uses the **Exp** method of the **Math** class to return *e* raised to a power.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim MyAngle, MyHSin As Double
' Define angle in radians.
MyAngle = 1.3
' Calculate hyperbolic sine.
MyHSin = (Exp(MyAngle) - Exp(-1 * MyAngle)) / 2
```

Log Example

This example uses the **Log** method of the **Math** class to return the natural logarithm of a number.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim MyAngle, MyLog As Double
' Define angle in radians.
MyAngle = 1.3
' Calculate inverse hyperbolic sine.
MyLog = Log(MyAngle + Sqrt(MyAngle * MyAngle + 1))
```

Round Example

This example uses the **Round** method of the **Math** class to round a number to the nearest integer.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim MyVar1 As Double = 2.8
Dim MyVar2 As Double
MyVar2 =Round(MyVar1)   ' Returns 3.
```

Sign Example

This example uses the **Sign** method of the **Math** class to determine the sign of a number.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim MyVar1, MyVar2, MyVar3 As Double
Dim MySign As Integer
MyVar1 = 12
MyVar2 = -2.4
MyVar3 = 0
MySign = Sign(MyVar1)   ' Returns 1.
MySign = Sign(MyVar2)   ' Returns -1.
MySign = Sign(MyVar3)   ' Returns 0.
```

Sin Example

This example uses the **Sin** method of the **Math** class to return the sine of an angle.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim MyAngle, MyCosecant As Double
MyAngle = 1.3   ' Define angle in radians.
MyCosecant = 1 / Sin(MyAngle)   ' Calculate cosecant.
```

Sqrt Example

This example uses the **Sqrt** method of the **Math** class to calculate the square root of a number.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim MySqr As Double
MySqr = Sqrt(4)    ' Returns 2.
MySqr = Sqrt(23)   ' Returns 4.79583152331272.
MySqr = Sqrt(0)    ' Returns 0.
MySqr = Sqrt(-4)   ' Returns NaN (not a number).
```

Tan Example

This example uses the **Tan** method of the **Math** class to return the tangent of an angle.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim MyAngle, MyCotangent As Double
MyAngle = 1.3   ' Define angle in radians.
MyCotangent = 1 / Tan(MyAngle)   ' Calculate cotangent.
```

Requirements

Namespace: System.Math

Assembly: mscorlib (in mscorlib.dll)

See Also

Rnd Function | Randomize Statement | Derived Math Functions | Double.NaN Field

Beep Function

Sounds a tone through the computer's speaker.

```
Public Sub Beep()
```

Remarks

The pitch and duration of the beep depend on your hardware and system software, and therefore vary among computers.

> **Note** The **Beep** function requires UIPermission at the SafeTopLevelWindows level, which may affect its execution in partial trust situations. For more information, see Requesting Permissions and UIPermission Class.

Example

This example uses the **Beep** function to sound a long, uninterrupted tone through the computer's speaker.

```
Dim I As Integer
For I = 1 To 100   ' Loop 100 times.
   Beep   ' Sound a tone.
Next I
```

Requirements

Namespace: Microsoft.VisualBasic

Module: Interaction

Assembly: Microsoft.Visual Basic .NET Runtime (in MicrosoftVisualBasic.dll)

See Also

Visual Basic Run-Time Library Members

CallByName Function

Executes a method on an object, or sets or returns a property on an object.

```
Public Function CallByName( _
   ByVal Object As System.Object, _
   ByVal ProcName As String, _
   ByVal UseCallType As CallType, _
   ByVal ParamArrayArgs() As Object _
) As Object
```

Parameters

Object
Required. **Object**. A pointer to the object exposing the property or method.
ProcName
Required. **String**. A string expression containing the name of the property or method on the object.
UseCallType
Required. An enumeration member of type **Microsoft.VisualBasic.CallType** representing the type of procedure being called. The value of **CallType** can be **Method**, **Get**, or **Set**.
ParamArrayArgs()
Optional. **ParamArray**. A parameter array containing the arguments to be passed to the property or method being called.

Exceptions/Errors

Exception type	Error number	Condition
ArgumentException	5	Invalid *UseCallType* value; must be **Method**, **Get**, or **Set.**

Remarks

The **CallByName** function is used at runtime to get a property, set a property, or invoke a method.

Example

In the following example, the first line uses **CallByName** to set the **Text** property of a text box, the second line retrieves the value of the **Text** property, and the third line invokes the **Move** method to move the text box.

```
Imports Microsoft.VisualBasic.CallType
' Imports statements must be at the top of a module.
...
Sub TestCallByName1()
'  Set a property.
   CallByName(TextBox1, "Text", CallType.Set, "New Text")
'  Retrieve the value of a property.
   MsgBox(CallByName(TextBox1, "Text", CallType.Get))
'  Call a method.
   CallByName(TextBox1, "Hide", CallType.Method)
End Sub
```

The next example uses the **CallByName** function to invoke the **Add** and **Item** methods of a collection object.

```
Public Sub TestCallByName2()
   Dim col As New Collection()
'  Store the string "Item One" in a collection by
'  calling the Add method.
   CallByName(col, "Add", CallType.Method, "Item One")
'  Retrieve the first entry from the collection using the
'  Item property and display it using MsgBox().
   MsgBox(CallByName(col, "Item", CallType.Get, 1))
End Sub
```

Smart Device Developer Notes

This function is not supported.

Requirements

Namespace: Microsoft.VisualBasic

Module: Interaction

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

CallType Enumeration | Parameter Arrays | Calling a Property or Method Using a String Name | ArgumentException Class

Type Conversion Functions

These functions are compiled inline, meaning the conversion code is part of the code that evaluates the expression. Execution is faster because there is no call to a procedure to accomplish the conversion. Each function coerces an expression to a specific data type.

```
CBool(expression)
CByte(expression)
CChar(expression)
CDate(expression)
CDbl(expression)
CDec(expression)
CInt(expression)
CLng(expression)
CObj(expression)
CShort(expression)
CSng(expression)
CStr(expression)
```

Part

expression
Required. Any **String** expression or numeric expression.

Return Types

The function name determines the return type, as shown in the following table:

Function name	Return type	Range for *expression* argument
CBool	**Boolean**	Any valid **String** or numeric expression.
CByte	**Byte**	0 through 255; fractions are rounded.
CChar	**Char**	Any valid **String** expression; value can be 0 through 65535.
CDate	**Date**	Any valid representation of a date and time.
CDbl	**Double**	-1.79769313486231E+308 through -4.94065645841247E-324 for negative values; 4.94065645841247E-324 through 1.79769313486231E+308 for positive values.
CDec	**Decimal**	+/-79,228,162,514,264,337,593,543,950,335 for zero-scaled numbers, that is, numbers with no decimal places. For numbers with 28 decimal places, the range is +/-7.9228162514264337593543950335. The smallest possible non-zero number is 0.0000000000000000000000000001.
CInt	**Integer**	-2,147,483,648 through 2,147,483,647; fractions are rounded.
CLng	**Long**	-9,223,372,036,854,775,808 through 9,223,372,036,854,775,807; fractions are rounded.
CObj	**Object**	Any valid expression.
CShort	**Short**	-32,768 through 32,767; fractions are rounded.
CSng	**Single**	-3.402823E+38 through -1.401298E-45 for negative values; 1.401298E-45 through 3.402823E+38 for positive values.
CStr	**String**	Returns for **CStr** depend on the *expression* argument. See Returns for CStr.

Remarks

If the *expression* passed to the function is outside the range of the data type to which it is being converted, an error occurs.

In general, you can use the data type conversion functions to force the result of some operation to a particular data type rather than the default data type. For example, use **CDec** to force decimal arithmetic in cases where single-precision, double-precision, or integer arithmetic normally would occur.

When the fractional part is exactly 0.5, **CInt** and **CLng** always round it to the nearest even number. For example, 0.5 rounds to 0

and 1.5 rounds to 2. **CInt** and **CLng** differ from the **Fix** and **Int** functions, which truncate, rather than round, the fractional part of a number. Also, **Fix** and **Int** always return a value of the same type as is passed in.

Use the **IsDate** function to determine if a value can be converted to a date and time. **CDate** recognizes date literals and time literals as well as numbers that fall within the range of acceptable dates.

CDate recognizes date formats according to the locale setting of your system. You must provide the day, month, and year in the correct order for your locale, or the date may not be interpreted correctly. A long date format is not recognized if it contains a day-of-the-week string, such as "Wednesday".

The **Date** data type always contains both date and time information. For purposes of type conversion, Visual Basic .NET considers 1/1/1 (January 1 of the year 1) to be a neutral value for the date, and 00:00:00 (midnight) to be a neutral value for the time. If you convert a **Date** value to a string, **CStr** does not include neutral values in the resulting string. For example, if you convert `#January 1, 0001 9:30:00#` to a string, the result is "9:30:00 AM"; the date information is suppressed. However, the date information is still present in the original **Date** value and can be recovered with functions such as **DatePart**.

The **CType** function takes a second argument, *typename*, and coerces *expression* to *typename*, where *typename* can be any data type, structure, class, or interface. For more information, see CType Function.

CBool Example

This example uses the **CBool** function to convert expressions to **Boolean** values. If an expression evaluates to a nonzero value, **CBool** returns **True**; otherwise, it returns **False**.

```
Dim A, B, C As Integer
Dim Check As Boolean
A = 5
B = 5
Check = CBool(A = B)   ' Check is set to True.
' ...
C = 0
Check = CBool(C)   ' Check is set to False.
```

CByte Example

This example uses the **CByte** function to convert an expression to a **Byte**.

```
Dim MyDouble As Double
Dim MyByte As Byte
MyDouble = 125.5678
MyByte = CByte(MyDouble)   ' MyByte is set to 126.
```

CChar Example

This example uses the **CChar** function to convert a the first character of a **String** expression to a **Char** type.

```
Dim MyString As String
Dim MyChar As Char
MyString = "BCD"   ' CChar converts only first character of string.
MyChar = CChar(MyString)   ' MyChar is set to "B".
```

The input argument to **CChar** must be of data type **String**. You cannot use **CChar** to convert a number to a character, because **CChar** cannot accept a numeric data type. This example obtains a number representing a code point (character code) and converts it to the corresponding character. It uses **InputBox** to obtain the string of digits, **CInt** to convert the string to type **Integer**, and **ChrW** to convert the number to type **Char**.

```
Dim MyDigits As String   ' Input string of digits to be converted.
Dim CodePoint As Integer   ' Number to be represented as a character.
Dim MyChar As Char
MyDigits = InputBox("Enter code point of character:")
CodePoint = CInt(MyDigits)   ' Convert entire string to Integer.
MyChar = ChrW(CodePoint)   ' MyChar is set to Char value of code point.
```

CDate Example

This example uses the **CDate** function to convert strings to **Date** values. In general, hard-coding dates and times as strings (as shown in this example) is not recommended. Use date literals and time literals, such as #Feb 12, 1969# and #4:45:23 PM#, instead.

```
Dim MyDateString, MyTimeString As String
Dim MyDate, MyTime As Date
MyDateString = "February 12, 1969"
MyTimeString = "4:35:47 PM"
' ...
MyDate = CDate(MyDateString)   ' Convert to Date data type.
MyTime = CDate(MyTimeString)   ' Convert to Date data type.
```

CDbl Example

This example uses the **CDbl** function to convert an expression to **Double**.

```
Dim MyDec As Decimal
Dim MyDouble As Double
MyDec = 234.456784D    ' Literal type character D makes MyDec a Decimal.
MyDouble = CDbl(MyDec * 8.2D * 0.01D)   ' Convert result to a Double.
```

CDec Example

This example uses the **CDec** function to convert a numeric value to **Decimal**.

```
Dim MyDouble As Double
Dim MyDecimal As Decimal
MyDouble = 10000000.0587
MyDecimal = CDec(MyDouble)   ' Convert to Decimal.
```

CInt Example

This example uses the **CInt** function to convert a value to **Integer**.

```
Dim MyDouble As Double
Dim MyInt As Integer
MyDouble = 2345.5678
MyInt = CInt(MyDouble)   ' MyInt is set to 2346.
```

CLng Example

This example uses the **CLng** function to convert values to **Long**.

```
Dim MyDbl1, MyDbl2 As Double
Dim MyLong1, MyLong2 As Long
MyDbl1 = 25427.45
MyDbl2 = 25427.55
MyLong1 = CLng(MyDbl1)   ' MyLong1 contains 25427.
MyLong2 = CLng(MyDbl2)   ' MyLong2 contains 25428.
```

CObj Example

This example uses the **CObj** function to convert a numeric value to **Object**. The **Object** variable itself contains only a four-byte pointer, which points to the **Double** value assigned to it.

```
Dim MyDouble As Double
Dim MyObject As Object
```

```
MyDouble = 2.7182818284
MyObject = CObj(MyDouble)   ' Double value is pointed to by MyObject.
```

CShort Example

This example uses the **CShort** function to convert a numeric value to **Short**.

```
Dim MyByte as Byte
Dim MyShort as Short
MyByte = 100
MyShort = CShort(MyByte)   ' Convert to Short.
```

CSng Example

This example uses the **CSng** function to convert values to **Single**.

```
Dim MyDouble1, MyDouble2 As Double
Dim MySingle1, MySingle2 As Single
MyDouble1 = 75.3421105
MyDouble2 = 75.3421567
MySingle1 = CSng(MyDouble1)   ' MySingle1 is set to 75.34211.
MySingle2 = CSng(MyDouble2)   ' MySingle2 is set to 75.34216.
```

CStr Example

This example uses the **CStr** function to convert a numeric value to **String**.

```
Dim MyDouble As Double
Dim MyString As String
MyDouble = 437.324
MyString = CStr(MyDouble)   ' MyString is set to "437.324".
```

This example uses the **CStr** function to convert **Date** values to **String** values.

```
Dim MyDate As Date
Dim MyString As String
' ...
MyDate = #February 12, 1969 00:00:00#   ' INVALID format.
' Date literals must be in the format #m/d/yyyy# or they are invalid.
' ...
MyDate = #2/12/69 00:00:00#   ' Time is midnight.
' The neutral time value of 00:00:00 is suppressed in the conversion.
MyString = CStr(MyDate)   ' MyString is set to "2/12/1969".
' ...
MyDate = #2/12/69 00:00:01#   ' Time is one second past midnight.
' The time component becomes part of the converted value.
MyString = CStr(MyDate)   ' MyString is set to "2/12/1969 12:00:01 AM".
```

CStr always renders a **Date** value in the standard short format for the current locale, for example, "6/15/2003 4:35:47 PM".

See Also

Conversion Functions | CType Function | Asc, AscW Functions | Chr, ChrW Functions | Format Function | Hex Function | Oct Function | Str Function | Val Function | DatePart Function | Returns for CStr | Type Conversions

Type Conversion Functions

These functions are compiled inline, meaning the conversion code is part of the code that evaluates the expression. Execution is faster because there is no call to a procedure to accomplish the conversion. Each function coerces an expression to a specific data type.

```
CBool(expression)
CByte(expression)
CChar(expression)
CDate(expression)
CDbl(expression)
CDec(expression)
CInt(expression)
CLng(expression)
CObj(expression)
CShort(expression)
CSng(expression)
CStr(expression)
```

Part

expression
Required. Any **String** expression or numeric expression.

Return Types

The function name determines the return type, as shown in the following table:

Function name	Return type	Range for *expression* argument
CBool	**Boolean**	Any valid **String** or numeric expression.
CByte	**Byte**	0 through 255; fractions are rounded.
CChar	**Char**	Any valid **String** expression; value can be 0 through 65535.
CDate	**Date**	Any valid representation of a date and time.
CDbl	**Double**	-1.79769313486231E+308 through -4.94065645841247E-324 for negative values; 4.94065645841247E-324 through 1.79769313486231E+308 for positive values.
CDec	**Decimal**	+/-79,228,162,514,264,337,593,543,950,335 for zero-scaled numbers, that is, numbers with no decimal places. For numbers with 28 decimal places, the range is +/-7.9228162514264337593543950335. The smallest possible non-zero number is 0.0000000000000000000000000001.
CInt	**Integer**	-2,147,483,648 through 2,147,483,647; fractions are rounded.
CLng	**Long**	-9,223,372,036,854,775,808 through 9,223,372,036,854,775,807; fractions are rounded.
CObj	**Object**	Any valid expression.
CShort	**Short**	-32,768 through 32,767; fractions are rounded.
CSng	**Single**	-3.402823E+38 through -1.401298E-45 for negative values; 1.401298E-45 through 3.402823E+38 for positive values.
CStr	**String**	Returns for **CStr** depend on the *expression* argument. See Returns for CStr.

Remarks

If the *expression* passed to the function is outside the range of the data type to which it is being converted, an error occurs.

In general, you can use the data type conversion functions to force the result of some operation to a particular data type rather than the default data type. For example, use **CDec** to force decimal arithmetic in cases where single-precision, double-precision, or integer arithmetic normally would occur.

When the fractional part is exactly 0.5, **CInt** and **CLng** always round it to the nearest even number. For example, 0.5 rounds to 0

and 1.5 rounds to 2. **CInt** and **CLng** differ from the **Fix** and **Int** functions, which truncate, rather than round, the fractional part of a number. Also, **Fix** and **Int** always return a value of the same type as is passed in.

Use the **IsDate** function to determine if a value can be converted to a date and time. **CDate** recognizes date literals and time literals as well as numbers that fall within the range of acceptable dates.

CDate recognizes date formats according to the locale setting of your system. You must provide the day, month, and year in the correct order for your locale, or the date may not be interpreted correctly. A long date format is not recognized if it contains a day-of-the-week string, such as "Wednesday".

The **Date** data type always contains both date and time information. For purposes of type conversion, Visual Basic .NET considers 1/1/1 (January 1 of the year 1) to be a neutral value for the date, and 00:00:00 (midnight) to be a neutral value for the time. If you convert a **Date** value to a string, **CStr** does not include neutral values in the resulting string. For example, if you convert `#January 1, 0001 9:30:00#` to a string, the result is "9:30:00 AM"; the date information is suppressed. However, the date information is still present in the original **Date** value and can be recovered with functions such as **DatePart**.

The **CType** function takes a second argument, *typename*, and coerces *expression* to *typename*, where *typename* can be any data type, structure, class, or interface. For more information, see CType Function.

CBool Example

This example uses the **CBool** function to convert expressions to **Boolean** values. If an expression evaluates to a nonzero value, **CBool** returns **True**; otherwise, it returns **False**.

```
Dim A, B, C As Integer
Dim Check As Boolean
A = 5
B = 5
Check = CBool(A = B)   ' Check is set to True.
' ...
C = 0
Check = CBool(C)   ' Check is set to False.
```

CByte Example

This example uses the **CByte** function to convert an expression to a **Byte**.

```
Dim MyDouble As Double
Dim MyByte As Byte
MyDouble = 125.5678
MyByte = CByte(MyDouble)   ' MyByte is set to 126.
```

CChar Example

This example uses the **CChar** function to convert a the first character of a **String** expression to a **Char** type.

```
Dim MyString As String
Dim MyChar As Char
MyString = "BCD"   ' CChar converts only first character of string.
MyChar = CChar(MyString)   ' MyChar is set to "B".
```

The input argument to **CChar** must be of data type **String**. You cannot use **CChar** to convert a number to a character, because **CChar** cannot accept a numeric data type. This example obtains a number representing a code point (character code) and converts it to the corresponding character. It uses **InputBox** to obtain the string of digits, **CInt** to convert the string to type **Integer**, and **ChrW** to convert the number to type **Char**.

```
Dim MyDigits As String   ' Input string of digits to be converted.
Dim CodePoint As Integer   ' Number to be represented as a character.
Dim MyChar As Char
MyDigits = InputBox("Enter code point of character:")
CodePoint = CInt(MyDigits)   ' Convert entire string to Integer.
MyChar = ChrW(CodePoint)   ' MyChar is set to Char value of code point.
```

CDate Example

This example uses the **CDate** function to convert strings to **Date** values. In general, hard-coding dates and times as strings (as shown in this example) is not recommended. Use date literals and time literals, such as #Feb 12, 1969# and #4:45:23 PM#, instead.

```
Dim MyDateString, MyTimeString As String
Dim MyDate, MyTime As Date
MyDateString = "February 12, 1969"
MyTimeString = "4:35:47 PM"
' ...
MyDate = CDate(MyDateString)   ' Convert to Date data type.
MyTime = CDate(MyTimeString)   ' Convert to Date data type.
```

CDbl Example

This example uses the **CDbl** function to convert an expression to **Double**.

```
Dim MyDec As Decimal
Dim MyDouble As Double
MyDec = 234.456784D    ' Literal type character D makes MyDec a Decimal.
MyDouble = CDbl(MyDec * 8.2D * 0.01D)   ' Convert result to a Double.
```

CDec Example

This example uses the **CDec** function to convert a numeric value to **Decimal**.

```
Dim MyDouble As Double
Dim MyDecimal As Decimal
MyDouble = 10000000.0587
MyDecimal = CDec(MyDouble)   ' Convert to Decimal.
```

CInt Example

This example uses the **CInt** function to convert a value to **Integer**.

```
Dim MyDouble As Double
Dim MyInt As Integer
MyDouble = 2345.5678
MyInt = CInt(MyDouble)   ' MyInt is set to 2346.
```

CLng Example

This example uses the **CLng** function to convert values to **Long**.

```
Dim MyDbl1, MyDbl2 As Double
Dim MyLong1, MyLong2 As Long
MyDbl1 = 25427.45
MyDbl2 = 25427.55
MyLong1 = CLng(MyDbl1)   ' MyLong1 contains 25427.
MyLong2 = CLng(MyDbl2)   ' MyLong2 contains 25428.
```

CObj Example

This example uses the **CObj** function to convert a numeric value to **Object**. The **Object** variable itself contains only a four-byte pointer, which points to the **Double** value assigned to it.

```
Dim MyDouble As Double
Dim MyObject As Object
```

```
MyDouble = 2.7182818284
MyObject = CObj(MyDouble)   ' Double value is pointed to by MyObject.
```

CShort Example

This example uses the **CShort** function to convert a numeric value to **Short**.

```
Dim MyByte as Byte
Dim MyShort as Short
MyByte = 100
MyShort = CShort(MyByte)   ' Convert to Short.
```

CSng Example

This example uses the **CSng** function to convert values to **Single**.

```
Dim MyDouble1, MyDouble2 As Double
Dim MySingle1, MySingle2 As Single
MyDouble1 = 75.3421105
MyDouble2 = 75.3421567
MySingle1 = CSng(MyDouble1)   ' MySingle1 is set to 75.34211.
MySingle2 = CSng(MyDouble2)   ' MySingle2 is set to 75.34216.
```

CStr Example

This example uses the **CStr** function to convert a numeric value to **String**.

```
Dim MyDouble As Double
Dim MyString As String
MyDouble = 437.324
MyString = CStr(MyDouble)   ' MyString is set to "437.324".
```

This example uses the **CStr** function to convert **Date** values to **String** values.

```
Dim MyDate As Date
Dim MyString As String
' ...
MyDate = #February 12, 1969 00:00:00#   ' INVALID format.
' Date literals must be in the format #m/d/yyyy# or they are invalid.
' ...
MyDate = #2/12/69 00:00:00#   ' Time is midnight.
' The neutral time value of 00:00:00 is suppressed in the conversion.
MyString = CStr(MyDate)   ' MyString is set to "2/12/1969".
' ...
MyDate = #2/12/69 00:00:01#   ' Time is one second past midnight.
' The time component becomes part of the converted value.
MyString = CStr(MyDate)   ' MyString is set to "2/12/1969 12:00:01 AM".
```

CStr always renders a **Date** value in the standard short format for the current locale, for example, "6/15/2003 4:35:47 PM".

See Also

Conversion Functions | CType Function | Asc, AscW Functions | Chr, ChrW Functions | Format Function | Hex Function | Oct Function | Str Function | Val Function | DatePart Function | Returns for CStr | Type Conversions

Type Conversion Functions

These functions are compiled inline, meaning the conversion code is part of the code that evaluates the expression. Execution is faster because there is no call to a procedure to accomplish the conversion. Each function coerces an expression to a specific data type.

```
CBool(expression)
CByte(expression)
CChar(expression)
CDate(expression)
CDbl(expression)
CDec(expression)
CInt(expression)
CLng(expression)
CObj(expression)
CShort(expression)
CSng(expression)
CStr(expression)
```

Part

expression
Required. Any **String** expression or numeric expression.

Return Types

The function name determines the return type, as shown in the following table:

Function name	Return type	Range for *expression* argument
CBool	**Boolean**	Any valid **String** or numeric expression.
CByte	**Byte**	0 through 255; fractions are rounded.
CChar	**Char**	Any valid **String** expression; value can be 0 through 65535.
CDate	**Date**	Any valid representation of a date and time.
CDbl	**Double**	-1.79769313486231E+308 through -4.94065645841247E-324 for negative values; 4.94065645841247E-324 through 1.79769313486231E+308 for positive values.
CDec	**Decimal**	+/-79,228,162,514,264,337,593,543,950,335 for zero-scaled numbers, that is, numbers with no decimal places. For numbers with 28 decimal places, the range is +/-7.9228162514264337593543950335. The smallest possible non-zero number is 0.0000000000000000000000000001.
CInt	**Integer**	-2,147,483,648 through 2,147,483,647; fractions are rounded.
CLng	**Long**	-9,223,372,036,854,775,808 through 9,223,372,036,854,775,807; fractions are rounded.
CObj	**Object**	Any valid expression.
CShort	**Short**	-32,768 through 32,767; fractions are rounded.
CSng	**Single**	-3.402823E+38 through -1.401298E-45 for negative values; 1.401298E-45 through 3.402823E+38 for positive values.
CStr	**String**	Returns for **CStr** depend on the *expression* argument. See Returns for CStr.

Remarks

If the *expression* passed to the function is outside the range of the data type to which it is being converted, an error occurs.

In general, you can use the data type conversion functions to force the result of some operation to a particular data type rather than the default data type. For example, use **CDec** to force decimal arithmetic in cases where single-precision, double-precision, or integer arithmetic normally would occur.

When the fractional part is exactly 0.5, **CInt** and **CLng** always round it to the nearest even number. For example, 0.5 rounds to 0

and 1.5 rounds to 2. **CInt** and **CLng** differ from the **Fix** and **Int** functions, which truncate, rather than round, the fractional part of a number. Also, **Fix** and **Int** always return a value of the same type as is passed in.

Use the **IsDate** function to determine if a value can be converted to a date and time. **CDate** recognizes date literals and time literals as well as numbers that fall within the range of acceptable dates.

CDate recognizes date formats according to the locale setting of your system. You must provide the day, month, and year in the correct order for your locale, or the date may not be interpreted correctly. A long date format is not recognized if it contains a day-of-the-week string, such as "Wednesday".

The **Date** data type always contains both date and time information. For purposes of type conversion, Visual Basic .NET considers 1/1/1 (January 1 of the year 1) to be a neutral value for the date, and 00:00:00 (midnight) to be a neutral value for the time. If you convert a **Date** value to a string, **CStr** does not include neutral values in the resulting string. For example, if you convert `#January 1, 0001 9:30:00#` to a string, the result is "9:30:00 AM"; the date information is suppressed. However, the date information is still present in the original **Date** value and can be recovered with functions such as **DatePart**.

The **CType** function takes a second argument, *typename*, and coerces *expression* to *typename*, where *typename* can be any data type, structure, class, or interface. For more information, see CType Function.

CBool Example

This example uses the **CBool** function to convert expressions to **Boolean** values. If an expression evaluates to a nonzero value, **CBool** returns **True**; otherwise, it returns **False**.

```
Dim A, B, C As Integer
Dim Check As Boolean
A = 5
B = 5
Check = CBool(A = B)   ' Check is set to True.
' ...
C = 0
Check = CBool(C)   ' Check is set to False.
```

CByte Example

This example uses the **CByte** function to convert an expression to a **Byte**.

```
Dim MyDouble As Double
Dim MyByte As Byte
MyDouble = 125.5678
MyByte = CByte(MyDouble)   ' MyByte is set to 126.
```

CChar Example

This example uses the **CChar** function to convert a the first character of a **String** expression to a **Char** type.

```
Dim MyString As String
Dim MyChar As Char
MyString = "BCD"   ' CChar converts only first character of string.
MyChar = CChar(MyString)   ' MyChar is set to "B".
```

The input argument to **CChar** must be of data type **String**. You cannot use **CChar** to convert a number to a character, because **CChar** cannot accept a numeric data type. This example obtains a number representing a code point (character code) and converts it to the corresponding character. It uses **InputBox** to obtain the string of digits, **CInt** to convert the string to type **Integer**, and **ChrW** to convert the number to type **Char**.

```
Dim MyDigits As String   ' Input string of digits to be converted.
Dim CodePoint As Integer   ' Number to be represented as a character.
Dim MyChar As Char
MyDigits = InputBox("Enter code point of character:")
CodePoint = CInt(MyDigits)   ' Convert entire string to Integer.
MyChar = ChrW(CodePoint)   ' MyChar is set to Char value of code point.
```

CDate Example

This example uses the **CDate** function to convert strings to **Date** values. In general, hard-coding dates and times as strings (as shown in this example) is not recommended. Use date literals and time literals, such as #Feb 12, 1969# and #4:45:23 PM#, instead.

```
Dim MyDateString, MyTimeString As String
Dim MyDate, MyTime As Date
MyDateString = "February 12, 1969"
MyTimeString = "4:35:47 PM"
' ...
MyDate = CDate(MyDateString)   ' Convert to Date data type.
MyTime = CDate(MyTimeString)   ' Convert to Date data type.
```

CDbl Example

This example uses the **CDbl** function to convert an expression to **Double**.

```
Dim MyDec As Decimal
Dim MyDouble As Double
MyDec = 234.456784D    ' Literal type character D makes MyDec a Decimal.
MyDouble = CDbl(MyDec * 8.2D * 0.01D)   ' Convert result to a Double.
```

CDec Example

This example uses the **CDec** function to convert a numeric value to **Decimal**.

```
Dim MyDouble As Double
Dim MyDecimal As Decimal
MyDouble = 10000000.0587
MyDecimal = CDec(MyDouble)   ' Convert to Decimal.
```

CInt Example

This example uses the **CInt** function to convert a value to **Integer**.

```
Dim MyDouble As Double
Dim MyInt As Integer
MyDouble = 2345.5678
MyInt = CInt(MyDouble)   ' MyInt is set to 2346.
```

CLng Example

This example uses the **CLng** function to convert values to **Long**.

```
Dim MyDbl1, MyDbl2 As Double
Dim MyLong1, MyLong2 As Long
MyDbl1 = 25427.45
MyDbl2 = 25427.55
MyLong1 = CLng(MyDbl1)   ' MyLong1 contains 25427.
MyLong2 = CLng(MyDbl2)   ' MyLong2 contains 25428.
```

CObj Example

This example uses the **CObj** function to convert a numeric value to **Object**. The **Object** variable itself contains only a four-byte pointer, which points to the **Double** value assigned to it.

```
Dim MyDouble As Double
Dim MyObject As Object
```

```
MyDouble = 2.7182818284
MyObject = CObj(MyDouble)   ' Double value is pointed to by MyObject.
```

CShort Example

This example uses the **CShort** function to convert a numeric value to **Short**.

```
Dim MyByte as Byte
Dim MyShort as Short
MyByte = 100
MyShort = CShort(MyByte)   ' Convert to Short.
```

CSng Example

This example uses the **CSng** function to convert values to **Single**.

```
Dim MyDouble1, MyDouble2 As Double
Dim MySingle1, MySingle2 As Single
MyDouble1 = 75.3421105
MyDouble2 = 75.3421567
MySingle1 = CSng(MyDouble1)   ' MySingle1 is set to 75.34211.
MySingle2 = CSng(MyDouble2)   ' MySingle2 is set to 75.34216.
```

CStr Example

This example uses the **CStr** function to convert a numeric value to **String**.

```
Dim MyDouble As Double
Dim MyString As String
MyDouble = 437.324
MyString = CStr(MyDouble)   ' MyString is set to "437.324".
```

This example uses the **CStr** function to convert **Date** values to **String** values.

```
Dim MyDate As Date
Dim MyString As String
' ...
MyDate = #February 12, 1969 00:00:00#   ' INVALID format.
' Date literals must be in the format #m/d/yyyy# or they are invalid.
' ...
MyDate = #2/12/69 00:00:00#   ' Time is midnight.
' The neutral time value of 00:00:00 is suppressed in the conversion.
MyString = CStr(MyDate)   ' MyString is set to "2/12/1969".
' ...
MyDate = #2/12/69 00:00:01#   ' Time is one second past midnight.
' The time component becomes part of the converted value.
MyString = CStr(MyDate)   ' MyString is set to "2/12/1969 12:00:01 AM".
```

CStr always renders a **Date** value in the standard short format for the current locale, for example, "6/15/2003 4:35:47 PM".

See Also

Conversion Functions | CType Function | Asc, AscW Functions | Chr, ChrW Functions | Format Function | Hex Function | Oct Function | Str Function | Val Function | DatePart Function | Returns for CStr | Type Conversions

Type Conversion Functions

These functions are compiled inline, meaning the conversion code is part of the code that evaluates the expression. Execution is faster because there is no call to a procedure to accomplish the conversion. Each function coerces an expression to a specific data type.

```
CBool(expression)
CByte(expression)
CChar(expression)
CDate(expression)
CDbl(expression)
CDec(expression)
CInt(expression)
CLng(expression)
CObj(expression)
CShort(expression)
CSng(expression)
CStr(expression)
```

Part

expression
Required. Any **String** expression or numeric expression.

Return Types

The function name determines the return type, as shown in the following table:

Function name	Return type	Range for *expression* argument
CBool	**Boolean**	Any valid **String** or numeric expression.
CByte	**Byte**	0 through 255; fractions are rounded.
CChar	**Char**	Any valid **String** expression; value can be 0 through 65535.
CDate	**Date**	Any valid representation of a date and time.
CDbl	**Double**	-1.79769313486231E+308 through -4.94065645841247E-324 for negative values; 4.94065645841247E-324 through 1.79769313486231E+308 for positive values.
CDec	**Decimal**	+/-79,228,162,514,264,337,593,543,950,335 for zero-scaled numbers, that is, numbers with no decimal places. For numbers with 28 decimal places, the range is +/-7.9228162514264337593543950335. The smallest possible non-zero number is 0.0000000000000000000000000001.
CInt	**Integer**	-2,147,483,648 through 2,147,483,647; fractions are rounded.
CLng	**Long**	-9,223,372,036,854,775,808 through 9,223,372,036,854,775,807; fractions are rounded.
CObj	**Object**	Any valid expression.
CShort	**Short**	-32,768 through 32,767; fractions are rounded.
CSng	**Single**	-3.402823E+38 through -1.401298E-45 for negative values; 1.401298E-45 through 3.402823E+38 for positive values.
CStr	**String**	Returns for **CStr** depend on the *expression* argument. See Returns for CStr.

Remarks

If the *expression* passed to the function is outside the range of the data type to which it is being converted, an error occurs.

In general, you can use the data type conversion functions to force the result of some operation to a particular data type rather than the default data type. For example, use **CDec** to force decimal arithmetic in cases where single-precision, double-precision, or integer arithmetic normally would occur.

When the fractional part is exactly 0.5, **CInt** and **CLng** always round it to the nearest even number. For example, 0.5 rounds to 0

and 1.5 rounds to 2. **CInt** and **CLng** differ from the **Fix** and **Int** functions, which truncate, rather than round, the fractional part of a number. Also, **Fix** and **Int** always return a value of the same type as is passed in.

Use the **IsDate** function to determine if a value can be converted to a date and time. **CDate** recognizes date literals and time literals as well as numbers that fall within the range of acceptable dates.

CDate recognizes date formats according to the locale setting of your system. You must provide the day, month, and year in the correct order for your locale, or the date may not be interpreted correctly. A long date format is not recognized if it contains a day-of-the-week string, such as "Wednesday".

The **Date** data type always contains both date and time information. For purposes of type conversion, Visual Basic .NET considers 1/1/1 (January 1 of the year 1) to be a neutral value for the date, and 00:00:00 (midnight) to be a neutral value for the time. If you convert a **Date** value to a string, **CStr** does not include neutral values in the resulting string. For example, if you convert `#January 1, 0001 9:30:00#` to a string, the result is "9:30:00 AM"; the date information is suppressed. However, the date information is still present in the original **Date** value and can be recovered with functions such as **DatePart**.

The **CType** function takes a second argument, *typename*, and coerces *expression* to *typename*, where *typename* can be any data type, structure, class, or interface. For more information, see CType Function.

CBool Example

This example uses the **CBool** function to convert expressions to **Boolean** values. If an expression evaluates to a nonzero value, **CBool** returns **True**; otherwise, it returns **False**.

```
Dim A, B, C As Integer
Dim Check As Boolean
A = 5
B = 5
Check = CBool(A = B)   ' Check is set to True.
' ...
C = 0
Check = CBool(C)   ' Check is set to False.
```

CByte Example

This example uses the **CByte** function to convert an expression to a **Byte**.

```
Dim MyDouble As Double
Dim MyByte As Byte
MyDouble = 125.5678
MyByte = CByte(MyDouble)   ' MyByte is set to 126.
```

CChar Example

This example uses the **CChar** function to convert a the first character of a **String** expression to a **Char** type.

```
Dim MyString As String
Dim MyChar As Char
MyString = "BCD"   ' CChar converts only first character of string.
MyChar = CChar(MyString)   ' MyChar is set to "B".
```

The input argument to **CChar** must be of data type **String**. You cannot use **CChar** to convert a number to a character, because **CChar** cannot accept a numeric data type. This example obtains a number representing a code point (character code) and converts it to the corresponding character. It uses **InputBox** to obtain the string of digits, **CInt** to convert the string to type **Integer**, and **ChrW** to convert the number to type **Char**.

```
Dim MyDigits As String   ' Input string of digits to be converted.
Dim CodePoint As Integer   ' Number to be represented as a character.
Dim MyChar As Char
MyDigits = InputBox("Enter code point of character:")
CodePoint = CInt(MyDigits)   ' Convert entire string to Integer.
MyChar = ChrW(CodePoint)   ' MyChar is set to Char value of code point.
```

CDate Example

This example uses the **CDate** function to convert strings to **Date** values. In general, hard-coding dates and times as strings (as shown in this example) is not recommended. Use date literals and time literals, such as #Feb 12, 1969# and #4:45:23 PM#, instead.

```
Dim MyDateString, MyTimeString As String
Dim MyDate, MyTime As Date
MyDateString = "February 12, 1969"
MyTimeString = "4:35:47 PM"
' ...
MyDate = CDate(MyDateString)   ' Convert to Date data type.
MyTime = CDate(MyTimeString)   ' Convert to Date data type.
```

CDbl Example

This example uses the **CDbl** function to convert an expression to **Double**.

```
Dim MyDec As Decimal
Dim MyDouble As Double
MyDec = 234.456784D    ' Literal type character D makes MyDec a Decimal.
MyDouble = CDbl(MyDec * 8.2D * 0.01D)   ' Convert result to a Double.
```

CDec Example

This example uses the **CDec** function to convert a numeric value to **Decimal**.

```
Dim MyDouble As Double
Dim MyDecimal As Decimal
MyDouble = 10000000.0587
MyDecimal = CDec(MyDouble)   ' Convert to Decimal.
```

CInt Example

This example uses the **CInt** function to convert a value to **Integer**.

```
Dim MyDouble As Double
Dim MyInt As Integer
MyDouble = 2345.5678
MyInt = CInt(MyDouble)   ' MyInt is set to 2346.
```

CLng Example

This example uses the **CLng** function to convert values to **Long**.

```
Dim MyDbl1, MyDbl2 As Double
Dim MyLong1, MyLong2 As Long
MyDbl1 = 25427.45
MyDbl2 = 25427.55
MyLong1 = CLng(MyDbl1)   ' MyLong1 contains 25427.
MyLong2 = CLng(MyDbl2)   ' MyLong2 contains 25428.
```

CObj Example

This example uses the **CObj** function to convert a numeric value to **Object**. The **Object** variable itself contains only a four-byte pointer, which points to the **Double** value assigned to it.

```
Dim MyDouble As Double
Dim MyObject As Object
```

```
MyDouble = 2.7182818284
MyObject = CObj(MyDouble)   ' Double value is pointed to by MyObject.
```

CShort Example

This example uses the **CShort** function to convert a numeric value to **Short**.

```
Dim MyByte as Byte
Dim MyShort as Short
MyByte = 100
MyShort = CShort(MyByte)   ' Convert to Short.
```

CSng Example

This example uses the **CSng** function to convert values to **Single**.

```
Dim MyDouble1, MyDouble2 As Double
Dim MySingle1, MySingle2 As Single
MyDouble1 = 75.3421105
MyDouble2 = 75.3421567
MySingle1 = CSng(MyDouble1)   ' MySingle1 is set to 75.34211.
MySingle2 = CSng(MyDouble2)   ' MySingle2 is set to 75.34216.
```

CStr Example

This example uses the **CStr** function to convert a numeric value to **String**.

```
Dim MyDouble As Double
Dim MyString As String
MyDouble = 437.324
MyString = CStr(MyDouble)   ' MyString is set to "437.324".
```

This example uses the **CStr** function to convert **Date** values to **String** values.

```
Dim MyDate As Date
Dim MyString As String
' ...
MyDate = #February 12, 1969 00:00:00#   ' INVALID format.
' Date literals must be in the format #m/d/yyyy# or they are invalid.
' ...
MyDate = #2/12/69 00:00:00#   ' Time is midnight.
' The neutral time value of 00:00:00 is suppressed in the conversion.
MyString = CStr(MyDate)   ' MyString is set to "2/12/1969".
' ...
MyDate = #2/12/69 00:00:01#   ' Time is one second past midnight.
' The time component becomes part of the converted value.
MyString = CStr(MyDate)   ' MyString is set to "2/12/1969 12:00:01 AM".
```

CStr always renders a **Date** value in the standard short format for the current locale, for example, "6/15/2003 4:35:47 PM".

See Also

Conversion Functions | CType Function | Asc, AscW Functions | Chr, ChrW Functions | Format Function | Hex Function | Oct Function | Str Function | Val Function | DatePart Function | Returns for CStr | Type Conversions

Type Conversion Functions

These functions are compiled inline, meaning the conversion code is part of the code that evaluates the expression. Execution is faster because there is no call to a procedure to accomplish the conversion. Each function coerces an expression to a specific data type.

```
CBool(expression)
CByte(expression)
CChar(expression)
CDate(expression)
CDbl(expression)
CDec(expression)
CInt(expression)
CLng(expression)
CObj(expression)
CShort(expression)
CSng(expression)
CStr(expression)
```

Part

expression
Required. Any **String** expression or numeric expression.

Return Types

The function name determines the return type, as shown in the following table:

Function name	Return type	Range for *expression* argument
CBool	**Boolean**	Any valid **String** or numeric expression.
CByte	**Byte**	0 through 255; fractions are rounded.
CChar	**Char**	Any valid **String** expression; value can be 0 through 65535.
CDate	**Date**	Any valid representation of a date and time.
CDbl	**Double**	-1.79769313486231E+308 through -4.94065645841247E-324 for negative values; 4.94065645841247E-324 through 1.79769313486231E+308 for positive values.
CDec	**Decimal**	+/-79,228,162,514,264,337,593,543,950,335 for zero-scaled numbers, that is, numbers with no decimal places. For numbers with 28 decimal places, the range is +/-7.9228162514264337593543950335. The smallest possible non-zero number is 0.0000000000000000000000000001.
CInt	**Integer**	-2,147,483,648 through 2,147,483,647; fractions are rounded.
CLng	**Long**	-9,223,372,036,854,775,808 through 9,223,372,036,854,775,807; fractions are rounded.
CObj	**Object**	Any valid expression.
CShort	**Short**	-32,768 through 32,767; fractions are rounded.
CSng	**Single**	-3.402823E+38 through -1.401298E-45 for negative values; 1.401298E-45 through 3.402823E+38 for positive values.
CStr	**String**	Returns for **CStr** depend on the *expression* argument. See Returns for CStr.

Remarks

If the *expression* passed to the function is outside the range of the data type to which it is being converted, an error occurs.

In general, you can use the data type conversion functions to force the result of some operation to a particular data type rather than the default data type. For example, use **CDec** to force decimal arithmetic in cases where single-precision, double-precision, or integer arithmetic normally would occur.

When the fractional part is exactly 0.5, **CInt** and **CLng** always round it to the nearest even number. For example, 0.5 rounds to 0

and 1.5 rounds to 2. **CInt** and **CLng** differ from the **Fix** and **Int** functions, which truncate, rather than round, the fractional part of a number. Also, **Fix** and **Int** always return a value of the same type as is passed in.

Use the **IsDate** function to determine if a value can be converted to a date and time. **CDate** recognizes date literals and time literals as well as numbers that fall within the range of acceptable dates.

CDate recognizes date formats according to the locale setting of your system. You must provide the day, month, and year in the correct order for your locale, or the date may not be interpreted correctly. A long date format is not recognized if it contains a day-of-the-week string, such as "Wednesday".

The **Date** data type always contains both date and time information. For purposes of type conversion, Visual Basic .NET considers 1/1/1 (January 1 of the year 1) to be a neutral value for the date, and 00:00:00 (midnight) to be a neutral value for the time. If you convert a **Date** value to a string, **CStr** does not include neutral values in the resulting string. For example, if you convert `#January 1, 0001 9:30:00#` to a string, the result is "9:30:00 AM"; the date information is suppressed. However, the date information is still present in the original **Date** value and can be recovered with functions such as **DatePart**.

The **CType** function takes a second argument, *typename*, and coerces *expression* to *typename*, where *typename* can be any data type, structure, class, or interface. For more information, see CType Function.

CBool Example

This example uses the **CBool** function to convert expressions to **Boolean** values. If an expression evaluates to a nonzero value, **CBool** returns **True**; otherwise, it returns **False**.

```
Dim A, B, C As Integer
Dim Check As Boolean
A = 5
B = 5
Check = CBool(A = B)   ' Check is set to True.
' ...
C = 0
Check = CBool(C)   ' Check is set to False.
```

CByte Example

This example uses the **CByte** function to convert an expression to a **Byte**.

```
Dim MyDouble As Double
Dim MyByte As Byte
MyDouble = 125.5678
MyByte = CByte(MyDouble)   ' MyByte is set to 126.
```

CChar Example

This example uses the **CChar** function to convert a the first character of a **String** expression to a **Char** type.

```
Dim MyString As String
Dim MyChar As Char
MyString = "BCD"   ' CChar converts only first character of string.
MyChar = CChar(MyString)   ' MyChar is set to "B".
```

The input argument to **CChar** must be of data type **String**. You cannot use **CChar** to convert a number to a character, because **CChar** cannot accept a numeric data type. This example obtains a number representing a code point (character code) and converts it to the corresponding character. It uses **InputBox** to obtain the string of digits, **CInt** to convert the string to type **Integer**, and **ChrW** to convert the number to type **Char**.

```
Dim MyDigits As String   ' Input string of digits to be converted.
Dim CodePoint As Integer   ' Number to be represented as a character.
Dim MyChar As Char
MyDigits = InputBox("Enter code point of character:")
CodePoint = CInt(MyDigits)   ' Convert entire string to Integer.
MyChar = ChrW(CodePoint)   ' MyChar is set to Char value of code point.
```

CDate Example

This example uses the **CDate** function to convert strings to **Date** values. In general, hard-coding dates and times as strings (as shown in this example) is not recommended. Use date literals and time literals, such as #Feb 12, 1969# and #4:45:23 PM#, instead.

```
Dim MyDateString, MyTimeString As String
Dim MyDate, MyTime As Date
MyDateString = "February 12, 1969"
MyTimeString = "4:35:47 PM"
' ...
MyDate = CDate(MyDateString)   ' Convert to Date data type.
MyTime = CDate(MyTimeString)   ' Convert to Date data type.
```

CDbl Example

This example uses the **CDbl** function to convert an expression to **Double**.

```
Dim MyDec As Decimal
Dim MyDouble As Double
MyDec = 234.456784D    ' Literal type character D makes MyDec a Decimal.
MyDouble = CDbl(MyDec * 8.2D * 0.01D)   ' Convert result to a Double.
```

CDec Example

This example uses the **CDec** function to convert a numeric value to **Decimal**.

```
Dim MyDouble As Double
Dim MyDecimal As Decimal
MyDouble = 10000000.0587
MyDecimal = CDec(MyDouble)   ' Convert to Decimal.
```

CInt Example

This example uses the **CInt** function to convert a value to **Integer**.

```
Dim MyDouble As Double
Dim MyInt As Integer
MyDouble = 2345.5678
MyInt = CInt(MyDouble)   ' MyInt is set to 2346.
```

CLng Example

This example uses the **CLng** function to convert values to **Long**.

```
Dim MyDbl1, MyDbl2 As Double
Dim MyLong1, MyLong2 As Long
MyDbl1 = 25427.45
MyDbl2 = 25427.55
MyLong1 = CLng(MyDbl1)   ' MyLong1 contains 25427.
MyLong2 = CLng(MyDbl2)   ' MyLong2 contains 25428.
```

CObj Example

This example uses the **CObj** function to convert a numeric value to **Object**. The **Object** variable itself contains only a four-byte pointer, which points to the **Double** value assigned to it.

```
Dim MyDouble As Double
Dim MyObject As Object
```

```
MyDouble = 2.7182818284
MyObject = CObj(MyDouble)   ' Double value is pointed to by MyObject.
```

CShort Example

This example uses the **CShort** function to convert a numeric value to **Short**.

```
Dim MyByte as Byte
Dim MyShort as Short
MyByte = 100
MyShort = CShort(MyByte)   ' Convert to Short.
```

CSng Example

This example uses the **CSng** function to convert values to **Single**.

```
Dim MyDouble1, MyDouble2 As Double
Dim MySingle1, MySingle2 As Single
MyDouble1 = 75.3421105
MyDouble2 = 75.3421567
MySingle1 = CSng(MyDouble1)   ' MySingle1 is set to 75.34211.
MySingle2 = CSng(MyDouble2)   ' MySingle2 is set to 75.34216.
```

CStr Example

This example uses the **CStr** function to convert a numeric value to **String**.

```
Dim MyDouble As Double
Dim MyString As String
MyDouble = 437.324
MyString = CStr(MyDouble)   ' MyString is set to "437.324".
```

This example uses the **CStr** function to convert **Date** values to **String** values.

```
Dim MyDate As Date
Dim MyString As String
' ...
MyDate = #February 12, 1969 00:00:00#   ' INVALID format.
' Date literals must be in the format #m/d/yyyy# or they are invalid.
' ...
MyDate = #2/12/69 00:00:00#   ' Time is midnight.
' The neutral time value of 00:00:00 is suppressed in the conversion.
MyString = CStr(MyDate)   ' MyString is set to "2/12/1969".
' ...
MyDate = #2/12/69 00:00:01#   ' Time is one second past midnight.
' The time component becomes part of the converted value.
MyString = CStr(MyDate)   ' MyString is set to "2/12/1969 12:00:01 AM".
```

CStr always renders a **Date** value in the standard short format for the current locale, for example, "6/15/2003 4:35:47 PM".

See Also

Conversion Functions | CType Function | Asc, AscW Functions | Chr, ChrW Functions | Format Function | Hex Function | Oct Function | Str Function | Val Function | DatePart Function | Returns for CStr | Type Conversions

Type Conversion Functions

These functions are compiled inline, meaning the conversion code is part of the code that evaluates the expression. Execution is faster because there is no call to a procedure to accomplish the conversion. Each function coerces an expression to a specific data type.

```
CBool(expression)
CByte(expression)
CChar(expression)
CDate(expression)
CDbl(expression)
CDec(expression)
CInt(expression)
CLng(expression)
CObj(expression)
CShort(expression)
CSng(expression)
CStr(expression)
```

Part

expression
Required. Any **String** expression or numeric expression.

Return Types

The function name determines the return type, as shown in the following table:

Function name	Return type	Range for *expression* argument
CBool	**Boolean**	Any valid **String** or numeric expression.
CByte	**Byte**	0 through 255; fractions are rounded.
CChar	**Char**	Any valid **String** expression; value can be 0 through 65535.
CDate	**Date**	Any valid representation of a date and time.
CDbl	**Double**	-1.79769313486231E+308 through -4.94065645841247E-324 for negative values; 4.94065645841247E-324 through 1.79769313486231E+308 for positive values.
CDec	**Decimal**	+/-79,228,162,514,264,337,593,543,950,335 for zero-scaled numbers, that is, numbers with no decimal places. For numbers with 28 decimal places, the range is +/-7.9228162514264337593543950335. The smallest possible non-zero number is 0.0000000000000000000000000001.
CInt	**Integer**	-2,147,483,648 through 2,147,483,647; fractions are rounded.
CLng	**Long**	-9,223,372,036,854,775,808 through 9,223,372,036,854,775,807; fractions are rounded.
CObj	**Object**	Any valid expression.
CShort	**Short**	-32,768 through 32,767; fractions are rounded.
CSng	**Single**	-3.402823E+38 through -1.401298E-45 for negative values; 1.401298E-45 through 3.402823E+38 for positive values.
CStr	**String**	Returns for **CStr** depend on the *expression* argument. See Returns for CStr.

Remarks

If the *expression* passed to the function is outside the range of the data type to which it is being converted, an error occurs.

In general, you can use the data type conversion functions to force the result of some operation to a particular data type rather than the default data type. For example, use **CDec** to force decimal arithmetic in cases where single-precision, double-precision, or integer arithmetic normally would occur.

When the fractional part is exactly 0.5, **CInt** and **CLng** always round it to the nearest even number. For example, 0.5 rounds to 0

and 1.5 rounds to 2. **CInt** and **CLng** differ from the **Fix** and **Int** functions, which truncate, rather than round, the fractional part of a number. Also, **Fix** and **Int** always return a value of the same type as is passed in.

Use the **IsDate** function to determine if a value can be converted to a date and time. **CDate** recognizes date literals and time literals as well as numbers that fall within the range of acceptable dates.

CDate recognizes date formats according to the locale setting of your system. You must provide the day, month, and year in the correct order for your locale, or the date may not be interpreted correctly. A long date format is not recognized if it contains a day-of-the-week string, such as "Wednesday".

The **Date** data type always contains both date and time information. For purposes of type conversion, Visual Basic .NET considers 1/1/1 (January 1 of the year 1) to be a neutral value for the date, and 00:00:00 (midnight) to be a neutral value for the time. If you convert a **Date** value to a string, **CStr** does not include neutral values in the resulting string. For example, if you convert `#January 1, 0001 9:30:00#` to a string, the result is "9:30:00 AM"; the date information is suppressed. However, the date information is still present in the original **Date** value and can be recovered with functions such as **DatePart**.

The **CType** function takes a second argument, *typename*, and coerces *expression* to *typename*, where *typename* can be any data type, structure, class, or interface. For more information, see CType Function.

CBool Example

This example uses the **CBool** function to convert expressions to **Boolean** values. If an expression evaluates to a nonzero value, **CBool** returns **True**; otherwise, it returns **False**.

```
Dim A, B, C As Integer
Dim Check As Boolean
A = 5
B = 5
Check = CBool(A = B)   ' Check is set to True.
' ...
C = 0
Check = CBool(C)   ' Check is set to False.
```

CByte Example

This example uses the **CByte** function to convert an expression to a **Byte**.

```
Dim MyDouble As Double
Dim MyByte As Byte
MyDouble = 125.5678
MyByte = CByte(MyDouble)   ' MyByte is set to 126.
```

CChar Example

This example uses the **CChar** function to convert a the first character of a **String** expression to a **Char** type.

```
Dim MyString As String
Dim MyChar As Char
MyString = "BCD"   ' CChar converts only first character of string.
MyChar = CChar(MyString)   ' MyChar is set to "B".
```

The input argument to **CChar** must be of data type **String**. You cannot use **CChar** to convert a number to a character, because **CChar** cannot accept a numeric data type. This example obtains a number representing a code point (character code) and converts it to the corresponding character. It uses **InputBox** to obtain the string of digits, **CInt** to convert the string to type **Integer**, and **ChrW** to convert the number to type **Char**.

```
Dim MyDigits As String   ' Input string of digits to be converted.
Dim CodePoint As Integer   ' Number to be represented as a character.
Dim MyChar As Char
MyDigits = InputBox("Enter code point of character:")
CodePoint = CInt(MyDigits)   ' Convert entire string to Integer.
MyChar = ChrW(CodePoint)   ' MyChar is set to Char value of code point.
```

CDate Example

This example uses the **CDate** function to convert strings to **Date** values. In general, hard-coding dates and times as strings (as shown in this example) is not recommended. Use date literals and time literals, such as #Feb 12, 1969# and #4:45:23 PM#, instead.

```
Dim MyDateString, MyTimeString As String
Dim MyDate, MyTime As Date
MyDateString = "February 12, 1969"
MyTimeString = "4:35:47 PM"
' ...
MyDate = CDate(MyDateString)   ' Convert to Date data type.
MyTime = CDate(MyTimeString)   ' Convert to Date data type.
```

CDbl Example

This example uses the **CDbl** function to convert an expression to **Double**.

```
Dim MyDec As Decimal
Dim MyDouble As Double
MyDec = 234.456784D    ' Literal type character D makes MyDec a Decimal.
MyDouble = CDbl(MyDec * 8.2D * 0.01D)   ' Convert result to a Double.
```

CDec Example

This example uses the **CDec** function to convert a numeric value to **Decimal**.

```
Dim MyDouble As Double
Dim MyDecimal As Decimal
MyDouble = 10000000.0587
MyDecimal = CDec(MyDouble)   ' Convert to Decimal.
```

CInt Example

This example uses the **CInt** function to convert a value to **Integer**.

```
Dim MyDouble As Double
Dim MyInt As Integer
MyDouble = 2345.5678
MyInt = CInt(MyDouble)   ' MyInt is set to 2346.
```

CLng Example

This example uses the **CLng** function to convert values to **Long**.

```
Dim MyDbl1, MyDbl2 As Double
Dim MyLong1, MyLong2 As Long
MyDbl1 = 25427.45
MyDbl2 = 25427.55
MyLong1 = CLng(MyDbl1)   ' MyLong1 contains 25427.
MyLong2 = CLng(MyDbl2)   ' MyLong2 contains 25428.
```

CObj Example

This example uses the **CObj** function to convert a numeric value to **Object**. The **Object** variable itself contains only a four-byte pointer, which points to the **Double** value assigned to it.

```
Dim MyDouble As Double
Dim MyObject As Object
```

```
MyDouble = 2.7182818284
MyObject = CObj(MyDouble)   ' Double value is pointed to by MyObject.
```

CShort Example

This example uses the **CShort** function to convert a numeric value to **Short**.

```
Dim MyByte as Byte
Dim MyShort as Short
MyByte = 100
MyShort = CShort(MyByte)   ' Convert to Short.
```

CSng Example

This example uses the **CSng** function to convert values to **Single**.

```
Dim MyDouble1, MyDouble2 As Double
Dim MySingle1, MySingle2 As Single
MyDouble1 = 75.3421105
MyDouble2 = 75.3421567
MySingle1 = CSng(MyDouble1)   ' MySingle1 is set to 75.34211.
MySingle2 = CSng(MyDouble2)   ' MySingle2 is set to 75.34216.
```

CStr Example

This example uses the **CStr** function to convert a numeric value to **String**.

```
Dim MyDouble As Double
Dim MyString As String
MyDouble = 437.324
MyString = CStr(MyDouble)   ' MyString is set to "437.324".
```

This example uses the **CStr** function to convert **Date** values to **String** values.

```
Dim MyDate As Date
Dim MyString As String
' ...
MyDate = #February 12, 1969 00:00:00#   ' INVALID format.
' Date literals must be in the format #m/d/yyyy# or they are invalid.
' ...
MyDate = #2/12/69 00:00:00#   ' Time is midnight.
' The neutral time value of 00:00:00 is suppressed in the conversion.
MyString = CStr(MyDate)   ' MyString is set to "2/12/1969".
' ...
MyDate = #2/12/69 00:00:01#   ' Time is one second past midnight.
' The time component becomes part of the converted value.
MyString = CStr(MyDate)   ' MyString is set to "2/12/1969 12:00:01 AM".
```

CStr always renders a **Date** value in the standard short format for the current locale, for example, "6/15/2003 4:35:47 PM".

See Also

Conversion Functions | CType Function | Asc, AscW Functions | Chr, ChrW Functions | Format Function | Hex Function | Oct Function | Str Function | Val Function | DatePart Function | Returns for CStr | Type Conversions

ChDir Function

Changes the current directory or folder.

```
Public Sub ChDir(ByVal Path As String)
```

Parameter

Path
Required. A **String** expression that identifies which directory or folder becomes the new default directory or folder. *Path* may include the drive. If no drive is specified, **ChDir** changes the default directory or folder on the current drive.

Exceptions/Errors

Exception type	Error number	Condition
ArgumentException	52	*Path* is empty.
FileNotFoundException	76	Invalid drive is specified, or drive is unavail able.

Remarks

The **ChDir** function changes the default directory, but not the default drive. For example, if the default drive is C, the following statement changes the default directory on drive D, but C remains the default drive:

```
ChDir("D:\TMP")
```

Making relative directory changes is accomplished using "..", as follows:

```
ChDir("..") ' Moves up one directory.
```

Security Note The **ChDir** function requires unmanaged code permission, which may affect its execution in partial trust situations. For more information, see SecurityPermission Class and Code Access Permissions.

Example

This example uses the **ChDir** function to change the current directory or folder.

```
' Change current directory or folder to "MYDIR".
ChDir("MYDIR")

' Assume "C:" is the current drive. The following statement changes
' the default directory on drive "D:". "C:" remains the current drive.
ChDir("D:\WINDOWS\SYSTEM")
```

Smart Device Developer Notes

This function is not supported.

Requirements

Namespace: Microsoft.VisualBasic

Module: FileSystem

Assembly: Microsoft Visual Basic .NET Runtime (in MicrosoftVisualBasic.dll)

See Also

ChDrive Function | CurDir Function | Dir Function | MkDir Function | RmDir Function | ArgumentException Class | FileNotFoundException Class

ChDrive Function

Changes the current drive.

```
Public Overloads Sub ChDrive(ByVal Drive As { Char | String })
```

Parameter

Drive
Required. String expression that specifies an existing drive. If you supply a zero-length string (""), the current drive doesn't change. If the *Drive* argument is a multiple-character string, **ChDrive** uses only the first letter.

Exceptions/Errors

Exception type	Error number	Condition
IOException	68	Invalid drive is specified, or drive is unavailable.

Remarks

The **ChDrive** function requires unmanaged code permission, which may affect its execution in partial trust situations. For more information, see SecurityPermission Class and Code Access Permissions.

Example

This example uses the **ChDrive** function to change the current drive. The function throws an exception if the drive does not exist.

```
ChDrive("D")   ' Make "D" the current drive.
```

Smart Device Developer Notes

This function is not supported.

Requirements

Namespace: Microsoft.VisualBasic

Module: FileSystem

Assembly: Microsoft Visual Basic .NET Runtime (in MicrosoftVisualBasic.dll)

See Also

ChDir Function | CurDir Function | MkDir Function | RmDir Function | IOException Class

Choose Function

Selects and returns a value from a list of arguments.

```
Public Function Choose( _
   ByVal Index As Double, _
   ByVal ParamArray Choice() As Object _
) As Object
```

Parameters

Index
Required. **Double**. Numeric expression that results in a value between 1 and the number of elements passed in the *Choice* argument.

Choice()
Required. **Object** parameter array. You can supply either a single variable or an expression that evaluates to the **Object** data type, to a list of **Object** variables or expressions separated by commas, or to a single-dimensional array of **Object** elements.

Exceptions/Errors

Exception type	Error number	Condition
ArgumentException	5	Rank of *Choice()* <> 1 .

Remarks

The **Choose** function returns a member of the list passed in *Choice()*, based on the value of *Index*. The first member of the list is selected when *Index* is 1. The last member of the list is selected when *Index* is **UBound**(*Choice()*). If *Index* is outside these limits, **Choose** returns **Nothing**.

If *Index* is not a whole number, it is rounded to the nearest whole number before being evaluated.

You can use **Choose** to look up a value in a list of possibilities.

> **Note** The expressions in the argument list can include function calls. As part of preparing the argument list for the call to **Choose**, the Visual Basic compiler calls every function in every expression. This means that you cannot rely on a particular function not being called if a different expression is selected by *Index*.

Example

This example uses the **Choose** function to display a name in response to an index passed into the procedure in the `Ind` parameter.

```
Function GetChoice(Ind As Integer) As String
   GetChoice = CStr(Choose(Ind, "Speedy", "United", "Federal"))
End Function
```

Requirements

Namespace: Microsoft.VisualBasic

Module: Interaction

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

IIf Function | Select...Case Statements | Switch Function

Chr, ChrW Functions

Returns the character associated with the specified character code.

```
Public Function Chr(ByVal CharCode As Integer) As Char
Public Function ChrW(ByVal CharCode As Integer) As Char
```

Parameter

CharCode
Required. An **Integer** expression representing the *code point*, or character code, for the character. If *CharCode* is outside the range -32768 through 65535, an **ArgumentException** error occurs.

Exceptions/Errors

Exception type	Error number	Condition
ArgumentException	5	*CharCode* < -32768 or > 65535 .

Remarks

The asymmetric range accepted for *CharCode* compensates for the storage differences between the **Short** and **Integer** data types. For example, &H8E01 is -29183 as a **Short** but +36353 as an **Integer**. This also facilitates compatibility with Visual Basic 6.0.

Chr uses the **Encoding** class in the **System.Text** namespace to determine if the current thread is using a single-byte character set (SBCS) or a double-byte character set (DBCS). It then takes *CharCode* as a code point in the appropriate set. The range can be 0 through 255 for SBCS characters and -32768 through 65535 for DBCS characters. The returned character depends on the code page for the current thread, which is contained in the **ANSICodePage** property of the **TextInfo** class. **TextInfo.ANSICodePage** can be obtained by specifying **System.Globalization.CultureInfo.CurrentCulture.TextInfo.ANSICodePage**.

ChrW takes *CharCode* as a Unicode code point. The range is independent of the culture and code page settings for the current thread. Values from -32768 through -1 are treated the same as values in the range +32768 through +65535.

Numbers from 0 through 31 are the same as standard, nonprintable ASCII codes. For example, `Chr(10)` returns a linefeed character.

> **Note** **The ChrB function of earlier versions of Visual Basic returns a single byte. It is used primarily for converting strings in double-byte character set (DBCS) applications. All Visual Basic .NET strings are in Unicode, and ChrB is no longer supported**.

Example

This example uses the **Chr** function to return the character associated with the specified character code.

```
Dim MyChar As Char
MyChar = Chr(65)   ' Returns "A".
MyChar = Chr(97)   ' Returns "a".
MyChar = Chr(62)   ' Returns ">".
MyChar = Chr(37)   ' Returns "%".
```

Requirements

Namespace: Microsoft.VisualBasic

Module: Strings

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Asc, AscW Functions | Str Function | Conversion Functions | Type Conversion Functions | System.Text Namespace | Encoding Class | System.Globalization Namespace | CultureInfo Class | TextInfo.ANSICodePage Property | ArgumentException Class

Chr, ChrW Functions

Returns the character associated with the specified character code.

```
Public Function Chr(ByVal CharCode As Integer) As Char
Public Function ChrW(ByVal CharCode As Integer) As Char
```

Parameter

CharCode
Required. An **Integer** expression representing the *code point*, or character code, for the character. If *CharCode* is outside the range -32768 through 65535, an **ArgumentException** error occurs.

Exceptions/Errors

Exception type	Error number	Condition
ArgumentException	5	*CharCode* < -32768 or > 65535 .

Remarks

The asymmetric range accepted for *CharCode* compensates for the storage differences between the **Short** and **Integer** data types. For example, &H8E01 is -29183 as a **Short** but +36353 as an **Integer**. This also facilitates compatibility with Visual Basic 6.0.

Chr uses the **Encoding** class in the **System.Text** namespace to determine if the current thread is using a single-byte character set (SBCS) or a double-byte character set (DBCS). It then takes *CharCode* as a code point in the appropriate set. The range can be 0 through 255 for SBCS characters and -32768 through 65535 for DBCS characters. The returned character depends on the code page for the current thread, which is contained in the **ANSICodePage** property of the **TextInfo** class. **TextInfo.ANSICodePage** can be obtained by specifying **System.Globalization.CultureInfo.CurrentCulture.TextInfo.ANSICodePage**.

ChrW takes *CharCode* as a Unicode code point. The range is independent of the culture and code page settings for the current thread. Values from -32768 through -1 are treated the same as values in the range +32768 through +65535.

Numbers from 0 through 31 are the same as standard, nonprintable ASCII codes. For example, `Chr(10)` returns a linefeed character.

> **Note** **The ChrB function of earlier versions of Visual Basic returns a single byte. It is used primarily for converting strings in double-byte character set (DBCS) applications. All Visual Basic .NET strings are in Unicode, and ChrB is no longer supported**.

Example

This example uses the **Chr** function to return the character associated with the specified character code.

```
Dim MyChar As Char
MyChar = Chr(65)   ' Returns "A".
MyChar = Chr(97)   ' Returns "a".
MyChar = Chr(62)   ' Returns ">".
MyChar = Chr(37)   ' Returns "%".
```

Requirements

Namespace: Microsoft.VisualBasic

Module: Strings

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Asc, AscW Functions | Str Function | Conversion Functions | Type Conversion Functions | System.Text Namespace | Encoding Class | System.Globalization Namespace | CultureInfo Class | TextInfo.ANSICodePage Property | ArgumentException Class

Type Conversion Functions

These functions are compiled inline, meaning the conversion code is part of the code that evaluates the expression. Execution is faster because there is no call to a procedure to accomplish the conversion. Each function coerces an expression to a specific data type.

```
CBool(expression)
CByte(expression)
CChar(expression)
CDate(expression)
CDbl(expression)
CDec(expression)
CInt(expression)
CLng(expression)
CObj(expression)
CShort(expression)
CSng(expression)
CStr(expression)
```

Part

expression
Required. Any **String** expression or numeric expression.

Return Types

The function name determines the return type, as shown in the following table:

Function name	Return type	Range for *expression* argument
CBool	**Boolean**	Any valid **String** or numeric expression.
CByte	**Byte**	0 through 255; fractions are rounded.
CChar	**Char**	Any valid **String** expression; value can be 0 through 65535.
CDate	**Date**	Any valid representation of a date and time.
CDbl	**Double**	-1.79769313486231E+308 through -4.94065645841247E-324 for negative values; 4.94065645841247E-324 through 1.79769313486231E+308 for positive values.
CDec	**Decimal**	+/-79,228,162,514,264,337,593,543,950,335 for zero-scaled numbers, that is, numbers with no decimal places. For numbers with 28 decimal places, the range is +/-7.9228162514264337593543950335. The smallest possible non-zero number is 0.0000000000000000000000000001.
CInt	**Integer**	-2,147,483,648 through 2,147,483,647; fractions are rounded.
CLng	**Long**	-9,223,372,036,854,775,808 through 9,223,372,036,854,775,807; fractions are rounded.
CObj	**Object**	Any valid expression.
CShort	**Short**	-32,768 through 32,767; fractions are rounded.
CSng	**Single**	-3.402823E+38 through -1.401298E-45 for negative values; 1.401298E-45 through 3.402823E+38 for positive values.
CStr	**String**	Returns for **CStr** depend on the *expression* argument. See Returns for CStr.

Remarks

If the *expression* passed to the function is outside the range of the data type to which it is being converted, an error occurs.

In general, you can use the data type conversion functions to force the result of some operation to a particular data type rather than the default data type. For example, use **CDec** to force decimal arithmetic in cases where single-precision, double-precision, or integer arithmetic normally would occur.

When the fractional part is exactly 0.5, **CInt** and **CLng** always round it to the nearest even number. For example, 0.5 rounds to 0

and 1.5 rounds to 2. **CInt** and **CLng** differ from the **Fix** and **Int** functions, which truncate, rather than round, the fractional part of a number. Also, **Fix** and **Int** always return a value of the same type as is passed in.

Use the **IsDate** function to determine if a value can be converted to a date and time. **CDate** recognizes date literals and time literals as well as numbers that fall within the range of acceptable dates.

CDate recognizes date formats according to the locale setting of your system. You must provide the day, month, and year in the correct order for your locale, or the date may not be interpreted correctly. A long date format is not recognized if it contains a day-of-the-week string, such as "Wednesday".

The **Date** data type always contains both date and time information. For purposes of type conversion, Visual Basic .NET considers 1/1/1 (January 1 of the year 1) to be a neutral value for the date, and 00:00:00 (midnight) to be a neutral value for the time. If you convert a **Date** value to a string, **CStr** does not include neutral values in the resulting string. For example, if you convert `#January 1, 0001 9:30:00#` to a string, the result is "9:30:00 AM"; the date information is suppressed. However, the date information is still present in the original **Date** value and can be recovered with functions such as **DatePart**.

The **CType** function takes a second argument, *typename*, and coerces *expression* to *typename*, where *typename* can be any data type, structure, class, or interface. For more information, see CType Function.

CBool Example

This example uses the **CBool** function to convert expressions to **Boolean** values. If an expression evaluates to a nonzero value, **CBool** returns **True**; otherwise, it returns **False**.

```
Dim A, B, C As Integer
Dim Check As Boolean
A = 5
B = 5
Check = CBool(A = B)   ' Check is set to True.
' ...
C = 0
Check = CBool(C)   ' Check is set to False.
```

CByte Example

This example uses the **CByte** function to convert an expression to a **Byte**.

```
Dim MyDouble As Double
Dim MyByte As Byte
MyDouble = 125.5678
MyByte = CByte(MyDouble)   ' MyByte is set to 126.
```

CChar Example

This example uses the **CChar** function to convert a the first character of a **String** expression to a **Char** type.

```
Dim MyString As String
Dim MyChar As Char
MyString = "BCD"   ' CChar converts only first character of string.
MyChar = CChar(MyString)   ' MyChar is set to "B".
```

The input argument to **CChar** must be of data type **String**. You cannot use **CChar** to convert a number to a character, because **CChar** cannot accept a numeric data type. This example obtains a number representing a code point (character code) and converts it to the corresponding character. It uses **InputBox** to obtain the string of digits, **CInt** to convert the string to type **Integer**, and **ChrW** to convert the number to type **Char**.

```
Dim MyDigits As String   ' Input string of digits to be converted.
Dim CodePoint As Integer   ' Number to be represented as a character.
Dim MyChar As Char
MyDigits = InputBox("Enter code point of character:")
CodePoint = CInt(MyDigits)   ' Convert entire string to Integer.
MyChar = ChrW(CodePoint)   ' MyChar is set to Char value of code point.
```

CDate Example

This example uses the **CDate** function to convert strings to **Date** values. In general, hard-coding dates and times as strings (as shown in this example) is not recommended. Use date literals and time literals, such as #Feb 12, 1969# and #4:45:23 PM#, instead.

```
Dim MyDateString, MyTimeString As String
Dim MyDate, MyTime As Date
MyDateString = "February 12, 1969"
MyTimeString = "4:35:47 PM"
' ...
MyDate = CDate(MyDateString)   ' Convert to Date data type.
MyTime = CDate(MyTimeString)   ' Convert to Date data type.
```

CDbl Example

This example uses the **CDbl** function to convert an expression to **Double**.

```
Dim MyDec As Decimal
Dim MyDouble As Double
MyDec = 234.456784D    ' Literal type character D makes MyDec a Decimal.
MyDouble = CDbl(MyDec * 8.2D * 0.01D)   ' Convert result to a Double.
```

CDec Example

This example uses the **CDec** function to convert a numeric value to **Decimal**.

```
Dim MyDouble As Double
Dim MyDecimal As Decimal
MyDouble = 10000000.0587
MyDecimal = CDec(MyDouble)   ' Convert to Decimal.
```

CInt Example

This example uses the **CInt** function to convert a value to **Integer**.

```
Dim MyDouble As Double
Dim MyInt As Integer
MyDouble = 2345.5678
MyInt = CInt(MyDouble)   ' MyInt is set to 2346.
```

CLng Example

This example uses the **CLng** function to convert values to **Long**.

```
Dim MyDbl1, MyDbl2 As Double
Dim MyLong1, MyLong2 As Long
MyDbl1 = 25427.45
MyDbl2 = 25427.55
MyLong1 = CLng(MyDbl1)   ' MyLong1 contains 25427.
MyLong2 = CLng(MyDbl2)   ' MyLong2 contains 25428.
```

CObj Example

This example uses the **CObj** function to convert a numeric value to **Object**. The **Object** variable itself contains only a four-byte pointer, which points to the **Double** value assigned to it.

```
Dim MyDouble As Double
Dim MyObject As Object
```

```
MyDouble = 2.7182818284
MyObject = CObj(MyDouble)   ' Double value is pointed to by MyObject.
```

CShort Example

This example uses the **CShort** function to convert a numeric value to **Short**.

```
Dim MyByte as Byte
Dim MyShort as Short
MyByte = 100
MyShort = CShort(MyByte)   ' Convert to Short.
```

CSng Example

This example uses the **CSng** function to convert values to **Single**.

```
Dim MyDouble1, MyDouble2 As Double
Dim MySingle1, MySingle2 As Single
MyDouble1 = 75.3421105
MyDouble2 = 75.3421567
MySingle1 = CSng(MyDouble1)   ' MySingle1 is set to 75.34211.
MySingle2 = CSng(MyDouble2)   ' MySingle2 is set to 75.34216.
```

CStr Example

This example uses the **CStr** function to convert a numeric value to **String**.

```
Dim MyDouble As Double
Dim MyString As String
MyDouble = 437.324
MyString = CStr(MyDouble)   ' MyString is set to "437.324".
```

This example uses the **CStr** function to convert **Date** values to **String** values.

```
Dim MyDate As Date
Dim MyString As String
' ...
MyDate = #February 12, 1969 00:00:00#   ' INVALID format.
' Date literals must be in the format #m/d/yyyy# or they are invalid.
' ...
MyDate = #2/12/69 00:00:00#   ' Time is midnight.
' The neutral time value of 00:00:00 is suppressed in the conversion.
MyString = CStr(MyDate)   ' MyString is set to "2/12/1969".
' ...
MyDate = #2/12/69 00:00:01#   ' Time is one second past midnight.
' The time component becomes part of the converted value.
MyString = CStr(MyDate)   ' MyString is set to "2/12/1969 12:00:01 AM".
```

CStr always renders a **Date** value in the standard short format for the current locale, for example, "6/15/2003 4:35:47 PM".

See Also

Conversion Functions | CType Function | Asc, AscW Functions | Chr, ChrW Functions | Format Function | Hex Function | Oct Function | Str Function | Val Function | DatePart Function | Returns for CStr | Type Conversions

Type Conversion Functions

These functions are compiled inline, meaning the conversion code is part of the code that evaluates the expression. Execution is faster because there is no call to a procedure to accomplish the conversion. Each function coerces an expression to a specific data type.

```
CBool(expression)
CByte(expression)
CChar(expression)
CDate(expression)
CDbl(expression)
CDec(expression)
CInt(expression)
CLng(expression)
CObj(expression)
CShort(expression)
CSng(expression)
CStr(expression)
```

Part

expression
Required. Any **String** expression or numeric expression.

Return Types

The function name determines the return type, as shown in the following table:

Function name	Return type	Range for *expression* argument
CBool	**Boolean**	Any valid **String** or numeric expression.
CByte	**Byte**	0 through 255; fractions are rounded.
CChar	**Char**	Any valid **String** expression; value can be 0 through 65535.
CDate	**Date**	Any valid representation of a date and time.
CDbl	**Double**	-1.79769313486231E+308 through -4.94065645841247E-324 for negative values; 4.94065645841247E-324 through 1.79769313486231E+308 for positive values.
CDec	**Decimal**	+/-79,228,162,514,264,337,593,543,950,335 for zero-scaled numbers, that is, numbers with no decimal places. For numbers with 28 decimal places, the range is +/-7.9228162514264337593543950335. The smallest possible non-zero number is 0.0000000000000000000000000001.
CInt	**Integer**	-2,147,483,648 through 2,147,483,647; fractions are rounded.
CLng	**Long**	-9,223,372,036,854,775,808 through 9,223,372,036,854,775,807; fractions are rounded.
CObj	**Object**	Any valid expression.
CShort	**Short**	-32,768 through 32,767; fractions are rounded.
CSng	**Single**	-3.402823E+38 through -1.401298E-45 for negative values; 1.401298E-45 through 3.402823E+38 for positive values.
CStr	**String**	Returns for **CStr** depend on the *expression* argument. See Returns for CStr.

Remarks

If the *expression* passed to the function is outside the range of the data type to which it is being converted, an error occurs.

In general, you can use the data type conversion functions to force the result of some operation to a particular data type rather than the default data type. For example, use **CDec** to force decimal arithmetic in cases where single-precision, double-precision, or integer arithmetic normally would occur.

When the fractional part is exactly 0.5, **CInt** and **CLng** always round it to the nearest even number. For example, 0.5 rounds to 0

and 1.5 rounds to 2. **CInt** and **CLng** differ from the **Fix** and **Int** functions, which truncate, rather than round, the fractional part of a number. Also, **Fix** and **Int** always return a value of the same type as is passed in.

Use the **IsDate** function to determine if a value can be converted to a date and time. **CDate** recognizes date literals and time literals as well as numbers that fall within the range of acceptable dates.

CDate recognizes date formats according to the locale setting of your system. You must provide the day, month, and year in the correct order for your locale, or the date may not be interpreted correctly. A long date format is not recognized if it contains a day-of-the-week string, such as "Wednesday".

The **Date** data type always contains both date and time information. For purposes of type conversion, Visual Basic .NET considers 1/1/1 (January 1 of the year 1) to be a neutral value for the date, and 00:00:00 (midnight) to be a neutral value for the time. If you convert a **Date** value to a string, **CStr** does not include neutral values in the resulting string. For example, if you convert `#January 1, 0001 9:30:00#` to a string, the result is "9:30:00 AM"; the date information is suppressed. However, the date information is still present in the original **Date** value and can be recovered with functions such as **DatePart**.

The **CType** function takes a second argument, *typename*, and coerces *expression* to *typename*, where *typename* can be any data type, structure, class, or interface. For more information, see CType Function.

CBool Example

This example uses the **CBool** function to convert expressions to **Boolean** values. If an expression evaluates to a nonzero value, **CBool** returns **True**; otherwise, it returns **False**.

```
Dim A, B, C As Integer
Dim Check As Boolean
A = 5
B = 5
Check = CBool(A = B)   ' Check is set to True.
' ...
C = 0
Check = CBool(C)   ' Check is set to False.
```

CByte Example

This example uses the **CByte** function to convert an expression to a **Byte**.

```
Dim MyDouble As Double
Dim MyByte As Byte
MyDouble = 125.5678
MyByte = CByte(MyDouble)   ' MyByte is set to 126.
```

CChar Example

This example uses the **CChar** function to convert a the first character of a **String** expression to a **Char** type.

```
Dim MyString As String
Dim MyChar As Char
MyString = "BCD"   ' CChar converts only first character of string.
MyChar = CChar(MyString)   ' MyChar is set to "B".
```

The input argument to **CChar** must be of data type **String**. You cannot use **CChar** to convert a number to a character, because **CChar** cannot accept a numeric data type. This example obtains a number representing a code point (character code) and converts it to the corresponding character. It uses **InputBox** to obtain the string of digits, **CInt** to convert the string to type **Integer**, and **ChrW** to convert the number to type **Char**.

```
Dim MyDigits As String   ' Input string of digits to be converted.
Dim CodePoint As Integer   ' Number to be represented as a character.
Dim MyChar As Char
MyDigits = InputBox("Enter code point of character:")
CodePoint = CInt(MyDigits)   ' Convert entire string to Integer.
MyChar = ChrW(CodePoint)   ' MyChar is set to Char value of code point.
```

CDate Example

This example uses the **CDate** function to convert strings to **Date** values. In general, hard-coding dates and times as strings (as shown in this example) is not recommended. Use date literals and time literals, such as #Feb 12, 1969# and #4:45:23 PM#, instead.

```
Dim MyDateString, MyTimeString As String
Dim MyDate, MyTime As Date
MyDateString = "February 12, 1969"
MyTimeString = "4:35:47 PM"
' ...
MyDate = CDate(MyDateString)   ' Convert to Date data type.
MyTime = CDate(MyTimeString)   ' Convert to Date data type.
```

CDbl Example

This example uses the **CDbl** function to convert an expression to **Double**.

```
Dim MyDec As Decimal
Dim MyDouble As Double
MyDec = 234.456784D    ' Literal type character D makes MyDec a Decimal.
MyDouble = CDbl(MyDec * 8.2D * 0.01D)   ' Convert result to a Double.
```

CDec Example

This example uses the **CDec** function to convert a numeric value to **Decimal**.

```
Dim MyDouble As Double
Dim MyDecimal As Decimal
MyDouble = 10000000.0587
MyDecimal = CDec(MyDouble)   ' Convert to Decimal.
```

CInt Example

This example uses the **CInt** function to convert a value to **Integer**.

```
Dim MyDouble As Double
Dim MyInt As Integer
MyDouble = 2345.5678
MyInt = CInt(MyDouble)   ' MyInt is set to 2346.
```

CLng Example

This example uses the **CLng** function to convert values to **Long**.

```
Dim MyDbl1, MyDbl2 As Double
Dim MyLong1, MyLong2 As Long
MyDbl1 = 25427.45
MyDbl2 = 25427.55
MyLong1 = CLng(MyDbl1)   ' MyLong1 contains 25427.
MyLong2 = CLng(MyDbl2)   ' MyLong2 contains 25428.
```

CObj Example

This example uses the **CObj** function to convert a numeric value to **Object**. The **Object** variable itself contains only a four-byte pointer, which points to the **Double** value assigned to it.

```
Dim MyDouble As Double
Dim MyObject As Object
```

```
MyDouble = 2.7182818284
MyObject = CObj(MyDouble)   ' Double value is pointed to by MyObject.
```

CShort Example

This example uses the **CShort** function to convert a numeric value to **Short**.

```
Dim MyByte as Byte
Dim MyShort as Short
MyByte = 100
MyShort = CShort(MyByte)   ' Convert to Short.
```

CSng Example

This example uses the **CSng** function to convert values to **Single**.

```
Dim MyDouble1, MyDouble2 As Double
Dim MySingle1, MySingle2 As Single
MyDouble1 = 75.3421105
MyDouble2 = 75.3421567
MySingle1 = CSng(MyDouble1)   ' MySingle1 is set to 75.34211.
MySingle2 = CSng(MyDouble2)   ' MySingle2 is set to 75.34216.
```

CStr Example

This example uses the **CStr** function to convert a numeric value to **String**.

```
Dim MyDouble As Double
Dim MyString As String
MyDouble = 437.324
MyString = CStr(MyDouble)   ' MyString is set to "437.324".
```

This example uses the **CStr** function to convert **Date** values to **String** values.

```
Dim MyDate As Date
Dim MyString As String
' ...
MyDate = #February 12, 1969 00:00:00#   ' INVALID format.
' Date literals must be in the format #m/d/yyyy# or they are invalid.
' ...
MyDate = #2/12/69 00:00:00#   ' Time is midnight.
' The neutral time value of 00:00:00 is suppressed in the conversion.
MyString = CStr(MyDate)   ' MyString is set to "2/12/1969".
' ...
MyDate = #2/12/69 00:00:01#   ' Time is one second past midnight.
' The time component becomes part of the converted value.
MyString = CStr(MyDate)   ' MyString is set to "2/12/1969 12:00:01 AM".
```

CStr always renders a **Date** value in the standard short format for the current locale, for example, "6/15/2003 4:35:47 PM".

See Also

Conversion Functions | CType Function | Asc, AscW Functions | Chr, ChrW Functions | Format Function | Hex Function | Oct Function | Str Function | Val Function | DatePart Function | Returns for CStr | Type Conversions

Type Conversion Functions

These functions are compiled inline, meaning the conversion code is part of the code that evaluates the expression. Execution is faster because there is no call to a procedure to accomplish the conversion. Each function coerces an expression to a specific data type.

```
CBool(expression)
CByte(expression)
CChar(expression)
CDate(expression)
CDbl(expression)
CDec(expression)
CInt(expression)
CLng(expression)
CObj(expression)
CShort(expression)
CSng(expression)
CStr(expression)
```

Part

expression
Required. Any **String** expression or numeric expression.

Return Types

The function name determines the return type, as shown in the following table:

Function name	Return type	Range for *expression* argument
CBool	**Boolean**	Any valid **String** or numeric expression.
CByte	**Byte**	0 through 255; fractions are rounded.
CChar	**Char**	Any valid **String** expression; value can be 0 through 65535.
CDate	**Date**	Any valid representation of a date and time.
CDbl	**Double**	-1.79769313486231E+308 through -4.94065645841247E-324 for negative values; 4.94065645841247E-324 through 1.79769313486231E+308 for positive values.
CDec	**Decimal**	+/-79,228,162,514,264,337,593,543,950,335 for zero-scaled numbers, that is, numbers with no decimal places. For numbers with 28 decimal places, the range is +/-7.9228162514264337593543950335. The smallest possible non-zero number is 0.0000000000000000000000000001.
CInt	**Integer**	-2,147,483,648 through 2,147,483,647; fractions are rounded.
CLng	**Long**	-9,223,372,036,854,775,808 through 9,223,372,036,854,775,807; fractions are rounded.
CObj	**Object**	Any valid expression.
CShort	**Short**	-32,768 through 32,767; fractions are rounded.
CSng	**Single**	-3.402823E+38 through -1.401298E-45 for negative values; 1.401298E-45 through 3.402823E+38 for positive values.
CStr	**String**	Returns for **CStr** depend on the *expression* argument. See Returns for CStr.

Remarks

If the *expression* passed to the function is outside the range of the data type to which it is being converted, an error occurs.

In general, you can use the data type conversion functions to force the result of some operation to a particular data type rather than the default data type. For example, use **CDec** to force decimal arithmetic in cases where single-precision, double-precision, or integer arithmetic normally would occur.

When the fractional part is exactly 0.5, **CInt** and **CLng** always round it to the nearest even number. For example, 0.5 rounds to 0

and 1.5 rounds to 2. **CInt** and **CLng** differ from the **Fix** and **Int** functions, which truncate, rather than round, the fractional part of a number. Also, **Fix** and **Int** always return a value of the same type as is passed in.

Use the **IsDate** function to determine if a value can be converted to a date and time. **CDate** recognizes date literals and time literals as well as numbers that fall within the range of acceptable dates.

CDate recognizes date formats according to the locale setting of your system. You must provide the day, month, and year in the correct order for your locale, or the date may not be interpreted correctly. A long date format is not recognized if it contains a day-of-the-week string, such as "Wednesday".

The **Date** data type always contains both date and time information. For purposes of type conversion, Visual Basic .NET considers 1/1/1 (January 1 of the year 1) to be a neutral value for the date, and 00:00:00 (midnight) to be a neutral value for the time. If you convert a **Date** value to a string, **CStr** does not include neutral values in the resulting string. For example, if you convert `#January 1, 0001 9:30:00#` to a string, the result is "9:30:00 AM"; the date information is suppressed. However, the date information is still present in the original **Date** value and can be recovered with functions such as **DatePart**.

The **CType** function takes a second argument, *typename*, and coerces *expression* to *typename*, where *typename* can be any data type, structure, class, or interface. For more information, see CType Function.

CBool Example

This example uses the **CBool** function to convert expressions to **Boolean** values. If an expression evaluates to a nonzero value, **CBool** returns **True**; otherwise, it returns **False**.

```
Dim A, B, C As Integer
Dim Check As Boolean
A = 5
B = 5
Check = CBool(A = B)   ' Check is set to True.
' ...
C = 0
Check = CBool(C)   ' Check is set to False.
```

CByte Example

This example uses the **CByte** function to convert an expression to a **Byte**.

```
Dim MyDouble As Double
Dim MyByte As Byte
MyDouble = 125.5678
MyByte = CByte(MyDouble)   ' MyByte is set to 126.
```

CChar Example

This example uses the **CChar** function to convert a the first character of a **String** expression to a **Char** type.

```
Dim MyString As String
Dim MyChar As Char
MyString = "BCD"   ' CChar converts only first character of string.
MyChar = CChar(MyString)   ' MyChar is set to "B".
```

The input argument to **CChar** must be of data type **String**. You cannot use **CChar** to convert a number to a character, because **CChar** cannot accept a numeric data type. This example obtains a number representing a code point (character code) and converts it to the corresponding character. It uses **InputBox** to obtain the string of digits, **CInt** to convert the string to type **Integer**, and **ChrW** to convert the number to type **Char**.

```
Dim MyDigits As String   ' Input string of digits to be converted.
Dim CodePoint As Integer   ' Number to be represented as a character.
Dim MyChar As Char
MyDigits = InputBox("Enter code point of character:")
CodePoint = CInt(MyDigits)   ' Convert entire string to Integer.
MyChar = ChrW(CodePoint)   ' MyChar is set to Char value of code point.
```

CDate Example

This example uses the **CDate** function to convert strings to **Date** values. In general, hard-coding dates and times as strings (as shown in this example) is not recommended. Use date literals and time literals, such as #Feb 12, 1969# and #4:45:23 PM#, instead.

```
Dim MyDateString, MyTimeString As String
Dim MyDate, MyTime As Date
MyDateString = "February 12, 1969"
MyTimeString = "4:35:47 PM"
' ...
MyDate = CDate(MyDateString)   ' Convert to Date data type.
MyTime = CDate(MyTimeString)   ' Convert to Date data type.
```

CDbl Example

This example uses the **CDbl** function to convert an expression to **Double**.

```
Dim MyDec As Decimal
Dim MyDouble As Double
MyDec = 234.456784D    ' Literal type character D makes MyDec a Decimal.
MyDouble = CDbl(MyDec * 8.2D * 0.01D)   ' Convert result to a Double.
```

CDec Example

This example uses the **CDec** function to convert a numeric value to **Decimal**.

```
Dim MyDouble As Double
Dim MyDecimal As Decimal
MyDouble = 10000000.0587
MyDecimal = CDec(MyDouble)   ' Convert to Decimal.
```

CInt Example

This example uses the **CInt** function to convert a value to **Integer**.

```
Dim MyDouble As Double
Dim MyInt As Integer
MyDouble = 2345.5678
MyInt = CInt(MyDouble)   ' MyInt is set to 2346.
```

CLng Example

This example uses the **CLng** function to convert values to **Long**.

```
Dim MyDbl1, MyDbl2 As Double
Dim MyLong1, MyLong2 As Long
MyDbl1 = 25427.45
MyDbl2 = 25427.55
MyLong1 = CLng(MyDbl1)   ' MyLong1 contains 25427.
MyLong2 = CLng(MyDbl2)   ' MyLong2 contains 25428.
```

CObj Example

This example uses the **CObj** function to convert a numeric value to **Object**. The **Object** variable itself contains only a four-byte pointer, which points to the **Double** value assigned to it.

```
Dim MyDouble As Double
Dim MyObject As Object
```

```
MyDouble = 2.7182818284
MyObject = CObj(MyDouble)   ' Double value is pointed to by MyObject.
```

CShort Example

This example uses the **CShort** function to convert a numeric value to **Short**.

```
Dim MyByte as Byte
Dim MyShort as Short
MyByte = 100
MyShort = CShort(MyByte)   ' Convert to Short.
```

CSng Example

This example uses the **CSng** function to convert values to **Single**.

```
Dim MyDouble1, MyDouble2 As Double
Dim MySingle1, MySingle2 As Single
MyDouble1 = 75.3421105
MyDouble2 = 75.3421567
MySingle1 = CSng(MyDouble1)   ' MySingle1 is set to 75.34211.
MySingle2 = CSng(MyDouble2)   ' MySingle2 is set to 75.34216.
```

CStr Example

This example uses the **CStr** function to convert a numeric value to **String**.

```
Dim MyDouble As Double
Dim MyString As String
MyDouble = 437.324
MyString = CStr(MyDouble)   ' MyString is set to "437.324".
```

This example uses the **CStr** function to convert **Date** values to **String** values.

```
Dim MyDate As Date
Dim MyString As String
' ...
MyDate = #February 12, 1969 00:00:00#   ' INVALID format.
' Date literals must be in the format #m/d/yyyy# or they are invalid.
' ...
MyDate = #2/12/69 00:00:00#   ' Time is midnight.
' The neutral time value of 00:00:00 is suppressed in the conversion.
MyString = CStr(MyDate)   ' MyString is set to "2/12/1969".
' ...
MyDate = #2/12/69 00:00:01#   ' Time is one second past midnight.
' The time component becomes part of the converted value.
MyString = CStr(MyDate)   ' MyString is set to "2/12/1969 12:00:01 AM".
```

CStr always renders a **Date** value in the standard short format for the current locale, for example, "6/15/2003 4:35:47 PM".

See Also

Conversion Functions | CType Function | Asc, AscW Functions | Chr, ChrW Functions | Format Function | Hex Function | Oct Function | Str Function | Val Function | DatePart Function | Returns for CStr | Type Conversions

Command Function

Returns the argument portion of the command line used to launch Visual Basic or an executable program developed with Visual Basic.

```
Public Function Command() As String
```

Remarks

For applications developed with Visual Basic and compiled to an .exe file, the **Command** function returns any arguments that appear after the name of the application on the command line, as in this example:

```
MyApp cmdlineargs
```

Example

This example uses the **Command** function to return the command-line arguments in an object containing an array.

```
Function GetCommandLineArgs() As String()
   ' Declare variables.
   Dim separators As String = " "
   Dim commands As String = Microsoft.VisualBasic.Command()
   Dim args() As String = commands.Split(separators.ToCharArray)
   Return args
End Function
```

Requirements

Namespace: Microsoft.VisualBasic

Module: Interaction

Assembly: Microsoft.Visual Basic .NET Runtime (in MicrosoftVisualBasic.dll)

See Also

Visual Basic Run-Time Library Members | Visual Basic Compiler Options

Conversion Functions

Asc Function

AscW Function

CBool Function

CByte Function

CChar Function

CDate Function

CDbl Function

CDec Function

Chr Function

ChrW Function

CInt Function

CLng Function

CObj Function

CShort Function

CSng Function

CStr Function

CType Function

Format Function

Hex Function

Oct Function

Str Function

Val Function

See Also

Type Conversion Functions

Math Functions

The math functions in Visual Basic 6 have been replaced by equivalent methods in the **System.Math** class of the .NET Framework.

Remarks

The .NET Framework math methods are functionally identical to their Visual Basic 6 counterparts, although some have slightly different names. For example, the .NET Framework equivalent of the Visual Basic 6 Atn function is **Atan**. The following table lists the Visual Basic 6 math function names and the equivalent .NET Framework methods.

Visual Basic 6 function	Visual Basic .NET method	Description
Abs	Math.Abs Method	Returns the absolute value of a specified number.
Atn	Math.Atan Method	Returns a **Double** value containing the angle whose tangent is the specified number.
Cos	Math.Cos Method	Returns a **Double** value containing the cosine of the specified angle.
Exp	Math.Exp Method	Returns a **Double** value containing e (the base of natural logarithms) raised to the specified power.
Log	Math.Log Method	Returns a **Double** value containing the logarithm of a specified number. This method is overloaded and can return either the natural (base e) logarithm of a specified number or the logarithm of a specified number in a specified base.
Round	Math.Round Method	Returns a **Double** value containing the number nearest the specified value. Additional round functions are available as methods of the intrinsic types such as Decimal.Round Method.
Sgn	Math.Sign Method	Returns an **Integer** value indicating the sign of a number.
Sin	Math.Sin Method	Returns a **Double** value specifying the sine of an angle.
Sqr	Math.Sqrt Method	Returns a **Double** value specifying the square root of a number.
Tan	Math.Tan Method	Returns a **Double** value containing the tangent of an angle.

In addition, the .NET Framework math class provides constants and other static methods for trigonometric, logarithmic, and other common mathematical functions. All of these can be used in a Visual Basic program.

To use these functions without qualification, import the **System.Math** namespace into your project by adding the following code to the top of the source code:

```
Imports System.Math
```

Requirements

Class: Math Class

Abs Example

This example uses the **Abs** method of the **Math** class to compute the absolute value of a number.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim MyNumber As Double
```

```
MyNumber = Abs(50.3)    ' Returns 50.3.
MyNumber = Abs(-50.3)   ' Returns 50.3.
```

Atan Example

This example uses the **Atan** method of the **Math** class to calculate the value of pi.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim pi As Double
pi = 4 * Atan(1)   ' Calculate the value of pi.
```

Cos Example

This example uses the **Cos** method of the **Math** class to return the cosine of an angle.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim MyAngle, MySecant As Double
MyAngle = 1.3   ' Define angle in radians.
MySecant = 1 / Cos(MyAngle)   ' Calculate secant.
```

Exp Example

This example uses the **Exp** method of the **Math** class to return *e* raised to a power.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim MyAngle, MyHSin As Double
' Define angle in radians.
MyAngle = 1.3
' Calculate hyperbolic sine.
MyHSin = (Exp(MyAngle) - Exp(-1 * MyAngle)) / 2
```

Log Example

This example uses the **Log** method of the **Math** class to return the natural logarithm of a number.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim MyAngle, MyLog As Double
' Define angle in radians.
MyAngle = 1.3
' Calculate inverse hyperbolic sine.
MyLog = Log(MyAngle + Sqrt(MyAngle * MyAngle + 1))
```

Round Example

This example uses the **Round** method of the **Math** class to round a number to the nearest integer.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim MyVar1 As Double = 2.8
Dim MyVar2 As Double
MyVar2 =Round(MyVar1)   ' Returns 3.
```

Sign Example

This example uses the **Sign** method of the **Math** class to determine the sign of a number.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim MyVar1, MyVar2, MyVar3 As Double
Dim MySign As Integer
MyVar1 = 12
MyVar2 = -2.4
MyVar3 = 0
MySign = Sign(MyVar1)   ' Returns 1.
MySign = Sign(MyVar2)   ' Returns -1.
MySign = Sign(MyVar3)   ' Returns 0.
```

Sin Example

This example uses the **Sin** method of the **Math** class to return the sine of an angle.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim MyAngle, MyCosecant As Double
MyAngle = 1.3   ' Define angle in radians.
MyCosecant = 1 / Sin(MyAngle)   ' Calculate cosecant.
```

Sqrt Example

This example uses the **Sqrt** method of the **Math** class to calculate the square root of a number.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim MySqr As Double
MySqr = Sqrt(4)    ' Returns 2.
MySqr = Sqrt(23)   ' Returns 4.79583152331272.
MySqr = Sqrt(0)    ' Returns 0.
MySqr = Sqrt(-4)   ' Returns NaN (not a number).
```

Tan Example

This example uses the **Tan** method of the **Math** class to return the tangent of an angle.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim MyAngle, MyCotangent As Double
MyAngle = 1.3   ' Define angle in radians.
MyCotangent = 1 / Tan(MyAngle)   ' Calculate cotangent.
```

Requirements

Namespace: System.Math

Assembly: mscorlib (in mscorlib.dll)

See Also

Rnd Function | Randomize Statement | Derived Math Functions | Double.NaN Field

CreateObject Function

Creates and returns a reference to a COM object. **CreateObject** cannot be used to create instances of classes in Visual Basic unless those classes are explicitly exposed as COM components.

```
Public Shared Function CreateObject( _
   ByVal ProgId As String, _
   Optional ByVal ServerName As String = "" _
) As Object
```

Parameters

ProgId
Required. **String**. The program ID of the object to create.
ServerName
Optional. **String**. The name of the network server where the object will be created. If *ServerName* is an empty string (""), the local machine is used.

Exceptions/Errors

Exception type	Error number	Condition
Exception	429	*ProgId* not found or not supplied.
Exception	462	Server is unavailable
FileNotFoundException	53	No object of the specified type exists .

Remarks

To create an instance of a COM component, assign the object returned by **CreateObject** to an object variable:

```
Sub CreateADODB()
   Dim adoApp As Object
   adoApp = CreateObject("ADODB.Connection")
End Sub
```

The type of object variable you use to store the returned object can affect your application's performance. Declaring an object variable with the **As Object** clause creates a variable that can contain a reference to any type of object. However, access to the object through that variable is late bound; that is, the binding occurs when your program is run. There are many reasons you should avoid late binding, including slower application performance. To create an object variable that results in early binding, that is, binding when the program is compiled, add a reference to the type library for your object from the **COM** tab of the **Add Reference** dialog box on the **Project** menu and declare the object variable of the specific type of your object. In most cases, it is more efficient to use the **Dim** statement and a primary interop assembly to create objects than it is to use the **CreateObject** function.

Another issue is that COM objects use unmanaged code — code without the benefit of the common language runtime. There is a fair degree of complexity involved in mixing the managed code of Visual Basic .NET with unmanaged code from COM. When you add a reference to a COM object, a search is made for a Primary Interop Assembly for that library; if one is found then it is used. If none is found Visual Basic .NET creates an interoperability assembly that contains local interoperability classes for each class in the COM library. For more information, see COM Interoperability in Visual Basic and Visual C#. You should generally use strongly bound objects and primary interop assemblies whenever possible. The examples below use the **CreateObject** function with Microsoft Office objects for demonstration purposes only. However, these objects are easier to use and more reliable when used with the appropriate primary interop assembly.

You can create an object on a remote networked computer by passing the name of the computer to the *ServerName* argument of the **CreateObject** function. That name is the same as the Machine Name portion of a share name: for a share named "\\MyServer\Public," *ServerName* is "MyServer."

> **Note** Refer to COM documentation (see Microsoft Developer Network) for additional information on making an application accessible on a remote networked computer. You may need to add a registry key for your application.

The following code returns the version number of an instance of Excel running on a remote computer named `MyServer`:

```
Sub CreateRemoteExcelObj()
   Dim xlApp As Object
   ' Replace string "\\MyServer" with name of the remote computer.
   xlApp = CreateObject("Excel.Application", "\\MyServer")
   MsgBox(xlApp.Version)
End Sub
```

If the remote server name is incorrect, or if it is unavailable, a run-time error occurs.

> **Note** Use **CreateObject** when there is no current instance of the object. If an instance of the object is already running, a new instance is started, and an object of the specified type is created. To use the current instance, or to start the application and have it load a file, use the **GetObject** function. If an object has registered itself as a single-instance object, only one instance of the object is created, no matter how many times **CreateObject** is executed.

Example

This example uses the **CreateObject** function to create a Microsoft Excel worksheet and saves the worksheet to a file. To use this example, Excel must be installed on the computer where this program will run, and you must add a reference to the type library from the **COM** tab of the **Add Reference** dialog box on the **Project** menu. The name of the type library will vary depending on the version of Excel installed on your computer. For example, the type library for Microsoft Excel 2002 is named **Microsoft Excel 10.0 Object Library**.

```
Sub TestExcel()
   Dim xlApp As Excel.Application
   Dim xlBook As Excel.Workbook
   Dim xlSheet As Excel.Worksheet
   xlApp = CType(CreateObject("Excel.Application"), Excel.Application)
   xlBook = CType(xlApp.Workbooks.Add, Excel.Workbook)
   xlSheet = CType(xlBook.Worksheets(1), Excel.Worksheet)
   ' Place some text in the second row of the sheet.
   xlSheet.Cells(2, 2) = "This is column B row 2"
   ' Show the sheet.
   xlSheet.Application.Visible = True
   ' Save the sheet to C:\Test.xls directory.
   xlSheet.SaveAs("C:\Test.xls")
   ' Optionally, you can call xlApp.Quit to close the work sheet.
End Sub
```

Smart Device Developer Notes

This function is not supported.

Requirements

Namespace: Microsoft.VisualBasic

Module: Interaction

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

GetObject Function | Dim Statement | Declare Statement | Exception Class | FileNotFoundException Class | COM Interoperability in Visual Basic and Visual C# | Interoperating with Unmanaged Code

Type Conversion Functions

These functions are compiled inline, meaning the conversion code is part of the code that evaluates the expression. Execution is faster because there is no call to a procedure to accomplish the conversion. Each function coerces an expression to a specific data type.

```
CBool(expression)
CByte(expression)
CChar(expression)
CDate(expression)
CDbl(expression)
CDec(expression)
CInt(expression)
CLng(expression)
CObj(expression)
CShort(expression)
CSng(expression)
CStr(expression)
```

Part

expression
Required. Any **String** expression or numeric expression.

Return Types

The function name determines the return type, as shown in the following table:

Function name	Return type	Range for *expression* argument
CBool	**Boolean**	Any valid **String** or numeric expression.
CByte	**Byte**	0 through 255; fractions are rounded.
CChar	**Char**	Any valid **String** expression; value can be 0 through 65535.
CDate	**Date**	Any valid representation of a date and time.
CDbl	**Double**	-1.79769313486231E+308 through -4.94065645841247E-324 for negative values; 4.94065645841247E-324 through 1.79769313486231E+308 for positive values.
CDec	**Decimal**	+/-79,228,162,514,264,337,593,543,950,335 for zero-scaled numbers, that is, numbers with no decimal places. For numbers with 28 decimal places, the range is +/-7.9228162514264337593543950335. The smallest possible non-zero number is 0.0000000000000000000000000001.
CInt	**Integer**	-2,147,483,648 through 2,147,483,647; fractions are rounded.
CLng	**Long**	-9,223,372,036,854,775,808 through 9,223,372,036,854,775,807; fractions are rounded.
CObj	**Object**	Any valid expression.
CShort	**Short**	-32,768 through 32,767; fractions are rounded.
CSng	**Single**	-3.402823E+38 through -1.401298E-45 for negative values; 1.401298E-45 through 3.402823E+38 for positive values.
CStr	**String**	Returns for **CStr** depend on the *expression* argument. See Returns for CStr.

Remarks

If the *expression* passed to the function is outside the range of the data type to which it is being converted, an error occurs.

In general, you can use the data type conversion functions to force the result of some operation to a particular data type rather than the default data type. For example, use **CDec** to force decimal arithmetic in cases where single-precision, double-precision, or integer arithmetic normally would occur.

When the fractional part is exactly 0.5, **CInt** and **CLng** always round it to the nearest even number. For example, 0.5 rounds to 0

and 1.5 rounds to 2. **CInt** and **CLng** differ from the **Fix** and **Int** functions, which truncate, rather than round, the fractional part of a number. Also, **Fix** and **Int** always return a value of the same type as is passed in.

Use the **IsDate** function to determine if a value can be converted to a date and time. **CDate** recognizes date literals and time literals as well as numbers that fall within the range of acceptable dates.

CDate recognizes date formats according to the locale setting of your system. You must provide the day, month, and year in the correct order for your locale, or the date may not be interpreted correctly. A long date format is not recognized if it contains a day-of-the-week string, such as "Wednesday".

The **Date** data type always contains both date and time information. For purposes of type conversion, Visual Basic .NET considers 1/1/1 (January 1 of the year 1) to be a neutral value for the date, and 00:00:00 (midnight) to be a neutral value for the time. If you convert a **Date** value to a string, **CStr** does not include neutral values in the resulting string. For example, if you convert `#January 1, 0001 9:30:00#` to a string, the result is "9:30:00 AM"; the date information is suppressed. However, the date information is still present in the original **Date** value and can be recovered with functions such as **DatePart**.

The **CType** function takes a second argument, *typename*, and coerces *expression* to *typename*, where *typename* can be any data type, structure, class, or interface. For more information, see CType Function.

CBool Example

This example uses the **CBool** function to convert expressions to **Boolean** values. If an expression evaluates to a nonzero value, **CBool** returns **True**; otherwise, it returns **False**.

```
Dim A, B, C As Integer
Dim Check As Boolean
A = 5
B = 5
Check = CBool(A = B)   ' Check is set to True.
' ...
C = 0
Check = CBool(C)   ' Check is set to False.
```

CByte Example

This example uses the **CByte** function to convert an expression to a **Byte**.

```
Dim MyDouble As Double
Dim MyByte As Byte
MyDouble = 125.5678
MyByte = CByte(MyDouble)   ' MyByte is set to 126.
```

CChar Example

This example uses the **CChar** function to convert a the first character of a **String** expression to a **Char** type.

```
Dim MyString As String
Dim MyChar As Char
MyString = "BCD"   ' CChar converts only first character of string.
MyChar = CChar(MyString)   ' MyChar is set to "B".
```

The input argument to **CChar** must be of data type **String**. You cannot use **CChar** to convert a number to a character, because **CChar** cannot accept a numeric data type. This example obtains a number representing a code point (character code) and converts it to the corresponding character. It uses **InputBox** to obtain the string of digits, **CInt** to convert the string to type **Integer**, and **ChrW** to convert the number to type **Char**.

```
Dim MyDigits As String   ' Input string of digits to be converted.
Dim CodePoint As Integer   ' Number to be represented as a character.
Dim MyChar As Char
MyDigits = InputBox("Enter code point of character:")
CodePoint = CInt(MyDigits)   ' Convert entire string to Integer.
MyChar = ChrW(CodePoint)   ' MyChar is set to Char value of code point.
```

CDate Example

This example uses the **CDate** function to convert strings to **Date** values. In general, hard-coding dates and times as strings (as shown in this example) is not recommended. Use date literals and time literals, such as #Feb 12, 1969# and #4:45:23 PM#, instead.

```
Dim MyDateString, MyTimeString As String
Dim MyDate, MyTime As Date
MyDateString = "February 12, 1969"
MyTimeString = "4:35:47 PM"
' ...
MyDate = CDate(MyDateString)   ' Convert to Date data type.
MyTime = CDate(MyTimeString)   ' Convert to Date data type.
```

CDbl Example

This example uses the **CDbl** function to convert an expression to **Double**.

```
Dim MyDec As Decimal
Dim MyDouble As Double
MyDec = 234.456784D    ' Literal type character D makes MyDec a Decimal.
MyDouble = CDbl(MyDec * 8.2D * 0.01D)   ' Convert result to a Double.
```

CDec Example

This example uses the **CDec** function to convert a numeric value to **Decimal**.

```
Dim MyDouble As Double
Dim MyDecimal As Decimal
MyDouble = 10000000.0587
MyDecimal = CDec(MyDouble)   ' Convert to Decimal.
```

CInt Example

This example uses the **CInt** function to convert a value to **Integer**.

```
Dim MyDouble As Double
Dim MyInt As Integer
MyDouble = 2345.5678
MyInt = CInt(MyDouble)   ' MyInt is set to 2346.
```

CLng Example

This example uses the **CLng** function to convert values to **Long**.

```
Dim MyDbl1, MyDbl2 As Double
Dim MyLong1, MyLong2 As Long
MyDbl1 = 25427.45
MyDbl2 = 25427.55
MyLong1 = CLng(MyDbl1)   ' MyLong1 contains 25427.
MyLong2 = CLng(MyDbl2)   ' MyLong2 contains 25428.
```

CObj Example

This example uses the **CObj** function to convert a numeric value to **Object**. The **Object** variable itself contains only a four-byte pointer, which points to the **Double** value assigned to it.

```
Dim MyDouble As Double
Dim MyObject As Object
```

```
MyDouble = 2.7182818284
MyObject = CObj(MyDouble)   ' Double value is pointed to by MyObject.
```

CShort Example

This example uses the **CShort** function to convert a numeric value to **Short**.

```
Dim MyByte as Byte
Dim MyShort as Short
MyByte = 100
MyShort = CShort(MyByte)   ' Convert to Short.
```

CSng Example

This example uses the **CSng** function to convert values to **Single**.

```
Dim MyDouble1, MyDouble2 As Double
Dim MySingle1, MySingle2 As Single
MyDouble1 = 75.3421105
MyDouble2 = 75.3421567
MySingle1 = CSng(MyDouble1)   ' MySingle1 is set to 75.34211.
MySingle2 = CSng(MyDouble2)   ' MySingle2 is set to 75.34216.
```

CStr Example

This example uses the **CStr** function to convert a numeric value to **String**.

```
Dim MyDouble As Double
Dim MyString As String
MyDouble = 437.324
MyString = CStr(MyDouble)   ' MyString is set to "437.324".
```

This example uses the **CStr** function to convert **Date** values to **String** values.

```
Dim MyDate As Date
Dim MyString As String
' ...
MyDate = #February 12, 1969 00:00:00#   ' INVALID format.
' Date literals must be in the format #m/d/yyyy# or they are invalid.
' ...
MyDate = #2/12/69 00:00:00#   ' Time is midnight.
' The neutral time value of 00:00:00 is suppressed in the conversion.
MyString = CStr(MyDate)   ' MyString is set to "2/12/1969".
' ...
MyDate = #2/12/69 00:00:01#   ' Time is one second past midnight.
' The time component becomes part of the converted value.
MyString = CStr(MyDate)   ' MyString is set to "2/12/1969 12:00:01 AM".
```

CStr always renders a **Date** value in the standard short format for the current locale, for example, "6/15/2003 4:35:47 PM".

See Also

Conversion Functions | CType Function | Asc, AscW Functions | Chr, ChrW Functions | Format Function | Hex Function | Oct Function | Str Function | Val Function | DatePart Function | Returns for CStr | Type Conversions

Type Conversion Functions

These functions are compiled inline, meaning the conversion code is part of the code that evaluates the expression. Execution is faster because there is no call to a procedure to accomplish the conversion. Each function coerces an expression to a specific data type.

```
CBool(expression)
CByte(expression)
CChar(expression)
CDate(expression)
CDbl(expression)
CDec(expression)
CInt(expression)
CLng(expression)
CObj(expression)
CShort(expression)
CSng(expression)
CStr(expression)
```

Part

expression
Required. Any **String** expression or numeric expression.

Return Types

The function name determines the return type, as shown in the following table:

Function name	Return type	Range for *expression* argument
CBool	**Boolean**	Any valid **String** or numeric expression.
CByte	**Byte**	0 through 255; fractions are rounded.
CChar	**Char**	Any valid **String** expression; value can be 0 through 65535.
CDate	**Date**	Any valid representation of a date and time.
CDbl	**Double**	-1.79769313486231E+308 through -4.94065645841247E-324 for negative values; 4.94065645841247E-324 through 1.79769313486231E+308 for positive values.
CDec	**Decimal**	+/-79,228,162,514,264,337,593,543,950,335 for zero-scaled numbers, that is, numbers with no decimal places. For numbers with 28 decimal places, the range is +/-7.9228162514264337593543950335. The smallest possible non-zero number is 0.0000000000000000000000000001.
CInt	**Integer**	-2,147,483,648 through 2,147,483,647; fractions are rounded.
CLng	**Long**	-9,223,372,036,854,775,808 through 9,223,372,036,854,775,807; fractions are rounded.
CObj	**Object**	Any valid expression.
CShort	**Short**	-32,768 through 32,767; fractions are rounded.
CSng	**Single**	-3.402823E+38 through -1.401298E-45 for negative values; 1.401298E-45 through 3.402823E+38 for positive values.
CStr	**String**	Returns for **CStr** depend on the *expression* argument. See Returns for CStr.

Remarks

If the *expression* passed to the function is outside the range of the data type to which it is being converted, an error occurs.

In general, you can use the data type conversion functions to force the result of some operation to a particular data type rather than the default data type. For example, use **CDec** to force decimal arithmetic in cases where single-precision, double-precision, or integer arithmetic normally would occur.

When the fractional part is exactly 0.5, **CInt** and **CLng** always round it to the nearest even number. For example, 0.5 rounds to 0

and 1.5 rounds to 2. **CInt** and **CLng** differ from the **Fix** and **Int** functions, which truncate, rather than round, the fractional part of a number. Also, **Fix** and **Int** always return a value of the same type as is passed in.

Use the **IsDate** function to determine if a value can be converted to a date and time. **CDate** recognizes date literals and time literals as well as numbers that fall within the range of acceptable dates.

CDate recognizes date formats according to the locale setting of your system. You must provide the day, month, and year in the correct order for your locale, or the date may not be interpreted correctly. A long date format is not recognized if it contains a day-of-the-week string, such as "Wednesday".

The **Date** data type always contains both date and time information. For purposes of type conversion, Visual Basic .NET considers 1/1/1 (January 1 of the year 1) to be a neutral value for the date, and 00:00:00 (midnight) to be a neutral value for the time. If you convert a **Date** value to a string, **CStr** does not include neutral values in the resulting string. For example, if you convert `#January 1, 0001 9:30:00#` to a string, the result is "9:30:00 AM"; the date information is suppressed. However, the date information is still present in the original **Date** value and can be recovered with functions such as **DatePart**.

The **CType** function takes a second argument, *typename*, and coerces *expression* to *typename*, where *typename* can be any data type, structure, class, or interface. For more information, see CType Function.

CBool Example

This example uses the **CBool** function to convert expressions to **Boolean** values. If an expression evaluates to a nonzero value, **CBool** returns **True**; otherwise, it returns **False**.

```
Dim A, B, C As Integer
Dim Check As Boolean
A = 5
B = 5
Check = CBool(A = B)   ' Check is set to True.
' ...
C = 0
Check = CBool(C)   ' Check is set to False.
```

CByte Example

This example uses the **CByte** function to convert an expression to a **Byte**.

```
Dim MyDouble As Double
Dim MyByte As Byte
MyDouble = 125.5678
MyByte = CByte(MyDouble)   ' MyByte is set to 126.
```

CChar Example

This example uses the **CChar** function to convert a the first character of a **String** expression to a **Char** type.

```
Dim MyString As String
Dim MyChar As Char
MyString = "BCD"   ' CChar converts only first character of string.
MyChar = CChar(MyString)   ' MyChar is set to "B".
```

The input argument to **CChar** must be of data type **String**. You cannot use **CChar** to convert a number to a character, because **CChar** cannot accept a numeric data type. This example obtains a number representing a code point (character code) and converts it to the corresponding character. It uses **InputBox** to obtain the string of digits, **CInt** to convert the string to type **Integer**, and **ChrW** to convert the number to type **Char**.

```
Dim MyDigits As String   ' Input string of digits to be converted.
Dim CodePoint As Integer   ' Number to be represented as a character.
Dim MyChar As Char
MyDigits = InputBox("Enter code point of character:")
CodePoint = CInt(MyDigits)   ' Convert entire string to Integer.
MyChar = ChrW(CodePoint)   ' MyChar is set to Char value of code point.
```

CDate Example

This example uses the **CDate** function to convert strings to **Date** values. In general, hard-coding dates and times as strings (as shown in this example) is not recommended. Use date literals and time literals, such as #Feb 12, 1969# and #4:45:23 PM#, instead.

```
Dim MyDateString, MyTimeString As String
Dim MyDate, MyTime As Date
MyDateString = "February 12, 1969"
MyTimeString = "4:35:47 PM"
' ...
MyDate = CDate(MyDateString)   ' Convert to Date data type.
MyTime = CDate(MyTimeString)   ' Convert to Date data type.
```

CDbl Example

This example uses the **CDbl** function to convert an expression to **Double**.

```
Dim MyDec As Decimal
Dim MyDouble As Double
MyDec = 234.456784D    ' Literal type character D makes MyDec a Decimal.
MyDouble = CDbl(MyDec * 8.2D * 0.01D)   ' Convert result to a Double.
```

CDec Example

This example uses the **CDec** function to convert a numeric value to **Decimal**.

```
Dim MyDouble As Double
Dim MyDecimal As Decimal
MyDouble = 10000000.0587
MyDecimal = CDec(MyDouble)   ' Convert to Decimal.
```

CInt Example

This example uses the **CInt** function to convert a value to **Integer**.

```
Dim MyDouble As Double
Dim MyInt As Integer
MyDouble = 2345.5678
MyInt = CInt(MyDouble)   ' MyInt is set to 2346.
```

CLng Example

This example uses the **CLng** function to convert values to **Long**.

```
Dim MyDbl1, MyDbl2 As Double
Dim MyLong1, MyLong2 As Long
MyDbl1 = 25427.45
MyDbl2 = 25427.55
MyLong1 = CLng(MyDbl1)   ' MyLong1 contains 25427.
MyLong2 = CLng(MyDbl2)   ' MyLong2 contains 25428.
```

CObj Example

This example uses the **CObj** function to convert a numeric value to **Object**. The **Object** variable itself contains only a four-byte pointer, which points to the **Double** value assigned to it.

```
Dim MyDouble As Double
Dim MyObject As Object
```

```
MyDouble = 2.7182818284
MyObject = CObj(MyDouble)   ' Double value is pointed to by MyObject.
```

CShort Example

This example uses the **CShort** function to convert a numeric value to **Short**.

```
Dim MyByte as Byte
Dim MyShort as Short
MyByte = 100
MyShort = CShort(MyByte)   ' Convert to Short.
```

CSng Example

This example uses the **CSng** function to convert values to **Single**.

```
Dim MyDouble1, MyDouble2 As Double
Dim MySingle1, MySingle2 As Single
MyDouble1 = 75.3421105
MyDouble2 = 75.3421567
MySingle1 = CSng(MyDouble1)   ' MySingle1 is set to 75.34211.
MySingle2 = CSng(MyDouble2)   ' MySingle2 is set to 75.34216.
```

CStr Example

This example uses the **CStr** function to convert a numeric value to **String**.

```
Dim MyDouble As Double
Dim MyString As String
MyDouble = 437.324
MyString = CStr(MyDouble)   ' MyString is set to "437.324".
```

This example uses the **CStr** function to convert **Date** values to **String** values.

```
Dim MyDate As Date
Dim MyString As String
' ...
MyDate = #February 12, 1969 00:00:00#   ' INVALID format.
' Date literals must be in the format #m/d/yyyy# or they are invalid.
' ...
MyDate = #2/12/69 00:00:00#   ' Time is midnight.
' The neutral time value of 00:00:00 is suppressed in the conversion.
MyString = CStr(MyDate)   ' MyString is set to "2/12/1969".
' ...
MyDate = #2/12/69 00:00:01#   ' Time is one second past midnight.
' The time component becomes part of the converted value.
MyString = CStr(MyDate)   ' MyString is set to "2/12/1969 12:00:01 AM".
```

CStr always renders a **Date** value in the standard short format for the current locale, for example, "6/15/2003 4:35:47 PM".

See Also

Conversion Functions | CType Function | Asc, AscW Functions | Chr, ChrW Functions | Format Function | Hex Function | Oct Function | Str Function | Val Function | DatePart Function | Returns for CStr | Type Conversions

Type Conversion Functions

These functions are compiled inline, meaning the conversion code is part of the code that evaluates the expression. Execution is faster because there is no call to a procedure to accomplish the conversion. Each function coerces an expression to a specific data type.

```
CBool(expression)
CByte(expression)
CChar(expression)
CDate(expression)
CDbl(expression)
CDec(expression)
CInt(expression)
CLng(expression)
CObj(expression)
CShort(expression)
CSng(expression)
CStr(expression)
```

Part

expression
Required. Any **String** expression or numeric expression.

Return Types

The function name determines the return type, as shown in the following table:

Function name	Return type	Range for *expression* argument
CBool	**Boolean**	Any valid **String** or numeric expression.
CByte	**Byte**	0 through 255; fractions are rounded.
CChar	**Char**	Any valid **String** expression; value can be 0 through 65535.
CDate	**Date**	Any valid representation of a date and time.
CDbl	**Double**	-1.79769313486231E+308 through -4.94065645841247E-324 for negative values; 4.94065645841247E-324 through 1.79769313486231E+308 for positive values.
CDec	**Decimal**	+/-79,228,162,514,264,337,593,543,950,335 for zero-scaled numbers, that is, numbers with no decimal places. For numbers with 28 decimal places, the range is +/-7.9228162514264337593543950335. The smallest possible non-zero number is 0.0000000000000000000000000001.
CInt	**Integer**	-2,147,483,648 through 2,147,483,647; fractions are rounded.
CLng	**Long**	-9,223,372,036,854,775,808 through 9,223,372,036,854,775,807; fractions are rounded.
CObj	**Object**	Any valid expression.
CShort	**Short**	-32,768 through 32,767; fractions are rounded.
CSng	**Single**	-3.402823E+38 through -1.401298E-45 for negative values; 1.401298E-45 through 3.402823E+38 for positive values.
CStr	**String**	Returns for **CStr** depend on the *expression* argument. See Returns for CStr.

Remarks

If the *expression* passed to the function is outside the range of the data type to which it is being converted, an error occurs.

In general, you can use the data type conversion functions to force the result of some operation to a particular data type rather than the default data type. For example, use **CDec** to force decimal arithmetic in cases where single-precision, double-precision, or integer arithmetic normally would occur.

When the fractional part is exactly 0.5, **CInt** and **CLng** always round it to the nearest even number. For example, 0.5 rounds to 0

and 1.5 rounds to 2. **CInt** and **CLng** differ from the **Fix** and **Int** functions, which truncate, rather than round, the fractional part of a number. Also, **Fix** and **Int** always return a value of the same type as is passed in.

Use the **IsDate** function to determine if a value can be converted to a date and time. **CDate** recognizes date literals and time literals as well as numbers that fall within the range of acceptable dates.

CDate recognizes date formats according to the locale setting of your system. You must provide the day, month, and year in the correct order for your locale, or the date may not be interpreted correctly. A long date format is not recognized if it contains a day-of-the-week string, such as "Wednesday".

The **Date** data type always contains both date and time information. For purposes of type conversion, Visual Basic .NET considers 1/1/1 (January 1 of the year 1) to be a neutral value for the date, and 00:00:00 (midnight) to be a neutral value for the time. If you convert a **Date** value to a string, **CStr** does not include neutral values in the resulting string. For example, if you convert `#January 1, 0001 9:30:00#` to a string, the result is "9:30:00 AM"; the date information is suppressed. However, the date information is still present in the original **Date** value and can be recovered with functions such as **DatePart**.

The **CType** function takes a second argument, *typename*, and coerces *expression* to *typename*, where *typename* can be any data type, structure, class, or interface. For more information, see CType Function.

CBool Example

This example uses the **CBool** function to convert expressions to **Boolean** values. If an expression evaluates to a nonzero value, **CBool** returns **True**; otherwise, it returns **False**.

```
Dim A, B, C As Integer
Dim Check As Boolean
A = 5
B = 5
Check = CBool(A = B)   ' Check is set to True.
' ...
C = 0
Check = CBool(C)   ' Check is set to False.
```

CByte Example

This example uses the **CByte** function to convert an expression to a **Byte**.

```
Dim MyDouble As Double
Dim MyByte As Byte
MyDouble = 125.5678
MyByte = CByte(MyDouble)   ' MyByte is set to 126.
```

CChar Example

This example uses the **CChar** function to convert a the first character of a **String** expression to a **Char** type.

```
Dim MyString As String
Dim MyChar As Char
MyString = "BCD"   ' CChar converts only first character of string.
MyChar = CChar(MyString)   ' MyChar is set to "B".
```

The input argument to **CChar** must be of data type **String**. You cannot use **CChar** to convert a number to a character, because **CChar** cannot accept a numeric data type. This example obtains a number representing a code point (character code) and converts it to the corresponding character. It uses **InputBox** to obtain the string of digits, **CInt** to convert the string to type **Integer**, and **ChrW** to convert the number to type **Char**.

```
Dim MyDigits As String   ' Input string of digits to be converted.
Dim CodePoint As Integer   ' Number to be represented as a character.
Dim MyChar As Char
MyDigits = InputBox("Enter code point of character:")
CodePoint = CInt(MyDigits)   ' Convert entire string to Integer.
MyChar = ChrW(CodePoint)   ' MyChar is set to Char value of code point.
```

CDate Example

This example uses the **CDate** function to convert strings to **Date** values. In general, hard-coding dates and times as strings (as shown in this example) is not recommended. Use date literals and time literals, such as #Feb 12, 1969# and #4:45:23 PM#, instead.

```
Dim MyDateString, MyTimeString As String
Dim MyDate, MyTime As Date
MyDateString = "February 12, 1969"
MyTimeString = "4:35:47 PM"
' ...
MyDate = CDate(MyDateString)   ' Convert to Date data type.
MyTime = CDate(MyTimeString)   ' Convert to Date data type.
```

CDbl Example

This example uses the **CDbl** function to convert an expression to **Double**.

```
Dim MyDec As Decimal
Dim MyDouble As Double
MyDec = 234.456784D    ' Literal type character D makes MyDec a Decimal.
MyDouble = CDbl(MyDec * 8.2D * 0.01D)   ' Convert result to a Double.
```

CDec Example

This example uses the **CDec** function to convert a numeric value to **Decimal**.

```
Dim MyDouble As Double
Dim MyDecimal As Decimal
MyDouble = 10000000.0587
MyDecimal = CDec(MyDouble)   ' Convert to Decimal.
```

CInt Example

This example uses the **CInt** function to convert a value to **Integer**.

```
Dim MyDouble As Double
Dim MyInt As Integer
MyDouble = 2345.5678
MyInt = CInt(MyDouble)   ' MyInt is set to 2346.
```

CLng Example

This example uses the **CLng** function to convert values to **Long**.

```
Dim MyDbl1, MyDbl2 As Double
Dim MyLong1, MyLong2 As Long
MyDbl1 = 25427.45
MyDbl2 = 25427.55
MyLong1 = CLng(MyDbl1)   ' MyLong1 contains 25427.
MyLong2 = CLng(MyDbl2)   ' MyLong2 contains 25428.
```

CObj Example

This example uses the **CObj** function to convert a numeric value to **Object**. The **Object** variable itself contains only a four-byte pointer, which points to the **Double** value assigned to it.

```
Dim MyDouble As Double
Dim MyObject As Object
```

```
MyDouble = 2.7182818284
MyObject = CObj(MyDouble)   ' Double value is pointed to by MyObject.
```

CShort Example

This example uses the **CShort** function to convert a numeric value to **Short**.

```
Dim MyByte as Byte
Dim MyShort as Short
MyByte = 100
MyShort = CShort(MyByte)   ' Convert to Short.
```

CSng Example

This example uses the **CSng** function to convert values to **Single**.

```
Dim MyDouble1, MyDouble2 As Double
Dim MySingle1, MySingle2 As Single
MyDouble1 = 75.3421105
MyDouble2 = 75.3421567
MySingle1 = CSng(MyDouble1)   ' MySingle1 is set to 75.34211.
MySingle2 = CSng(MyDouble2)   ' MySingle2 is set to 75.34216.
```

CStr Example

This example uses the **CStr** function to convert a numeric value to **String**.

```
Dim MyDouble As Double
Dim MyString As String
MyDouble = 437.324
MyString = CStr(MyDouble)   ' MyString is set to "437.324".
```

This example uses the **CStr** function to convert **Date** values to **String** values.

```
Dim MyDate As Date
Dim MyString As String
' ...
MyDate = #February 12, 1969 00:00:00#   ' INVALID format.
' Date literals must be in the format #m/d/yyyy# or they are invalid.
' ...
MyDate = #2/12/69 00:00:00#   ' Time is midnight.
' The neutral time value of 00:00:00 is suppressed in the conversion.
MyString = CStr(MyDate)   ' MyString is set to "2/12/1969".
' ...
MyDate = #2/12/69 00:00:01#   ' Time is one second past midnight.
' The time component becomes part of the converted value.
MyString = CStr(MyDate)   ' MyString is set to "2/12/1969 12:00:01 AM".
```

CStr always renders a **Date** value in the standard short format for the current locale, for example, "6/15/2003 4:35:47 PM".

See Also

Conversion Functions | CType Function | Asc, AscW Functions | Chr, ChrW Functions | Format Function | Hex Function | Oct Function | Str Function | Val Function | DatePart Function | Returns for CStr | Type Conversions

CType Function

Returns the result of explicitly converting an expression to a specified data type, object, structure, class, or interface.

```
CType(expression, typename)
```

Parts

expression
Any valid expression. If the value of *expression* is outside the range allowed by *typename*, an error occurs.

typename
Any expression that is legal within an **As** clause in a **Dim** statement, that is, the name of any data type, object, structure, class, or interface.

Remarks

CType is compiled inline, meaning the conversion code is part of the code that evaluates the expression. Execution is faster because there is no call to a procedure to accomplish the conversion.

Example

This example uses the **CType** function to convert an expression to the specified data type.

```
Dim MyNumber As Long
Dim MyNewType As Single
MyNumber = 1000
MyNewType = CType(MyNumber,Single)   ' MyNewType is set to 1000.0.
```

See Also

Type Conversion Functions

CurDir Function

Returns a string representing the current path.

```
Public Overloads Function CurDir([ ByVal Drive As Char ]) As String
```

Parameter

Drive
Optional. **Char** expression that specifies an existing drive. If no drive is specified, or if *Drive* is a zero-length string (""), **CurDir** returns the path for the current drive.

Exceptions/Errors

Exception type	Error number	Condition
IOException	68	*Drive* is not found.
ArgumentException	68	Invalid *Drive* is specified.

Example

This example uses the **CurDir** function to return the current path.

```
' Assume current path on C drive is "C:\WINDOWS\SYSTEM".
' Assume current path on D drive is "D:\EXCEL".
' Assume C is the current drive.
Dim MyPath As String
MyPath = CurDir()   ' Returns "C:\WINDOWS\SYSTEM".
MyPath = CurDir("C"c)   ' Returns "C:\WINDOWS\SYSTEM".
MyPath = CurDir("D"c)   ' Returns "D:\EXCEL".
```

Smart Device Developer Notes

This function is not supported.

Requirements

Namespace: Microsoft.VisualBasic

Module: FileSystem

Assembly: Microsoft Visual Basic .NET Runtime (in MicrosoftVisualBasic.dll)

See Also

ChDir Function | ChDrive Function | MkDir Function | RmDir Function | IOException Class | ArgumentException Class

D-G Functions

The following table contains a listing of the Visual Basic .NET run-time member functions.

DateAdd	DateDiff	DatePart	DateSerial
DateValue	Day	DDB	DeleteSetting
Derived Math	Dir	Environ	EOF
ErrorToString	Exp	FileAttr	FileClose
FileCopy	FileDateTime	FileGet	FileGetObject
FileLen	FileOpen	FilePut	FilePutObject
FileWidth	Filter	Fix	Format
FormatCurrency	FormatDateTime	FormatNumber	FormatPercent
FreeFile	FV	GetAllSettings	GetAttr
GetChar	GetException	GetObject	GetSetting

See Also

A-C Functions | H-L Functions | M-R Functions | S-Z Functions | Visual Basic Language and Run-Time Reference

DateAdd Function

Returns a **Date** value containing a date and time value to which a specified time interval has been added.

```
Public Overloads Function DateAdd( _
   ByVal Interval As DateInterval, _
   ByVal Number As Double, _
   ByVal DateValue As DateTime _
) As DateTime
```

-or-

```
Public Overloads Function DateAdd( _
   ByVal Interval As String, _
   ByVal Number As Double, _
   ByVal DateValue As Object _
) As DateTime
```

Parameters

Interval
Required. **DateInterval** enumeration value or **String** expression representing the time interval you want to add.

Number
Required. **Double**. Floating-point expression representing the number of intervals you want to add. *Number* can be positive (to get date/time values in the future) or negative (to get date/time values in the past). It can contain a fractional part when *Interval* specifies hours, minutes, or seconds. For other values of *Interval*, any fractional part of *Number* is ignored.

DateValue
Required. **Date**. An expression representing the date and time to which the interval is to be added. *DateValue* itself is not changed in the calling program.

Settings

The *Interval* argument can have one of the following settings.

Enumeration value	String	Unit of time interval to add
DateInterval.Day	d	Day; truncated to integral value
DateInterval.DayOfYear	y	Day; truncated to integral value
DateInterval.Hour	h	Hour; rounded to nearest millisecond
DateInterval.Minute	n	Minute; rounded to nearest millisecond
DateInterval.Month	m	Month; truncated to integral value
DateInterval.Quarter	q	Quarter; truncated to integral value
DateInterval.Second	s	Second; rounded to nearest millisecond
DateInterval.Weekday	w	Day; truncated to integral value
DateInterval.WeekOfYear	ww	Week; truncated to integral value
DateInterval.Year	yyyy	Year; truncated to integral value

Exceptions/Errors

Exception type	Error number	Condition
InvalidCastException	13	*DateValue* is not coercible to **Date.**
ArgumentException	5	*Interval* is invalid.
ArgumentOutOfRangeException	9	Calculated date is before 00:00:00 on January 1 of the year 1, or later than 23:59:59 on December 31, 9999.

Remarks

You can use the **DateAdd** function to add or subtract a specified time interval from a date. For example, you can calculate a date

30 days from today or a time 45 minutes before now.

To add days to *DateValue*, you can use **DateInterval.Day**, **DateInterval.DayOfYear**, or **DateInterval.Weekday**. These are treated as equivalent because **DayOfYear** and **Weekday** are not meaningful time intervals.

The **DateAdd** function never returns an invalid date. If necessary, the day part of the resulting date is adjusted downward to the last day of the resulting month in the resulting year. The following example adds one month to January 31:

```
Dim NextMonth As Date = DateAdd(DateInterval.Month, 1, #1/31/1995#)
```

In this example, **DateAdd** returns `#2/28/1995#`, not `#2/31/1995#`. If *DateValue* is `#1/31/1996#`, it returns `#2/29/1996#` because 1996 is a leap year.

If any argument has an invalid value, an **ArgumentException** error occurs. If the calculated date is earlier than 00:00:00 on January 1 of the year 1, or later than 23:59:59 on December 31, 9999, an **ArgumentOutOfRangeException** error occurs. If the *DateValue* argument has a value that cannot be coerced to a valid **Date** value, an **InvalidCastException** error occurs.

> **Note** **DateAdd** uses the current calendar setting from the **CurrentCulture** property of the **CultureInfo** class in the **System.Globalization** namespace. The default **CurrentCulture** values are determined by **Control Panel** settings.

Since every **Date** value is supported by a **DateTime** structure, its methods give you additional options in adding time intervals. For example, you can add a fractional number of days, rounded to the nearest millisecond, to a **Date** variable as follows:

```
Dim NextTime As Date = Now        ' Current date and time.
NextTime = NextTime.AddDays(3.4)  ' Increment by 3 2/5 days.
```

Example

This example takes a date and, using the **DateAdd** function, displays a corresponding date a specified number of months in the future.

```
Dim Msg, Number, StartDate As String   'Declare variables.
Dim Months As Double
Dim SecondDate As Date
Dim IntervalType As DateInterval
IntervalType = DateInterval.Month   ' Specifies months as interval.
StartDate = InputBox("Enter a date")
SecondDate = CDate(StartDate)
Number = InputBox("Enter number of months to add")
Months = Val(Number)
Msg = "New date: " & DateAdd(IntervalType, Months, SecondDate)
MsgBox (Msg)
```

Requirements

Namespace: Microsoft.VisualBasic

Module: DateAndTime

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

DateDiff Function | DatePart Function | Day Function | Format Function | Now Property | Weekday Function | Year Function | Date Data Type | System Namespace | System.Globalization Namespace | DateTime Structure | ArgumentException Class | ArgumentOutOfRangeException Class | InvalidCastException Class | CultureInfo Class

DateDiff Function

Returns a **Long** value specifying the number of time intervals between two **Date** values.

```
Public Overloads Function DateDiff( _
   ByVal Interval As DateInterval, _
   ByVal Date1 As DateTime, _
   ByVal Date2 As DateTime, _
   Optional ByVal DayOfWeek As FirstDayOfWeek = FirstDayOfWeek.Sunday, _
   Optional ByVal  WeekOfYear As FirstWeekOfYear = FirstWeekOfYear.Jan1 _
) As Long
```

-or-

```
Public Overloads Function DateDiff( _
   ByVal Interval As String, _
   ByVal Date1 As Object, _
   ByVal Date2 As Object, _
   Optional ByVal DayOfWeek As FirstDayOfWeek = FirstDayOfWeek.Sunday, _
   Optional ByVal WeekOfYear As FirstWeekOfYear = FirstWeekOfYear.Jan1 _
) As Long
```

Parameters

Interval
Required. **DateInterval** enumeration value or **String** expression representing the time interval you want to use as the unit of difference between *Date1* and *Date2*.

Date1, *Date2*
Required. **Date**. The two date/time values you want to use in the calculation. The value of *Date1* is subtracted from the value of *Date2* to produce the difference. Neither value is changed in the calling program.

DayOfWeek
Optional. A value chosen from the **FirstDayOfWeek** enumeration that specifies the first day of the week. If not specified, **FirstDayOfWeek.Sunday** is used.

WeekOfYear
Optional. A value chosen from the **FirstWeekOfYear** enumeration that specifies the first week of the year. If not specified, **FirstWeekOfYear.Jan1** is used.

Settings

The *Interval* argument can have one of the following settings.

Enumeration value	String	Unit of time difference
DateInterval.Day	d	Day
DateInterval.DayOfYear	y	Day
DateInterval.Hour	h	Hour
DateInterval.Minute	n	Minute
DateInterval.Month	m	Month
DateInterval.Quarter	q	Quarter
DateInterval.Second	s	Second
DateInterval.Weekday	w	Week
DateInterval.WeekOfYear	ww	Calendar week
DateInterval.Year	yyyy	Year

The *DayOfWeek* argument can have one of the following settings.

Enumeration value	Value	Description
FirstDayOfWeek.System	0	First day of week specified in system settings
FirstDayOfWeek.Sunday	1	Sunday (default)

FirstDayOfWeek.Monday	2	Monday (complies with ISO standard 8601, section 3.17)
FirstDayOfWeek.Tuesday	3	Tuesday
FirstDayOfWeek.Wednesday	4	Wednesday
FirstDayOfWeek.Thursday	5	Thursday
FirstDayOfWeek.Friday	6	Friday
FirstDayOfWeek.Saturday	7	Saturday

The *WeekOfYear* argument can have one of the following settings.

Enumeration value	Value	Description
FirstWeekOfYear.System	0	First week of year specified in system settings
FirstWeekOfYear.Jan1	1	Week in which January 1 occurs (default)
FirstWeekOfYear.FirstFourDays	2	Week that has at least four days in the new year (complies with ISO standard 8601, section 3.17)
FirstWeekOfYear.FirstFullWeek	3	First full week in the new year

Exceptions/Errors

Exception type	Error number	Condition
ArgumentException	5	Invalid *Interval*.
ArgumentException	5	*Date* or *DayofWeek* are out of range .
InvalidCastException	13	*Date1* or *Date2* are invalid types.

Remarks

You can use the **DateDiff** function to determine how many specified time intervals exist between two date/time values. For example, you might use **DateDiff** to calculate the number of days between two dates, or the number of weeks between today and the end of the year.

If *Interval* is set to **DateInterval.DayOfYear**, it is treated the same as **DateInterval.Day**, because **DayOfYear** is not a meaningful unit for a time interval.

If *Interval* is set to **DateInterval.WeekOfYear**, the return value represents the number of weeks between the first day of the week containing *Date1* and the first day of the week containing *Date2*. The following example shows how this produces different results from **DateInterval.Weekday**.

```
Dim DatTim1 As Date = #1/4/2001#   ' This is a Thursday.
Dim DatTim2 As Date = #1/9/2001#   ' This is the next Tuesday.
' Assume Sunday is specified as first day of the week.
Dim WD As Long = DateDiff(DateInterval.Weekday, DatTim1, DatTim2)
Dim WY As Long = DateDiff(DateInterval.WeekOfYear, DatTim1, DatTim2)
```

In the preceding example, **DateDiff** returns **0** to `WD` because the difference between the two dates is less than seven days, but it returns **1** to `WY` because there is a seven-day difference between the first days of the respective calendar weeks.

If *Interval* is set to **DateInterval.Year**, the return value is calculated purely from the year parts of *Date1* and *Date2*.

Because *Date1* and *Date2* are of the **Date** data type, they hold date and time values accurate to 100-nanosecond ticks on the system timer. However, **DateDiff** always returns the number of time intervals as a **Long** value.

If *Date1* represents a later date and time than *Date2*, **DateDiff** returns a negative number.

If any argument has an invalid value, an **ArgumentException** error occurs. If either the *Date1* or *Date2* argument has a value that cannot be coerced to a valid **Date** value, an **InvalidCastException** error occurs.

> **Note** When comparing December 31 to January 1 of the following year, **DateDiff** returns 1 for **DateInterval.Year**, **DateInterval.Quarter**, or **DateInterval.Month**, even though only one day has elapsed.

Since every **Date** value is supported by a **DateTime** structure, its methods give you additional options in finding time intervals. For example, you can use the **Subtract** method in either of its overloaded forms. One of these subtracts a **TimeSpan** from a **Date** variable to return another **Date** value; the other subtracts a **Date** value to return a **TimeSpan**. You can time a process to find out

how many milliseconds it takes, as follows:

```
Dim StartTime As Date = Now   ' Starting date/time.
' Run the process that is to be timed.
Dim RunLength As System.TimeSpan = Now.Subtract(StartTime)
Dim Millisecs As Integer = RunLength.Milliseconds
```

Example

This example uses the **DateDiff** function to display the number of days between a given date and today.

```
Dim FirstDate, Msg As String   ' Declare variables.
Dim SecondDate As Date
FirstDate = InputBox("Enter a date")
SecondDate = CDate(FirstDate)
Msg = "Days from today: " & DateDiff(DateInterval.Day, Now, SecondDate)
MsgBox (Msg)
```

Requirements

Namespace: Microsoft.VisualBasic

Module: DateAndTime

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

DateAdd Function | DatePart Function | Day Function | Format Function | Now Property | Weekday Function | Year Function | Date Data Type | System Namespace | DateTime Structure | TimeSpan Structure | ArgumentException Class | InvalidCastException Class

DatePart Function

Returns an **Integer** value containing the specified component of a given **Date** value.

```
Public Overloads Function DatePart( _
   ByVal Interval As DateInterval, _
   ByVal DateValue As DateTime, _
   Optional ByVal FirstDayOfWeekValue As FirstDayOfWeek = VbSunday, _
   Optional ByVal FirstWeekOfYearValue As FirstWeekOfYear = VbFirstJan1 _
) As Integer
```

-or-

```
Public Overloads Function DatePart( _
    ByVal Interval As String, _
   ByVal DateValue As Object, _
   Optional ByVal DayOfWeek As FirstDayOfWeek = FirstDayOfWeek.Sunday, _
   Optional ByVal WeekOfYear As FirstWeekOfYear = FirstWeekOfYear.Jan1 _
) As Integer
```

Parameters

Interval
Required. **DateInterval** enumeration value or **String** expression representing the part of the date/time value you want to return.

DateValue
Required. **Date** value that you want to evaluate.

FirstDayOfWeekValue
Optional. A value chosen from the **FirstDayOfWeek** enumeration that specifies the first day of the week. If not specified, **FirstDayOfWeek.Sunday** is used.

FirstWeekOfYearValue
Optional. A value chosen from the **FirstWeekOfYear** enumeration that specifies the first week of the year. If not specified, **FirstWeekOfYear.Jan1** is used.

Settings

The *Interval* argument can have one of the following settings.

Enumeration value	String	Part of date/time value to return
DateInterval.Day	d	Day of month (1 through 31)
DateInterval.DayOfYear	y	Day of year (1 through 366)
DateInterval.Hour	h	Hour
DateInterval.Minute	n	Minute
DateInterval.Month	m	Month
DateInterval.Quarter	q	Quarter
DateInterval.Second	s	Second
DateInterval.Weekday	w	Day of week (1 through 7)
DateInterval.WeekOfYear	ww	Week of year (1 through 53)
DateInterval.Year	yyyy	Year

The *FirstDayOfWeekValue* argument can have one of the following settings.

Enumeration value	Value	Description
FirstDayOfWeek.System	0	First day of week specified in system settings
FirstDayOfWeek.Sunday	1	Sunday (default)
FirstDayOfWeek.Monday	2	Monday (complies with ISO standard 8601, section 3.17)
FirstDayOfWeek.Tuesday	3	Tuesday
FirstDayOfWeek.Wednesday	4	Wednesday

FirstDayOfWeek.Thursday	5	Thursday
FirstDayOfWeek.Friday	6	Friday
FirstDayOfWeek.Saturday	7	Saturday

The *FirstWeekOfYearValue* argument can have one of the following settings.

Enumeration value	Value	Description
FirstWeekOfYear.System	0	First week of year specified in system settings
FirstWeekOfYear.Jan1	1	Week in which January 1 occurs (default)
FirstWeekOfYear.FirstFourDays	2	Week that has at least four days in the new year (complies with ISO standard 8601, section 3.17)
FirstWeekOfYear.FirstFullWeek	3	First full week in new year

Exceptions/Errors

Exception type	Error number	Condition
ArgumentException	5	*Interval* is invalid.
InvalidCastException	13	*DateValue* is not coercible to **Date**.

Remarks

You can use the **DatePart** function to evaluate a date/time value and return a specific component. For example, you might use **DatePart** to calculate the day of the week or the current hour.

If you choose **DateInterval.Weekday** for the *Interval* argument, the returned value is consistent with the values of the **FirstDayOfWeek** enumeration. If you choose **DateInterval.WeekOfYear**, **DatePart** uses the **Calendar** and **CultureInfo** classes of the **System.Globalization** namespace to determine your current settings.

The *FirstDayOfWeekValue* argument affects calculations that use the **DateInterval.Weekday** and **DateInterval.WeekOfYear** *Interval* settings. The *FirstWeekOfYearValue* argument affects calculations that specify **DateInterval.WeekOfYear** for *Interval*.

If any argument has an invalid value, an **ArgumentException** error occurs. If the *DateValue* argument has a value that cannot be coerced to a valid **Date** value, an **InvalidCastException** error occurs.

Since every **Date** value is supported by a **DateTime** structure, its methods give you additional options in retrieving date/time parts. For example, you can obtain the entire date value of a **Date** variable, with the time value set to midnight, as follows:

```
Dim CurrDatTim As Date = Now   ' Current date and time.
Dim LastMidnight As Date = CurrDatTim.Date   ' At midnight.
```

Example

This example takes a date and, using the **DatePart** function, displays the quarter of the year in which it occurs.

```
Dim FirstDate, Msg As String   'Declare variables.
Dim SecondDate As Date
FirstDate = InputBox("Enter a date:")
SecondDate = CDate(FirstDate)
Msg = "Quarter: " & DatePart(DateInterval.Quarter, SecondDate)
MsgBox (Msg)
```

Requirements

Namespace: Microsoft.VisualBasic

Module: DateAndTime

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

DateAdd Function | DateDiff Function | Day Function | Format Function | Now Property | Weekday Function | Year Function | Date Data Type | System Namespace | DateTime Structure | Calendar Class | CultureInfo Class | ArgumentException Class | InvalidCastException Class

DateSerial Function

Returns a **Date** value representing a specified year, month, and day, with the time information set to midnight (00:00:00).

```
Public Function DateSerial( _
   ByVal [Year] As Integer, _
   ByVal [Month] As Integer, _
   ByVal [Day] As Integer _
) As DateTime
```

Parameters

Year
Required. **Integer** expression from 1 through 9999. However, values below this range are also accepted. If *Year* is 0 through 99, it is interpreted as being between 1930 and 2029, as explained in the Remarks section below. If *Year* is less than 1, it is subtracted from the current year.

Month
Required. **Integer** expression from 1 through 12. However, values outside this range are also accepted. The value of *Month* is offset by 1 and applied to January of the calculated year. In other words, (*Month* - 1) is added to January. The year is recalculated if necessary. The following results illustrate this effect:

- If *Month* is 1, the result is January of the calculated year.
- If *Month* is 0, the result is December of the previous year.
- If *Month* is -1, the result is November of the previous year.
- If *Month* is 13, the result is January of the following year.

Day
Required. **Integer** expression from 1 through 31. However, values outside this range are also accepted. The value of *Day* is offset by 1 and applied to the first day of the calculated month. In other words, (*Day* - 1) is added to the first of the month. The month and year are recalculated if necessary. The following results illustrate this effect:

- If *Day* is 1, the result is the first day of the calculated month.
- If *Day* is 0, the result is the last day of the previous month.
- If *Day* is -1, the result is the penultimate day of the previous month.
- If *Day* is past the end of the current month, the result is the appropriate day of the following month. For example, if *Month* is 4 and *Day* is 31, the result is May 1.

Remarks

Under Windows® 98 or Windows 2000, two-digit years for the *Year* argument are interpreted based on user-defined machine settings. The default settings are that values from 0 through 29 are interpreted as the years 2000–2029, and values from 30 through 99 are interpreted as the years 1930–1999. For all other *Year* arguments, use a four-digit year; for example, 1924.

Earlier versions of Windows interpret two-digit years based on the defaults described previously. To be sure the function returns the proper value, use a four-digit *Year*.

The following example demonstrates negative, zero, and positive argument values. Here, the **DateSerial** function returns a **Date** representing the day before the first day of March in the year 10 years before the current year; in other words, the last day of February ten years ago.

```
Dim EndFeb As Date = DateSerial(-10, 3, 0)
```

If either *Month* or *Day* exceeds its normal range, it is applied to the next larger unit as appropriate. For example, if you specify 32 days, it is evaluated as one month and from one to four days, depending on the value of *Month*. If *Year* is greater than 9999, or if any argument is outside the range -2,147,483,648 through 2,147,483,647, an **ArgumentException** error occurs. If the date specified by the three arguments is earlier than 00:00:00 on January 1 of the year 1, or later than 23:59:59 on December 31, 9999, an **ArgumentOutOfRangeException** error occurs.

The **Date** data type includes time components. **DateSerial** sets all of these to 0, so the returned value represents the beginning of the calculated day.

Since every **Date** value is supported by a **DateTime** structure, its methods give you additional options in assembling a **Date** value. For example, you can use one of the overloaded **DateTime** constructors to populate a **Date** variable using the desired combination of components. The following example sets `NewDateTime` to May 6, 1978 at one tenth of a second before 8:30 in the morning:

```
Dim NewDateTime As Date = New Date(1978, 5, 6, 8, 29, 59, 900)
```

Example

This example uses the **DateSerial** function to return the date for the specified year, month, and day.

```
Dim MyDate As Date
' MyDate contains the date for February 12, 1969.
MyDate = DateSerial(1969, 2, 12)   ' Return a date.
```

Requirements

Namespace: Microsoft.VisualBasic

Module: DateAndTime

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

DateValue Function | Day Function | Month Function | Now Property | TimeSerial Function | TimeValue Function | Weekday Function | Year Function | Date Data Type | System Namespace | DateTime Structure | ArgumentException Class | ArgumentOutOfRangeException Class

DateValue Function

Returns a **Date** value containing the date information represented by a string, with the time information set to midnight (00:00:00).

```
Public Function DateValue(ByVal StringDate As String) As DateTime
```

Parameter

StringDate
Required. **String** expression representing a date/time value from 00:00:00 on January 1 of the year 1 through 23:59:59 on December 31, 9999.

Exceptions/Errors

Exception type	Error number	Condition
InvalidCastException	13	*StringDate* includes invalid time informatio n.

Remarks

If *StringDate* includes only numbers from 1 through 12 separated by valid date separators, **DateValue** recognizes the order for month, day, and year according to the Short Date format specified for your system. **DateValue** uses the current calendar setting from the **CurrentCulture** property of the **CultureInfo** class in the **System.Globalization** namespace. The default **CurrentCulture** values are determined by **Control Panel** settings. You can override the Short Date format by setting the **ShortDatePattern** property of the **DateTimeFormatInfo** class in the **System.Globalization** namespace.

DateValue recognizes month names in long, abbreviated, and numeric form. For example, in addition to recognizing 12/30/1991 and 12/30/91, **DateValue** also recognizes December 30, 1991 and Dec 30, 1991.

If the year part of *StringDate* is omitted, **DateValue** uses the current year from your computer's system date.

If the *StringDate* argument includes time information, **DateValue** does not include it in the returned value. However, if *StringDate* includes invalid time information, such as "89:98", an **InvalidCastException** error occurs.

Example

This example uses the **DateValue** function to convert a string to a date. You can also use date literals to directly assign a date to an **Object** or **Date** variable, for example, `MyDate = #2/12/69#`.

```
Dim MyDate As Date
MyDate = DateValue("February 12, 1969")   ' Returns a date.
```

Requirements

Namespace: Microsoft.VisualBasic

Module: DateAndTime

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

DateSerial Function | Day Function | Month Function | Now Property | TimeSerial Function | TimeValue Function | Weekday Function | Year Function | Date Data Type | System Namespace | System.Globalization Namespace | DateTime Structure | InvalidCastException Class | CultureInfo Class | DateTimeFormatInfo Class

Day Function

Returns an **Integer** value from 1 through 31 representing the day of the month.

```
Public Function Day(ByVal DateValue As DateTime) As Integer
```

Parameter

DateValue
Required. **Date** value from which you want to extract the day.

Remarks

If you use the **Day** function, you might have to qualify it with the **Microsoft.VisualBasic** namespace, because the **System.Windows.Forms** namespace defines **Day** as an enumeration. The following example shows how qualifying **Day** resolves this ambiguity:

```
Dim ThisDay As Integer = Microsoft.VisualBasic.DateAndTime.Day(Now)
```

You can also obtain the day of the month by calling **DatePart** and specifying **DateInterval.Day** for the *Interval* argument.

Example

This example uses the **Day** function to obtain the day of the month from a specified date. In the development environment, the date literal is displayed in standard short format (such as "02/12/1969") using the locale settings of your code.

```
Dim MyDate As Date
Dim MyDay As Integer
MyDate = #2/12/1969#   ' Assign a date using standard short format.
MyDay = Microsoft.VisualBasic.DateAndTime.Day(MyDate)   ' MyDay contains 12.
```

Day is qualified to distinguish it from the **System.Windows.Forms.Day** enumeration.

Requirements

Namespace: Microsoft.VisualBasic

Module: DateAndTime

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Hour Function | Minute Function | Month Function | Now Property | Second Function | Weekday Function | Year Function | DatePart Function | System Namespace | System.Windows.Forms Namespace | DateTime Structure | ArgumentException Class | ArgumentOutOfRangeException Class

DDB Function

Returns a **Double** specifying the depreciation of an asset for a specific time period using the double-declining balance method or some other method you specify.

```
Function DDB( _
   ByVal Cost As Double, _
   ByVal Salvage As Double, _
   ByVal Life As Double, _
   ByVal Period As Double, _
   Optional ByVal Factor As Double = 2.0 _
) As Double
```

Parameters

Cost
Required. **Double** specifying initial cost of the asset.
Salvage
Required. **Double** specifying value of the asset at the end of its useful life.
Life
Required. **Double** specifying length of useful life of the asset.
Period
Required. **Double** specifying period for which asset depreciation is calculated.
Factor
Optional. **Double** specifying rate at which the balance declines. If omitted, 2 (double-declining method) is assumed.

Exceptions/Errors

Exception type	Error number	Condition
ArgumentException	5	Argument value < 0 or *Period* > *Life* .

Remarks

The double-declining balance method computes depreciation at an accelerated rate. Depreciation is highest in the first period and decreases in successive periods.

The *Life* and *Period* arguments must be expressed in the same units. For example, if *Life* is given in months, *Period* must also be given in months. All arguments must be positive numbers.

The **DDB** function uses the following formula to calculate depreciation for a given period:

```
Depreciation / Period = ((Cost - Salvage) * Factor) / Life
```

Example

This example uses the **DDB** function to return the depreciation of an asset for a specified period given the initial cost (`InitCost`), the salvage value at the end of the asset's useful life (`SalvageVal`), the total life of the asset in years (`LifeTime`), and the period in years for which the depreciation is calculated (`Depr`).

```
Sub testDDB()
   Dim InitCost, SalvageVal, MonthLife, DepYear As Double
   Dim Fmt As String
   Dim LifeTime, Depr As Double
   Const YRMOS As Integer = 12    ' Number of months in a year.
   Fmt = "###,##0.00"
   InitCost = CDbl(InputBox("What's the initial cost of the asset?"))
   SalvageVal = CDbl(InputBox("Enter the asset's value at end of its life."))
   MonthLife = CDbl(InputBox("What's the asset's useful life in months?"))
   Do While MonthLife < YRMOS   ' Ensure period is >= 1 year.
      MsgBox("Asset life must be a year or more.")
```

```
        MonthLife = CDbl(InputBox("What's the asset's useful life in months?"))
    Loop
    LifeTime = MonthLife / YRMOS   ' Convert months to years.
    If LifeTime <> Int(MonthLife / YRMOS) Then
        LifeTime = Int(LifeTime + 1)   ' Round up to nearest year.
    End If
    DepYear = CInt(InputBox("Enter year for depreciation calculation."))
    Do While DepYear < 1 Or DepYear > LifeTime
        MsgBox("You must enter at least 1 but not more than " & LifeTime)
        DepYear = CDbl(InputBox("Enter year for depreciation calculation."))
    Loop
    Depr = DDB(InitCost, SalvageVal, LifeTime, DepYear)
    MsgBox("The depreciation for year " & DepYear & " is " & _
    Format(Depr, Fmt) & ".")
End Sub
```

Requirements

Namespace: Microsoft.VisualBasic

Module: Financial

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

FV Function | IPmt Function | IRR Function | MIRR Function | NPer Function | NPV Function | Pmt Function | PPmt Function | PV Function | Rate Function | SLN Function | SYD Function | ArgumentException Class

DeleteSetting Function

Deletes a section or key setting from an application's entry in the Windows registry.

```
Public Sub DeleteSetting( _
   ByVal AppName As String, _
   Optional ByVal Section As String = Nothing, _
   Optional ByVal Key As String = Nothing _
)
```

Parameters

AppName
Required. **String** expression containing the name of the application or project to which the section or key setting applies.

Section
Required. **String** expression containing the name of the section from which the key setting is being deleted. If only *AppName* and *Section* are provided, the specified section is deleted along with all related key settings.

Key
Optional. **String** expression containing the name of the key setting being deleted.

Exceptions/Errors

Exception type	Error number	Condition
ArgumentException	5	*Section*, *AppName*, or *Key* setting does not exist.
ArgumentException	5	User is not logged in.

Remarks

If all arguments are provided, the specified setting is deleted. A run-time error occurs if you attempt to use **DeleteSetting** on a nonexistent section or key setting.

DeleteSetting requires that a user be logged on since it operates under the HKEY_LOCAL_USER registry key, which is not active until a user logs on interactively.

Registry settings that are to be accessed from a non-interactive process (such as mtx.exe) should be stored under either the HKEY_LOCAL_MACHINE\Software\ or the HKEY_USER\DEFAULT\Software registry keys.

Example

The following example first uses the **SaveSetting** procedure to make entries in the Windows registry for the `MyApp` application, and then uses the **DeleteSetting** function to remove them. Because no *Key* argument is specified, the whole `Startup` section is deleted, including the section name and all of its keys.

```
' Place some settings in the registry.
SaveSetting("MyApp", "Startup", "Top", "75")
SaveSetting("MyApp","Startup", "Left", "50")
' Remove section and all its settings from registry.
DeleteSetting ("MyApp", "Startup")
' Remove MyApp from the registry.
DeleteSetting ("MyApp")
```

Smart Device Developer Notes

This function is not supported.

Requirements

Namespace: Microsoft.VisualBasic

Module: Interaction

Assembly: Microsoft.Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

GetAllSettings Function | GetSetting Function | SaveSetting Function | ArgumentException Class

Dir Function

Returns a string representing the name of a file, directory, or folder that matches a specified pattern or file attribute, or the volume label of a drive.

```
Public Overloads Function Dir() As String
```

-or-

```
Public Overloads Function Dir( _
   Optional ByVal PathName As String, _
   Optional ByVal Attributes As FileAttribute = FileAttribute.Normal _
) As String
```

Parameters

PathName
Optional. **String** expression that specifies a file name, directory or folder name, or drive volume label. A zero-length string ("") is returned if *PathName* is not found.

Attributes
Optional. Enumeration or numeric expression whose value specifies file attributes. If omitted, returns files that match *PathName,* but have no attributes.

Settings

The *Attributes* argument enumeration values are as follows:

Value	Constant	Description
Normal	**vbnormal**	Default. Specifies files with no attributes.
ReadOnly	**vbReadOnly**	Specifies read-only files in addition to files with no attributes.
Hidden	**vbHidden**	Specifies hidden files in addition to files with no attributes.
System	**vbSystem**	Specifies system files in addition to files with no attributes.
Volume	**vbVolume**	Specifies volume label; if any other attribute is specified, **vbVolume** is ignored.
Directory	**vbDirectory**	Specifies directories or folders in addition to files with no attributes.
Archive	**vbArchive**	File has changed since last backup.
Alias	**vbAlias**	File has a different name.

> **Note** These enumerations are specified by the Visual Basic language and can be used anywhere in your code in place of the actual values.

Remarks

The **Dir** function supports the use of multiple-character (*****) and single-character (?) wildcards to specify multiple files.

You must supply a **PathName** the first time you call the **Dir** function. Subsequent calls to the **Dir** function may be made with no parameters to retrieve the next item.

> **Security Note** Do not make decisions about the contents of a file based on the file name extension. For example, a file named Form1.vb may not be a Visual Basic .NET source file.

Example

This example uses the **Dir** function to check if certain files and directories exist.

```
Dim MyFile, MyPath, MyName
' Returns "WIN.INI" if it exists.
MyFile = Dir("C:\WINDOWS\WIN.INI")
```

```
' Returns filename with specified extension. If more than one *.INI
' file exists, the first file found is returned.
MyFile = Dir("C:\WINDOWS\*.INI")

' Call Dir again without arguments to return the next *.INI file in the
' same directory.
MyFile = Dir()

' Return first *.TXT file, including files with a set hidden attribute.
MyFile = Dir("*.TXT", vbHidden)

' Display the names in C:\ that represent directories.
MyPath = "c:\"   ' Set the path.
MyName = Dir(MyPath, vbDirectory)   ' Retrieve the first entry.
Do While MyName <> ""   ' Start the loop.
      ' Use bitwise comparison to make sure MyName is a directory.
      If (GetAttr(MyPath & MyName) And vbDirectory) = vbDirectory Then
         ' Display entry only if it's a directory.
         Debug.WriteLine(MyName)
      End If
   MyName = Dir()   ' Get next entry.
Loop
```

Smart Device Developer Notes

This function is not supported.

Requirements

Namespace: Microsoft.VisualBasic

Module: FileSystem

Assembly: Microsoft Visual Basic .NET Runtime (in MicrosoftVisualBasic.dll)

See Also

ChDir Function | CurDir Function | FileAttribute Enumeration

Environ Function

Returns the string associated with an operating system environment variable.

```
Overloads Function Environ(ByVal Expression As Integer) As String
```

-or-

```
Overloads Function Environ(ByVal Expression As String) As String
```

Parameter

Expression
Required. Expression that evaluates to either a string containing the name of an environment variable, or an integer corresponding to the numeric order of an environment string in the environment-string table.

Exceptions/Errors

Exception type	Error number	Condition
ArgumentException	5	*Expression* is missing.

Remarks

If *Expression* contains a string, the **Environ** function returns the text assigned to the specified environment string; that is, the text following the equal sign (=) in the environment-string table for that environment variable. If the string in *Expression* cannot be found in the environment-string table, a zero-length string ("") is returned.

If *Expression* contains an integer, the string occupying that numeric position in the environment-string table is returned. In this case, **Environ** returns all of the text, including the name of the environment variable. If there is no environment string in the specified position, **Environ** returns a zero-length string.

Example

This example uses the **Environ** function to supply the entry number and length of the `PATH` statement from the environment-string table.

```
Sub tenv()
   Dim envString As String
   Dim found As Boolean = False
   Dim index As Integer = 1
   Dim pathLength As Integer
   Dim message As String

   envString = Environ(index)
   While Not found And (envString <> "")
      If (envString.Substring(0, 5) = "Path=") Then
         found = True
      Else
         index += 1
         envString = Environ(index)
      End If
   End While

   If found Then
      pathLength = Environ("PATH").Length
      message = "PATH entry = " & index & " and length = " & pathLength
   Else
      message = "No PATH environment variable exists."
   End If
```

```
    MsgBox(message)
End Sub
```

Smart Device Developer Notes

This function is not supported.

Requirements

Namespace: Microsoft.VisualBasic

Module: Interaction

Assembly: Microsoft Visual Basic .NET Runtime (in MicrosoftVisualBasic.dll)

See Also

Visual Basic Run-Time Library Members | ArgumentException Class

EOF Function

Returns a Boolean value **True** when the end of a file opened for **Random** or sequential **Input** has been reached.

```
Public Function EOF(ByVal FileNumber As Integer) As Boolean
```

Parameter

FileNumber
Required. An **Integer** containing any valid file number.

Exceptions/Errors

Exception type	Error number	Condition
IOException	52	*FileNumber* does not exist.
IOException	54	File mode is invalid.

Remarks

Use **EOF** to avoid the error generated by attempting to get input past the end of a file.

The **EOF** function returns **False** until the end of the file has been reached. With files opened for **Random** or **Binary** access, **EOF** returns **False** until the last executed **FileGet** function is unable to read an entire record.

With files opened for **Binary** access, an attempt to read through the file using the **Input** function until **EOF** returns **True** generates an error. Use the **LOF** and **Loc** functions instead of **EOF** when reading binary files with **Input**, or use **Get** when using the **EOF** function. With files opened for **Output**, **EOF** always returns **True**.

Example

This example uses the **EOF** function to detect the end of a file. This example assumes that `TESTFILE` is a text file with a few lines of text.

```
Dim TextLine As String
FileOpen(1, "TESTFILE", OpenMode.Input)   ' Open file.
Do While Not EOF(1)   ' Loop until end of file.
   TextLine = LineInput(1)   ' Read line into variable.
   Debug.WriteLine(TextLine)   ' Print to the Command window.
Loop
FileClose(1)   ' Close file.
```

Smart Device Developer Notes

This function is not supported.

Requirements

Namespace: Microsoft.VisualBasic

Module: FileSystem

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

FileGet Function | Loc Function | LOF Function | FileOpen Function | IOException Class

ErrorToString Function

Returns the error message that corresponds to a given error number.

```
Public Shared Function ErrorToString(ByVal errornumber As Integer) As String
```

Parameter

errornumber
Optional. Any valid error number.

Exceptions/Errors

Exception type	Error number	Condition
ArgumentException	5	*ErrorNumber* is out of range.

Remarks

The **ErrorToString** function examines the property settings of the **Err** object to identify the most recent run-time error. The return value of the **ErrorToString** function corresponds to the **Description** property of the **Err** object. If *errornumber* is a valid error number but is not defined, **ErrorToString** returns the string, "Application-defined or object-defined error." If *errornumber* is not valid, an error occurs. If *errornumber* is omitted, the message corresponding to the most recent run-time error is returned. If no run-time error has occurred, or *errornumber* is 0, **ErrorToString** returns a zero-length string ("").

In Visual Basic version 6.0 and earlier, this functionality was provided by the **Error** function.

Example

The following code uses the **ErrorToString** function to display error messages that correspond to the specified error numbers:

```
Dim ErrorNumber As Integer
For ErrorNumber = 61 To 64   ' Loop through values 61 - 64.
   MsgBox(ErrorToString(ErrorNumber))   ' Display error names in message box.
Next ErrorNumber
```

Requirements

Namespace: Microsoft.VisualBasic

Module: Conversion

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Err Object | Description Property | Structured Exception Handling | Unstructured Exception Handling | ArgumentException Class

Math Functions

The math functions in Visual Basic 6 have been replaced by equivalent methods in the **System.Math** class of the .NET Framework.

Remarks

The .NET Framework math methods are functionally identical to their Visual Basic 6 counterparts, although some have slightly different names. For example, the .NET Framework equivalent of the Visual Basic 6 Atn function is **Atan**. The following table lists the Visual Basic 6 math function names and the equivalent .NET Framework methods.

Visual Basic 6 function	Visual Basic .NET method	Description
Abs	Math.Abs Method	Returns the absolute value of a specified number.
Atn	Math.Atan Method	Returns a **Double** value containing the angle whose tangent is the specified number.
Cos	Math.Cos Method	Returns a **Double** value containing the cosine of the specified angle.
Exp	Math.Exp Method	Returns a **Double** value containing e (the base of natural logarithms) raised to the specified power.
Log	Math.Log Method	Returns a **Double** value containing the logarithm of a specified number. This method is overloaded and can return either the natural (base e) logarithm of a specified number or the logarithm of a specified number in a specified base.
Round	Math.Round Method	Returns a **Double** value containing the number nearest the specified value. Additional round functions are available as methods of the intrinsic types such as Decimal.Round Method.
Sgn	Math.Sign Method	Returns an **Integer** value indicating the sign of a number.
Sin	Math.Sin Method	Returns a **Double** value specifying the sine of an angle.
Sqr	Math.Sqrt Method	Returns a **Double** value specifying the square root of a number.
Tan	Math.Tan Method	Returns a **Double** value containing the tangent of an angle.

In addition, the .NET Framework math class provides constants and other static methods for trigonometric, logarithmic, and other common mathematical functions. All of these can be used in a Visual Basic program.

To use these functions without qualification, import the **System.Math** namespace into your project by adding the following code to the top of the source code:

```
Imports System.Math
```

Requirements

Class: Math Class

Abs Example

This example uses the **Abs** method of the **Math** class to compute the absolute value of a number.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim MyNumber As Double
```

```
MyNumber = Abs(50.3)    ' Returns 50.3.
MyNumber = Abs(-50.3)   ' Returns 50.3.
```

Atan Example

This example uses the **Atan** method of the **Math** class to calculate the value of pi.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim pi As Double
pi = 4 * Atan(1)   ' Calculate the value of pi.
```

Cos Example

This example uses the **Cos** method of the **Math** class to return the cosine of an angle.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim MyAngle, MySecant As Double
MyAngle = 1.3   ' Define angle in radians.
MySecant = 1 / Cos(MyAngle)   ' Calculate secant.
```

Exp Example

This example uses the **Exp** method of the **Math** class to return *e* raised to a power.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim MyAngle, MyHSin As Double
' Define angle in radians.
MyAngle = 1.3
' Calculate hyperbolic sine.
MyHSin = (Exp(MyAngle) - Exp(-1 * MyAngle)) / 2
```

Log Example

This example uses the **Log** method of the **Math** class to return the natural logarithm of a number.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim MyAngle, MyLog As Double
' Define angle in radians.
MyAngle = 1.3
' Calculate inverse hyperbolic sine.
MyLog = Log(MyAngle + Sqrt(MyAngle * MyAngle + 1))
```

Round Example

This example uses the **Round** method of the **Math** class to round a number to the nearest integer.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim MyVar1 As Double = 2.8
Dim MyVar2 As Double
MyVar2 =Round(MyVar1)   ' Returns 3.
```

Sign Example

This example uses the **Sign** method of the **Math** class to determine the sign of a number.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim MyVar1, MyVar2, MyVar3 As Double
Dim MySign As Integer
MyVar1 = 12
MyVar2 = -2.4
MyVar3 = 0
MySign = Sign(MyVar1)   ' Returns 1.
MySign = Sign(MyVar2)   ' Returns -1.
MySign = Sign(MyVar3)   ' Returns 0.
```

Sin Example

This example uses the **Sin** method of the **Math** class to return the sine of an angle.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim MyAngle, MyCosecant As Double
MyAngle = 1.3   ' Define angle in radians.
MyCosecant = 1 / Sin(MyAngle)   ' Calculate cosecant.
```

Sqrt Example

This example uses the **Sqrt** method of the **Math** class to calculate the square root of a number.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim MySqr As Double
MySqr = Sqrt(4)    ' Returns 2.
MySqr = Sqrt(23)   ' Returns 4.79583152331272.
MySqr = Sqrt(0)    ' Returns 0.
MySqr = Sqrt(-4)   ' Returns NaN (not a number).
```

Tan Example

This example uses the **Tan** method of the **Math** class to return the tangent of an angle.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim MyAngle, MyCotangent As Double
MyAngle = 1.3   ' Define angle in radians.
MyCotangent = 1 / Tan(MyAngle)   ' Calculate cotangent.
```

Requirements

Namespace: System.Math

Assembly: mscorlib (in mscorlib.dll)

See Also

Rnd Function | Randomize Statement | Derived Math Functions | Double.NaN Field

FileAttr Function

Returns an enumeration representing the file mode for files opened using the **FileOpen** function.

```
Public Function FileAttr(ByVal FileNumber As Integer) As OpenMode
```

Parameter

FileNumber
Required. **Integer**. Any valid file number.

Exceptions/Errors

Exception type	Error number	Condition
IOException	52	*FileNumber* does not exist.
IOException	54	File mode is invalid.

Return Values

The following enumeration values indicate the file access mode:

Value	Mode
1	**OpenMode.Input**
2	**OpenMode.Output**
4	**OpenMode.Random**
8	**OpenMode.Append**
32	**OpenMode.Binary**

Example

This example uses the **FileAttr** function to return the file mode of an open file.

```
Dim mode As OpenMode
FileOpen(1, "c:\MYFILE.TXT", OpenMode.Input)
mode = FileAttr(1)
MsgBox("The file mode is " & mode.ToString())
FileClose(1)
```

Smart Device Developer Notes

This function is not supported.

Requirements

Namespace: Microsoft.VisualBasic

Module: FileSystem

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

GetAttr Function | FileOpen Function | SetAttr Function | IOException Class

FileClose Function

Concludes input/output (I/O) to a file opened using the **FileOpen** function.

```
Public Sub FileClose(ParamArray FileNumbers() As Integer)
```

Parameter

FileNumbers()
Optional. Parameter array of 0 or more channels to be closed.

Exceptions/Errors

Exception type	Error number	Condition
IOException	52	*FileNumber* does not exist.

Remarks

If you omit *FileNumbers()*, all active files opened by the **FileOpen** function are closed.

When you close files that were opened for **Output** or **Append**, the final buffer of output is written to the operating system buffer for that file. All buffer space associated with the closed file is released.

When the **FileClose** function is executed, the association of a file with its file number ends.

Example

This example uses the **FileClose** function to close a file opened for **Input**.

```
Dim TextLine As String
FileOpen(1, "TESTFILE", OpenMode.Input)   ' Open file.
Do While Not EOF(1)   ' Loop until end of file.
   TextLine = LineInput(1)   ' Read line into variable.
   Debug.WriteLine(TextLine)   ' Print to the Immediate window.
Loop
FileClose(1)   ' Close file.
```

Smart Device Developer Notes

This function is not supported.

Requirements

Namespace: Microsoft.VisualBasic

Module: FileSystem

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

End Statement | FileOpen Function | Reset Function | Stop Statement | File Access with Visual Basic Run-Time Functions| IOException Class

FileCopy Function

Copies a file.

```
Public Sub FileCopy( _
   ByVal Source As String, _
   ByVal Destination As String _
)
```

Parameters

Source
Required. **String** expression that specifies the name of the file to be copied. *Source* may include the directory or folder, and drive, of the source file.

Destination
Required. **String** expression that specifies the target file name. *Destination* may include the directory or folder, and drive, of the destination file.

Exceptions/Errors

Exception type	Error number	Condition
ArgumentException	52	*Source* or *Destination* is invalid or not speci fied.
IOException	55	File is already open.
FileNotFoundException	53	File does not exist.

Remarks

If you try to use the **FileCopy** function on a currently open file, an error occurs.

Example

This example uses the **FileCopy** function to copy one file to another. For purposes of this example, assume that `SRCFILE` is a file containing some data.

```
Dim SourceFile, DestinationFile As String
SourceFile = "SRCFILE"   ' Define source file name.
DestinationFile = "DESTFILE"   ' Define target file name.
FileCopy(SourceFile, DestinationFile)   ' Copy source to target.
```

Smart Device Developer Notes

This function is not supported.

Requirements

Namespace: Microsoft.VisualBasic

Module: FileSystem

Assembly: Microsoft Visual Basic .NET Runtime (in MicrosoftVisualBasic.dll)

See Also

Kill Function | ArgumentException Class | IOException Class | FileNotFoundException Class

FileDateTime Function

Returns a **Date** value that indicates the date and time when a file was created or last modified.

```
Public Function FileDateTime(ByVal PathName As String) As DateTime
```

Parameter

PathName
Required. **String** expression that specifies a file name. *PathName* may include the directory or folder, and the drive.

Exceptions/Errors

Exception type	Error number	Condition
ArgumentException	52	*PathName* is invalid or contains wildcards .
FileNotFoundException	53	Target file does not exist.

Example

This example uses the **FileDateTime** function to determine the date and time a file was created or last modified. The format of the date and time displayed is based on the locale settings of your system.

```
Dim MyStamp As Date
' Assume TESTFILE was last modified on October 12, 2001 at 4:35:47 PM.
' Assume English/U.S. locale settings.
' Returns "10/12/2001 4:35:47 PM".
MyStamp = FileDateTime("C:\TESTFILE.txt")
```

Smart Device Developer Notes

This function is not supported.

Requirements

Namespace: Microsoft.VisualBasic

Module: FileSystem

Assembly: Microsoft Visual Basic .NET Runtime (in MicrosoftVisualBasic.dll)

See Also

FileLen Function | GetAttr Function | ArgumentException Class | FileNotFoundException Class

FileGet Function

Reads data from an open disk file into a variable.

```
Public Overloads Sub FileGet( _
   ByVal FileNumber As Integer, _
   ByRef Value As Object, _
   Optional RecordNumber As Integer = -1 _
)
```

-or-

```
Public Overloads Sub FileGet( _
   ByVal FileNumber As Integer, _
   ByRef Value As Short, _
   Optional RecordNumber As Integer = -1 _
)
```

-or-

```
Public Overloads Sub FileGet( _
   ByVal FileNumber As Integer, _
   ByRef Value As Integer, _
   Optional RecordNumber As Integer = -1 _
)
```

-or-

```
Public Overloads Sub FileGet( _
   ByVal FileNumber As Integer, _
   ByRef Value As Single, _
   Optional RecordNumber As Integer = -1 _
)
```

-or-

```
Public Overloads Sub FileGet( _
   ByVal FileNumber As Integer, _
   ByRef Value As Double, _
   Optional RecordNumber As Integer = -1 _
)
```

-or-

```
Public Overloads Sub FileGet( _
   ByVal FileNumber As Integer, _
   ByRef Value As Decimal, _
   Optional RecordNumber As Integer = -1 _
)
```

-or-

```
Public Overloads Sub FileGet( _
   ByVal FileNumber As Integer, _
   ByRef Value As Byte, _
   Optional RecordNumber As Integer = -1 _
)
```

-or-

```
Public Overloads Sub FileGet( _
   ByVal FileNumber As Integer, _
   ByRef Value As Boolean, _
   Optional RecordNumber As Integer = -1 _
)
```

-or-

```
Public Overloads Sub FileGet( _
   ByVal FileNumber As Integer, _
   ByRef Value As Date, _
   Optional RecordNumber As Integer = -1 _
)
```

-or-

```
Public Overloads Sub FileGet( _
   ByVal FileNumber As Integer, _
   ByRef Value As System.Array, _
   Optional RecordNumber As Integer = -1, _
   Optional ArrayIsDynamic as Boolean = False _
)
```

-or-

```
Public Overloads Sub FileGet(
   ByVal FileNumber As Integer, _
   ByRef Value As String, _
   Optional RecordNumber As Integer = -1, _
   Optional StringIsFixedLength as Boolean = False _
)
```

Parameters

FileNumber
Required. Any valid file number.
Value
Required. Valid variable name into which data is read.
RecordNumber
Optional. Record number (**Random** mode files) or byte number (**Binary** mode files) at which reading begins.
ArrayIsDynamic
Optional. Applies only when writing an array. Specifies whether the array is to be treated as dynamic and so whether to write an array descriptor describing the size and bounds of the array.
StringIsFixedLength
Optional. Applies only when writing a string. Specifies whether to write a two-byte descriptor for the string describing the length. The default is **False**.

Exceptions/Errors

Exception type	Error number	Condition
ArgumentException	63	*RecordNumber* < 1 and not equal to -1 .
IOException	52	*FileNumber* does not exist.
IOException	54	File mode is invalid.

Remarks

FileGet is only valid in **Random** and **Binary** mode.

Data read with **FileGet** is usually written to a file with **FilePut**.

The first record or byte in a file is at position 1, the second record or byte at position 2, and so on. If you omit *RecordNumber*, the next record or byte following the last **FileGet** or **FilePut** function (or pointed to by the last **Seek** function) is read.

> **Security Note** When reading from files, do not make decisions about the contents of a file based on the file name extension. For example, a file named Form1.vb may not be a Visual Basic .NET source file.

For files opened in **Random** mode, the following rules apply:

- If the length of the data being read is less than the length specified in the *RecordLength* clause of the **FileOpen** function, **FileGet** reads subsequent records on record-length boundaries. The space between the end of one record and the beginning of the next record is padded with the existing contents of the file buffer. Because the amount of padding data can't be determined with any certainty, it is generally a good idea to have the record length match the length of the data being read.
- If the variable being read into is a string, by default **FileGet** reads a two-byte descriptor containing the string length and then reads the data that goes into the variable. Therefore, the record length specified by the *RecordLength* clause of the **FileOpen** function must be at least two bytes greater than the actual length of the string. Visual Basic 6 and earlier versions supported fixed-length strings and when put to a file, the length descriptor would not be written. If you wish to read a string without the descriptor, you should pass **True** to the *StringIsFixedLength* parameter, and the string you read into should be the correct length.
- If the variable being read into is an array, then you have a choice as to whether to read a descriptor for the size and dimension of the array. To write the descriptor, set the *ArrayIsDynamic* parameter to **True**. When reading the array, you need to match the way the array was written. If it was written with the descriptor, then you need to read the descriptor. If the descriptor is not used, then the size and bounds of the array passed into **FileGet** will be used to determine what to read.

  The descriptor specifies the rank of the array, the size, and the lower bounds for each rank. Its length equals 2 plus 8 times the number of dimensions, that is, 2 + 8 * *NumberOfDimensions*. The record length specified by the *RecordLength* parameter in the **FileOpen** function must be greater than or equal to the sum of all the bytes required to write the array data and the array descriptor. For example, the following array declaration requires 118 bytes when the array is written to disk.

  ```
  Dim MyArray(4,9) As Integer
  ```

  The 118 bytes are distributed as follows: 18 bytes for the descriptor (2 + 8 * 2), and 100 bytes for the data (5 * 10 * 2).

- If the variable being read into is any other type of variable (not a variable-length string or an object), **FileGet** reads only the variable data. The record length specified by the *RecordLength* clause in the **FileOpen** function must be greater than or equal to the length of the data being read.
- **FileGet** reads elements of structures as if each were being read individually, except that there is no padding between elements. On disk, a dynamic array in a user-defined type (written with **FilePut**) is prefixed by a descriptor whose length equals 2 plus 8 times the number of dimensions, that is, 2 + 8 * *NumberOfDimensions*. The record length specified by the *RecordLength* clause in the **FileOpen** function must be greater than or equal to the sum of all the bytes required to read the individual elements, including any arrays and their descriptors. The **VBFixedString** attribute can be applied to string fields in the structures to indicate the size of string when written to disk.

For files opened in **Binary** mode, all of the **Random** mode rules apply, except:

- The *RecordLength* clause in the **FileOpen** function has no effect. **FileGet** reads all variables from disk contiguously; that is, with no padding between records.
- For any array other than an array in a structure, **FileGet** reads only the data. No descriptor is read.
- **FileGet** reads variable-length strings that aren't elements of structures without expecting the two-byte length descriptor. The number of bytes read equals the number of characters already in the string. For example, the following statements read 11 bytes from file number 1:

  ```
  Dim hellow As New String(" ", 11)
  FileOpen(1, "C:\TESTFILE.txt", OpenMode.Binary)
  FileGet(1, hellow)
  Console.WriteLine(hellow)
  FileClose(1)
  ```

  > **Security Note** Reading from a file with the **FileGet** function requires **Read** access from the

FileIOPermissionAccess enumeration. For more information, see FileIOPermissionAccess Enumeration.

Smart Device Developer Notes

This function is not supported.

Requirements

Namespace: Microsoft.VisualBasic

Module: FileSystem

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

FileOpen Function | FilePut Function | Seek Function | FileGetObject Function

FileGetObject Function

Reads data from an open disk file into a variable.

```
Public Sub FileGetObject( _
   ByVal FileNumber As Integer, _
   ByRef Value As Object, _
   Optional RecordNumber As Integer = -1 _
)
```

-or-

```
Overloads Public Sub FileGetObject( _
   ByVal FileNumber As Integer, _
   ByRef Value As Short, _
   Optional RecordNumber As Integer = -1 _
)
```

-or-

```
Overloads Public Sub FileGetObject( _
   ByVal FileNumber As Integer, _
   ByRef Value As Integer, _
   Optional RecordNumber As Integer = -1 _
)
```

-or-

```
Overloads Public Sub FileGetObject( _
   ByVal FileNumber As Integer, _
   ByRef Value As Single, _
   Optional RecordNumber As Integer = -1 _
)
```

-or-

```
Overloads Public Sub FileGetObject( _
   ByVal FileNumber As Integer, _
   ByRef Value As Double, _
   Optional RecordNumber As Integer = -1 _
)
```

-or-

```
Overloads Public Sub FileGetObject( _
   ByVal FileNumber As Integer, _
   ByRef Value As Decimal, _
   Optional RecordNumber As Integer = -1 _
)
```

-or-

```
Overloads Public Sub FileGetObject( _
   ByVal FileNumber As Integer, _
   ByRef Value As Byte, _
   Optional RecordNumber As Integer = -1 _
)
```

-or-

```
Overloads Public Sub FileGetObject( _
   ByVal FileNumber As Integer, _
   ByRef Value As Boolean, _
   Optional RecordNumber As Integer = -1 _
)
```

-or-

```
Overloads Public Sub FileGetObject( _
   ByVal FileNumber As Integer, _
   ByRef Value As Date, _
   Optional RecordNumber As Integer = -1 _
)
```

-or-

```
Overloads Public Sub FileGetObject( _
   ByVal FileNumber As Integer, _
   ByRef Value As System.Array, _
   Optional RecordNumber As Integer = -1, _
   Optional ArrayIsDynamic as Boolean = False _
)
```

-or-

```
Overloads Public Sub FileGetObject( _
   ByVal FileNumber As Integer, _
   ByRef Value As String, _
   Optional RecordNumber As Integer = -1, _
   Optional StringIsFixedLength as Boolean = False _
)
```

Parameters

FileNumber
Required. Any valid file number.
Value
Required. Valid variable name into which data is read.
RecordNumber
Optional. Record number (**Random** mode files) or byte number (**Binary** mode files) at which reading begins.
ArrayIsDynamic
Optional. Applies only when writing an array. Specifies whether the array is to be treated as dynamic and so whether to write an array descriptor describing the size and bounds of the array.
StringIsFixedLength
Optional. Applies only when writing a string. Specifies whether to write a two-byte descriptor for the string describing the length. The default is **False**.

Remarks

The **FileGetObject** function is used in place of **FileGet** to avoid ambiguities at compile time if type **Object** is returned rather than another type, such as **Integer**, **Long**, **Short**, and so forth.

If you intend to write out the **Variant** type, **FileGetObject** is required. When in doubt, if you are using an object for the second parameter, it is always safer to use **FilePutObject** and **FileGetObject**.

FileGetObject is only valid in **Random** and **Binary** mode.

Data read with **FileGetObject** is usually written with **FilePutObject**.

The first record or byte in a file is at position 1, the second record or byte is at position 2, and so on. If you omit *RecordNumber*, **FileGetObject** reads the record or byte after the last **FileGetObject** or **FilePutObject** function (or pointed to by the last **Seek** function).

For files opened in **Random** mode, the following rules apply:

- If the length of the data being read is less than the length specified in the *RecordLength* clause of the **FileOpen** function, **FileGetObject** reads subsequent records on record-length boundaries. The space between the end of one record and the beginning of the next record is padded with the existing contents of the file buffer. Because the amount of padding data can't be precisely determined, it is a good idea to have the record length match the length of the data being read.
- If the variable being read into is a string, by default **FileGetObject** reads a two-byte descriptor containing the string length and then reads the data that goes into the variable. Therefore, the record length specified by the *RecordLength* clause of the **FileOpen** function must be at least two bytes greater than the actual length of the string. Visual Basic 6 and earlier versions supported fixed-length strings and when read to a file, the length descriptor would not be written. If you wish to read a string without the descriptor, you should pass **True** to the *StringIsFixedLength* parameter, and the string you read into should be the correct length.
- If the variable being read into is an array, then you can choose to read a descriptor for the size and dimension of the array. To read the descriptor, set the *ArrayIsDynamic* parameter to **True**. When reading the array, you need to match the way the array was written. If it was written with the descriptor, you need to read the descriptor. If the descriptor is not used, the size and bounds of the array passed into **FileGetObject** is used to determine what to read.

  The descriptor specifies the rank of the array, the size, and the lower bounds for each rank. Its length equals 2 plus 8 times the number of dimensions, that is, 2 + 8 * *NumberOfDimensions*. The record length specified by the *RecordLength* parameter in the **FileOpen** function must be greater than or equal to the sum of all the bytes required to write the array data and the array descriptor. For example, the following array declaration requires 118 bytes when the array is written to disk:

  ```
  Dim MyArray(4,9) As Integer
  ```

  The 118 bytes are distributed as follows: 18 bytes for the descriptor (2 + 8 * 2), and 100 bytes for the data (5 * 10 * 2).

- **FileGetObject** reads elements of structures as if each were being read individually, except that there is no padding between elements. On disk, a dynamic array in a user-defined type (written with **FilePutObject**) is prefixed by a descriptor whose length equals 2 plus 8 times the number of dimensions, that is, 2 + 8 * *NumberOfDimensions*. The record length specified by the *RecordLength* clause in the **FileOpen** function must be greater than or equal to the sum of all the bytes required to read the individual elements, including any arrays and their descriptors. The **VBFixedString** attribute can be applied to string fields in the structures to indicate the size of string when written to disk.

For files opened in **Binary** mode, all of the **Random** rules apply, except:

- The *RecordLength* clause in the **FileOpen** function has no effect. **FileGetObject** reads all variables from disk contiguously; that is, with no padding between records.
- For any array other than an array in a structure, **FileGetObject** reads only the data. No descriptor is read.

FileGetObject reads variable-length strings that aren't elements of structures without expecting the two-byte length descriptor. The number of bytes read equals the number of characters already in the string.

> **Security Note** When reading from files, do not make decisions about the contents of a file based on the file name extension. For example, a file named Form1.vb may not be a Visual Basic .NET source file.

Example

The following example reads a record into a test file and then retrieves it.

```
Dim c As String
FileSystem.FileOpen(1, "test.dat", OpenMode.Binary)
FileSystem.FilePutObject(1, "ABCDEF")
FileSystem.Seek(1, 1)
FileSystem.FileGetObject(1, c)
MsgBox(c)
FileSystem.FileClose(1)
```

Smart Device Developer Notes

This function is not supported.

Requirements

Namespace: Microsoft.VisualBasic

Module: FileSystem

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

FilePut Function | FileOpen Function | Seek Function | File Access Types and Functions | FileGet Function

FileLen Function

Returns a **Long** value specifying the length of a file in bytes.

```
Public Function FileLen(ByVal PathName As String) As Long
```

Parameter

PathName
Required. **String** expression that specifies a file. *PathName* may include the directory or folder, and the drive.

Exceptions/Errors

Exception type	Error number	Condition
FileNotFoundException	53	File does not exist.

Remarks

If the specified file is open when the **FileLen** function is called, the value returned represents the size of the file at the time it was opened.

Note To obtain the current length of an open file, use the **LOF** function.

Example

This example uses the **FileLen** function to return the length of a file in bytes. For purposes of this example, assume that `TESTFILE` is a file containing some data.

```
Dim MySize As Long
MySize = FileLen("TESTFILE")   ' Returns file length (bytes).
```

Smart Device Developer Notes

This function is not supported.

Requirements

Namespace: Microsoft.VisualBasic

Module: FileSystem

Assembly: Microsoft Visual Basic .NET Runtime (in MicrosoftVisualBasic.dll)

See Also

FileDateTime Function | GetAttr Function | LOF Function | FileNotFoundException Class

FileOpen Function

Opens a file for input or output.

```
Public Sub FileOpen( _
   ByVal FileNumber As Integer, _
   ByVal FileName As String, _
   ByVal Mode As OpenMode, _
   Optional ByVal Access As OpenAccess = OpenAccess.Default, _
   Optional ByVal Share As OpenShare = OpenShare.Default, _
   Optional ByVal RecordLength As Integer = -1 _
)
```

Parameters

FileNumber
Required. Any valid file number. Use the **FreeFile** function to obtain the next available file number.
FileName
Required. **String** expression that specifies a file name — may include directory or folder, and drive.
Mode
Required. **Enum** specifying the file mode: **Append**, **Binary**, **Input**, **Output**, or **Random**.
Access
Optional. Keyword specifying the operations permitted on the open file: **Read**, **Write**, or **ReadWrite**. Defaults to **ReadWrite**.
Share
Optional. **Enum** specifying the operations restricted on the open file by other processes: **Shared**, **Lock Read**, **Lock Write**, and **Lock Read Write**. Defaults to **Lock Read Write**.
RecordLength
Optional. Number less than or equal to 32,767 (bytes). For files opened for random access, this value is the record length. For sequential files, this value is the number of characters buffered.

Exceptions/Errors

Exception type	Error number	Condition
ArgumentException	5	Invalid **Access, Share,** or **Mode.**
ArgumentException	5	**WriteOnly** file is opened for **Input.**
ArgumentException	5	**ReadOnly** file is opened for **Output.**
ArgumentException	5	**ReadOnly** file is opened for **Append.**
ArgumentException	5	Record length is negative (and not equal to -1).
IOException	52	*FileNumber* is invalid (<-1 or >255), or *File Number* is already in use.
IOException	55	*FileName* is already open, or *FileName* is in valid.

Remarks

You must open a file before any I/O operation can be performed on it. **FileOpen** allocates a buffer for I/O to the file and determines the mode of access to use with the buffer.

> **Security Note** When writing to files, an application may need to create a file if the file to which it is trying to write does not exist. To do so, it needs permission for the directory in which the file is to be created. However, if the file specified by *FileName* does exist, the application only needs **Write** permission to the file itself. Wherever possible, it is more secure to create the file during deployment and only grant **Write** permission to that file, rather than to the entire directory. It is also more secure to write data to user directories than to the root directory or the Program Files directory.

The channel to open can be found using the **FreeFile()** function.

> **Security Note** The **FileOpen** function requires **Read** access from the **FileIOPermissionAccess** enumeration, which may affect its execution in partial trust situations. For more information, see FileIOPermissionAccess Enumeration and

Requesting Permissions.

Example

This example illustrates various uses of the **FileOpen** function to enable input and output to a file.

The following code opens the file `TESTFILE` in **Input** mode.

```
FileOpen(1, "TESTFILE", OpenMode.Input)
' Close before reopening in another mode.
FileClose(1)
```

This example opens the file in **Binary** mode for writing operations only.

```
FileOpen(1, "TESTFILE", OpenMode.Binary,OpenAccess.Write)
' Close before reopening in another mode.
FileClose(1)
```

The following example opens the file in **Random** mode. The file contains records of the structure `Person`.

```
Structure Person
   <VBFixedString(30)> Dim Name As String
   Dim ID As Integer
End Structure
' Count 30 for the string, plus 4 for the integer.
FileOpen(1, "TESTFILE", OpenMode.Random, , , 34)
' Close before reopening in another mode.
FileClose(1)
```

This code example opens the file in **Output** mode; any process can read or write to file.

```
FileOpen(1, "TESTFILE", OpenMode.Output, OpenShare.Shared)
' Close before reopening in another mode.
FileClose(1)
```

This code example opens the file in **Binary** mode for reading; other processes can't read file.

```
FileOpen(1, "TESTFILE", OpenMode.Binary, OpenAccess.Read, _
   OpenShare.LockRead)
```

Smart Device Developer Notes

This function is not supported.

Requirements

Namespace: Microsoft.VisualBasic

Module: FileSystem

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

FileClose Function | FreeFile Function | File Access with Visual Basic Run-Time Functions | ArgumentException Class | IOException Class

FilePut Function

Writes data from a variable to a disk file.

```
Public Overloads Sub FilePut( _
   FileNumber As Integer, _
   Value As Short, _
   Optional RecordNumber As Integer = -1 _
)
```

-or-

```
Public Overloads Sub FilePut( _
   FileNumber As Integer, _
   Value As Integer, _
   Optional RecordNumber As Integer = -1 _
)
```

-or-

```
Public Overloads Sub FilePut( _
   FileNumber As Integer, _
   Value As Single, _
   Optional RecordNumber As Integer = -1 _
)
```

-or-

```
Public Overloads Sub FilePut( _
   FileNumber As Integer, _
   Value As Double, _
   RecordNumber As Integer = -1 _
)
```

-or-

```
Public Overloads Sub FilePut( _
   FileNumber As Integer, _
   Value As Decimal, _
   Optional RecordNumber As Integer = -1 _
)
```

-or-

```
Public Overloads Sub FilePut( _
   FileNumber As Integer, _
   Value As Byte, _
   Optional RecordNumber As Integer = -1 _
)
```

-or-

```
Public Overloads Sub FilePut( _
   FileNumber As Integer, _
   Value As Boolean, _
   Optional RecordNumber As Integer = -1 _
)
```

-or-

```
Public Overloads Sub FilePut( _
   FileNumber As Integer, _
   Value As Date, _
   Optional RecordNumber As Integer = -1 _
)
```

-or-

```
Public Overloads Sub FilePut( _
   FileNumber As Integer, _
   Value As System.Array, _
   Optional RecordNumber As Integer = -1, _
   Optional ArrayIsDynamic As Boolean = False _
)
```

-or-

```
Public Overloads Sub FilePut (
   FileNumber As Integer,
   Value As String, _
   Optional RecordNumber As Integer = -1,
   Optional StringIsFixedLength As Boolean = False
)
```

Parameters

FileNumber
Required. Any valid file number.
Value
Required. Valid variable name containing data written to disk.
RecordNumber
Optional. Record number (**Random** mode files) or byte number (**Binary** mode files) at which writing begins.
ArrayIsDynamic
Optional. Applies only when writing an array. Specifies whether the array is to be treated as dynamic, and whether to write an array descriptor for the string describing the length.
StringIsFixedLength
Optional. Applies only when writing a string. Specifies whether to write a two-byte string length descriptor for the string to the file. The default is **False**.

Exceptions/Errors

Exception type	Error number	Condition
ArgumentException	63	*RecordNumber* < 1 and not equal to -1 .
IOException	52	*FileNumber* does not exist.
IOException	54	File mode is invalid.

Remarks

FilePut is only valid in **Random** and **Binary** mode.

Data written with **FilePut** is usually read from a file with **FileGet**.

The first record or byte in a file is at position 1, the second record or byte at position 2, and so on. If you omit *RecordNumber*, the next record or byte after the last **FileGet** or **FilePut** function or pointed to by the last **Seek** function is written.

The *StringIsFixedLength* argument controls whether the function interprets strings as variable or fixed length. **FilePut** does not write the length descriptor when the argument is **True**. If you use *StringIsFixedLength* = **True** with **FilePut**, you will need to do the same with **FileGet**, and also make sure the string is initialized to the length expected.

For files opened in **Random** mode, the following rules apply:

- If the length of the data being written is less than the length specified in the *RecordLength* clause of the **FileOpen** function, **FilePut** writes subsequent records on record-length boundaries. The space between the end of one record and the beginning of the next record is padded with the existing contents of the file buffer. Because the amount of padding data can't be determined with any certainty, it is generally a good idea to have the record length match the length of the data being written. If the length of the data being written is greater than the length specified in the *RecordLength* clause of the **FileOpen** function, an exception will be thrown.
- If the variable being written is a string, **FilePut** writes a two-byte descriptor containing the string length, and then writes the data that goes into the variable. Therefore, the record length specified by the *RecordLength* clause in the **FileOpen** function must be at least two bytes greater than the actual length of the string.
- If the variable being written is an object containing a numeric type, **FilePut** writes two bytes identifying the **VarType** of the object and then writes the variable. For example, when writing an object containing an integer, **FilePut** writes six bytes: two bytes identifying the object as **VarType(**3**)** (**Integer**) and four bytes containing the data. The record length specified by the *RecordLength* parameter in the **FileOpen** function must be at least two bytes greater than the actual number of bytes required to store the variable.
- If the variable being written is an object containing a string, **FilePut** writes a two byte descriptor identifying the **VarType(**8**)** of the object, a two-byte descriptor indicating the length of the string, and then writes the string data. The record length specified by the *RecordLength* parameter in the **FileOpen** function must be at least four bytes greater than the actual length of the string. If you wish to put a string without the descriptor, then you should pass **True** to the *StringIsFixedLength* parameter, and the string you read into should be the correct length.
- If the variable being written is an array, then you have a choice as to whether or not to write a descriptor for the size and dimensions of the array. Visual Basic 6 and earlier versions would write the file descriptor for a dynamic array, but not for a fixed-size array. Visual Basic .NET defaults to not writing the descriptor. To write the descriptor, set the *ArrayIsDynamic* parameter to **True**. When writing the array, you need to match the way the array will be read; if it will be read with the descriptor, then you need to write the descriptor. The descriptor specifies the rank of the array, the size, and the lower bounds for each rank. Its length equals 2 plus 8 times the number of dimensions, that is, 2 + 8 * *NumberOfDimensions*. The record length specified by the *RecordLength* clause in the **FileOpen** function must be greater than or equal to the sum of all the bytes required to write the array data and the array descriptor. For example, the following array declaration requires 118 bytes when the array is written to disk.

  ```
  Dim MyArray(4,9) As Integer
  ```

- If the variable being written is any other type of variable (not a variable-length string or an object), **FilePut** writes only the variable data. The record length specified by the *RecordLength* clause in the **FileOpen** function must be greater than or equal to the length of the data being written.
- **FilePut** writes elements of structures as if each were written individually, except there is no padding between elements. The **VBFixedString** attribute can be applied to string fields in the structures to indicate the size of the string when written to disk.

For files opened in **Binary** mode, all of the **Random** mode rules apply, except:

- The *RecordLength* clause in the **FileOpen** function has no effect. **FilePut** writes all variables to disk contiguously; that is, with no padding between records.
- For any array other than an array in a structure, **FilePut** writes only the data. No descriptor is written.
- **FilePut** writes variable-length strings that are not elements of structures without the two-byte length descriptor. The number of bytes written equals the number of characters in the string. For example, the following statements write 11 bytes to file number 1:

  ```
  Dim hellow As String = "Hello World"
  FilePut(1,hellow)
  ```

- Writing to a file with the **FilePut** function requires **Write** access from the **FileIOPermissionAccess** enumeration. For more information, see FileIOPermissionAccess Enumeration.

Example

This example uses the **FilePut** function to write data to a file. Five records of the structure `Person` are written to the file.

```
Structure Person
   Public ID As Integer
```

```
    Public Name As String
End Structure

Sub WriteData()
    Dim MyRecord As Person
    Dim recordNumber As Integer
'     Open file for random access.
    FileOpen(1, "C:\TESTFILE.txt", OpenMode.Binary)
    For recordNumber = 1 To 5    ' Loop 5 times.
        MyRecord.ID = recordNumber    ' Define ID.
        MyRecord.Name = "My Name" & recordNumber    ' Create a string.
        FilePut(1, MyRecord)    ' Write record to file.
    Next recordNumber
    FileClose(1)
End Sub
```

Smart Device Developer Notes

This function is not supported.

Requirements

Namespace: Microsoft.VisualBasic

Module: FileSystem

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

FileGet Function | FileOpen Function | Seek Function | File Access with Visual Basic Run-Time Functions | FileGetObject Function | ArgumentException Class | IOException Class

FilePutObject Function

Writes data from a variable to a disk file.

```
Public Sub FilePutObject( _
   FileNumber As Integer, _
   Value As Object, _
   RecordNumber As Integer = -1 _
)
```

-or-

```
Overloads Public Sub FilePutObject( _
   FileNumber As Integer, _
   Value As Short, _
   Optional RecordNumber As Integer = -1 _
)
```

-or-

```
Overloads Public Sub FilePutObject( _
   FileNumber As Integer, _
   Value As Integer, _
   Optional RecordNumber As Integer = -1 _
)
```

-or-

```
Overloads Public Sub FilePutObject( _
   FileNumber As Integer, _
   Value As Single, _
   Optional RecordNumber As Integer = -1 _
)
```

-or-

```
Overloads Public Sub FilePutObject( _
   FileNumber As Integer, _
   Value As Double, _
   RecordNumber As Integer = -1 _
)
```

-or-

```
Overloads Public Sub FilePutObject( _
   FileNumber As Integer, _
   Value As Decimal, _
   Optional RecordNumber As Integer = -1 _
)
```

-or-

```
Overloads Public Sub FilePutObject( _
   FileNumber As Integer, _
   Value As Byte, _
   Optional RecordNumber As Integer = -1 _
)
```

-or-

```
Overloads Public Sub FilePutObject( _
   FileNumber As Integer, _
   Value As Boolean, _
   Optional RecordNumber As Integer = -1 _
)
```

-or-

```
Overloads Public Sub FilePutObject( _
   FileNumber As Integer, _
   Value As Date, _
   Optional RecordNumber As Integer = -1 _
)
```

-or-

```
Overloads Public Sub FilePutObject( _
   FileNumber As Integer, _
   Value As System.Array, _
   Optional RecordNumber As Integer = -1, _
   Optional ArrayIsDynamic As Boolean = False _
)
```

-or-

```
Overloads Public Sub FilePutObject( _
   FileNumber As Integer, _
   Value As String, _
   Optional RecordNumber As Integer = -1, _
   Optional StringIsFixedLength as Boolean = False _
)
```

Parameters

FileNumber
Required. Any valid file number.
Value
Required. Valid variable name containing data written to disk.
RecordNumber
Optional. Record number (**Random** mode files) or byte number (**Binary** mode files) at which writing begins.
ArrayIsDynamic
Optional. Applies only when writing an array. Specifies whether the array is to be treated as dynamic, and whether to write an array descriptor for the string describing the length.
StringIsFixedLength
Optional. Applies only when writing a string. Specifies whether to write a descriptor for the string describing the length. The default is **False**.

Remarks

The **FilePutObject** function is used in place of **FilePut** to avoid ambiguities at compile time if type **Object** is passed rather than another type, such as **Integer**, **Long**, **Short**, and so forth.

FilePutObject writes and reads descriptors that describe the object. If you intend to write out the **Variant** type, **FilePutObject** is required. When in doubt, if you are using an object for the second parameter, it is always safer to use **FilePutObject** and **FileGetObject**.

FilePutObject is only valid in **Random** and **Binary** mode.

Data written with **FilePutObject** is usually read from a file with **FileGetObject**.

The first record or byte in a file is at position 1, the second record or byte is at position 2, and so on. If you omit *RecordNumber*, **FilePutObject** writes the next record or byte after the last **FileGetObject** or **FilePutObject** function (or the record or byte

pointed to by the last **Seek** function).

The *StringIsFixedLength* argument controls whether the function interprets strings as variable or fixed length. **FilePutObject** does not write the length descriptor when the argument is **True**. If you use *StringIsFixedLength* = **True** with **FilePutObject**, you need to do the same with **FileGetObject**, and must also make sure that the string is initialized to the length expected.

For files opened in **Random** mode, the following rules apply:

- If the length of the data being written is less than the length specified in the *RecordLength* clause of the **FileOpen** function, **FilePutObject** writes subsequent records on record-length boundaries. The space between the end of one record and the beginning of the next record is padded with the existing contents of the file buffer. Because the amount of padding data can't be precisely determined, it is generally a good idea to have the record length match the length of the data being written. If the length of the data being written is greater than the length specified in the *RecordLength* clause of the **FileOpen** function, an exception is thrown.
- If the variable being written is an object containing a numeric type, **FilePutObject** writes two bytes identifying the **VarType** of the object and then writes the variable. For example, when writing an object containing an integer, **FilePutObject** writes six bytes: two bytes identifying the object as **VarType(**3**)** (**Integer**) and four bytes containing the data. The record length specified by the *RecordLength* parameter in the **FileOpen** function must be at least two bytes greater than the actual number of bytes required to store the variable.
- If the variable being written is an object containing a string, **FilePutObject** writes a two-byte descriptor identifying the **VarType(**8**)** of the object, a two-byte descriptor indicating the length of the string, and then writes the string data. The record length specified by the *RecordLength* parameter in the **FileOpen** function must be at least four bytes greater than the actual length of the string. If you wish to put a string without the descriptor, then you should pass **True** to the *StringIsFixedLength* parameter, and the string you read into should be the correct length.
- If the variable being written is an array, you can choose to write a descriptor for the size and dimensions of the array. Visual Basic 6 and earlier versions would write the file descriptor for a dynamic array, but not for a fixed-size array; Visual Basic .NET defaults to not writing the descriptor. To write the descriptor, set the *ArrayIsDynamic* parameter to **True**. When writing the array, you need to match the way the array is read; if it is read with the descriptor, you need to write the descriptor. The descriptor specifies the rank of the array, the size, and the lower bounds for each rank. Its length equals 2 plus 8 times the number of dimensions, that is, 2 + 8 * *NumberOfDimensions*. The record length specified by the *RecordLength* clause in the **FileOpen** function must be greater than or equal to the sum of all the bytes required to write the array data and the array descriptor.

For files opened in **Binary** mode, all of the **Random** mode rules apply, except:

- The *RecordLength* clause in the **FileOpen** function has no effect. **FilePutObject** writes all variables to disk contiguously, that is, with no padding between records.

Example

This example uses the **FilePutObject** function to write a string to a file.

```
Sub WriteData()
    Dim text As String = "test"
    FileOpen(1, "test.bin", OpenMode.Binary)
    FilePutObject(1, text)
    FileClose(1)
End Sub
```

Smart Device Developer Notes

This function is not supported.

Requirements

Namespace: Microsoft.VisualBasic

Module: FileSystem

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

FileWidth Function

Assigns an output line width to a file opened using the **FileOpen** function.

```
Public Sub FileWidth( _
   FileNumber As Integer, _
   RecordWidth As Integer _
)
```

Parameters

FileNumber
Required. Any valid file number.

RecordWidth
Required. Numeric expression in the range 0–255, inclusive, which indicates how many characters appear on a line before a new line is started. If *RecordWidth* equals 0, there is no limit to the length of a line. The default value for *RecordWidth* is 0.

Exceptions/Errors

Exception type	Error number	Condition
IOException	52	*FileNumber* does not exist.
IOException	54	File mode is invalid.

Example

This example uses the **FileWidth** function to set the output line width for a file.

```
Dim i As Integer
FileOpen(1, "TESTFILE", OpenMode.Output) ' Open file for output.
FileWidth(1, 5)   ' Set output line width to 5.
For i = 0 To 9   ' Loop 10 times.
   Print(1, Chr(48 + I))   ' Prints five characters per line.
Next
FileClose(1)   ' Close file.
```

Smart Device Developer Notes

This function is not supported.

Requirements

Namespace: Microsoft.VisualBasic

Module: FileSystem

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

FileOpen Function | Print, PrintLine Functions | IOException Class

Filter Function

Returns a zero-based array containing a subset of a **String** array based on specified filter criteria.

```
Function Filter(
   ByVal Source() As { Object | String },
   ByVal Match As String,
   Optional ByVal Include As Boolean = True,
   Optional ByVal Compare As CompareMethod = CompareMethod.Binary
)  As String()
```

Parameters

Source
Required. One-dimensional array of strings to be searched.
Match
Required. String to search for.
Include
Optional. **Boolean** value indicating whether to return substrings that include or exclude *Match*. If *Include* is **True**, the **Filter** function returns the subset of the array that contains *Match* as a substring. If *Include* is **False**, the **Filter** function returns the subset of the array that does not contain *Match* as a substring.
Compare
Optional. Numeric value indicating the kind of string comparison to use. See Settings for values.

Settings

The *Compare* argument can have the following values:

Constant	Description
CompareMethod.Binary	Performs a binary comparison
CompareMethod.Text	Performs a textual comparison

Exceptions/Errors

Exception type	Error number	Condition
ArgumentException	5	*Source* is **Nothing** or is not a one-dimensional array.

Remarks

If no matches of *Match* are found within *Source*, the **Filter** function returns an empty array. An error occurs if *Source* is set to **Nothing** or is not a one-dimensional array.

The array returned by the **Filter** function contains only enough elements to contain the number of matched items.

Example

This example demonstrates the use of the **Filter** function.

```
Dim myStrings(2) As String
MyStrings(0) = "This"
MyStrings(1) = "Is"
MyStrings(2) = "It"
' Returns ["This", "Is"].
Dim subStrings() As String = Filter(myStrings, "is", True, _
   CompareMethod.Text)
' Returns ["This"].
Dim subStrings() As String = Filter(myStrings, "is", True, _
   CompareMethod.Binary)
' Returns ["Is", "It"].
Dim subStrings() As String = Filter(myStrings, "is", False, _
```

```
    CompareMethod.Binary)
```

Requirements

Namespace: Microsoft.VisualBasic

Module: Strings

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Replace Function | ArgumentException

Int, Fix Functions

Return the integer portion of a number.

```
Public Shared Function Int(ByVal number As { Double | Integer | Long | Object | Short | Singl
e | Decimal }) As { Double | Integer | Long | Object | Short | Single | Decimal }
Public Shared Function Fix(ByVal number As { Double | Integer | Long | Object | Short | Singl
e | Decimal }) As { Double | Integer | Long | Object | Short | Single | Decimal }
```

Parameter

number
Required. A number of type **Double** or any valid numeric expression. If *number* contains Nothing, **Nothing** is returned.

Exceptions/Errors

Exception type	Error number	Condition
ArgumentNullException	5	*Number* is not specified.
ArgumentException	5	*Number* is not a numeric type .

Remarks

Both the **Int** and **Fix** functions remove the fractional part of *number* and return the resulting integer value.

The difference between **Int** and **Fix** functions is that if *number* is negative, **Int** returns the first negative integer less than or equal to *number*, whereas **Fix** returns the first negative integer greater than or equal to *number*. For example, **Int** converts -8.4 to -9, and **Fix** converts -8.4 to -8.

`Fix(number)` is equivalent to `Sign(number) * Int(Abs(number))`.

Example

This example illustrates how the **Int** and **Fix** functions return integer portions of numbers. In the case of a negative number argument, the **Int** function returns the first negative integer less than or equal to the number; the **Fix** function returns the first negative integer greater than or equal to the number. The following example requires you to specify **Option Strict Off** because implicit conversions from type **Double** to type **Integer** are not allowed under **Option Strict On**:

```
Option Strict Off
' Code will not function unless placed within a Sub or Function
Dim MyNumber As Integer
MyNumber = Int(99.8)   ' Returns 99.
MyNumber = Fix(99.8)   ' Returns 99.

MyNumber = Int(-99.8)   ' Returns -100.
MyNumber = Fix(-99.8)   ' Returns -99.

MyNumber = Int(-99.2)   ' Returns -100.
MyNumber = Fix(-99.2)   ' Returns -99.
```

You can use the **CInt** function to explicitly convert other data types to type **Integer** with **Option Strict Off**. However, **Cint** rounds to the nearest integer instead of truncating the fractional part of numbers. For example:

```
MyNumber = CInt(99.8)    ' Returns 100.
MyNumber = CInt(-99.8)   ' Returns -100.
MyNumber = CInt(-99.2)   ' Returns -99.
```

You can use the **CInt** function on the result of a call to **Fix** or **Int** to perform explicit conversion to integer without rounding. For example:

```
MyNumber = CInt(Fix(99.8))   ' Returns 99.
MyNumber = CInt(Int(99.8))   ' Returns 99.
```

Requirements

Namespace: Microsoft.VisualBasic

Module: Conversion

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

CInt Function | Integer Data Type | Math Summary | Math Functions | ArgumentNullException Class

Format Function

Returns a string formatted according to instructions contained in a format **String** expression.

```
Public Shared Function Format( _
   ByVal Expression As Object, _
   Optional ByVal Style As String = "" _
) As String
```

Parameters

Expression
Required. Any valid expression.
Style
Optional. A valid named or user-defined format **String** expression.

Settings

For information on how to create the *Style* argument, see the appropriate topic listed below:

To format	Do this
Numbers	Use predefined numeric formats or create user-defined numeric formats.
Dates and times	Use predefined date/time formats or create user-defined date/time formats.
Date and time serial numbers	Use date and time formats or numeric formats.

If you try to format a number without specifying *Style*, the **Format** function provides functionality similar to the **Str** function, although it is internationally aware. However, positive numbers formatted as strings using the **Format** function don't include a leading space reserved for the sign of the value; those converted using the **Str** function retain the leading space.

Remarks

If you are formatting a nonlocalized numeric string, you should use a user-defined numeric format to ensure that you get the look you want.

The **String.Format** method also provides similar functionality.

Example

This example shows various uses of the **Format** function to format values using both **String** formats and user-defined formats. For the date separator (**/**), time separator (**:**), and the AM/PM indicators (**t** and **tt**), the actual formatted output displayed by your system depends on the locale settings the code is using. When times and dates are displayed in the development environment, the short time format and short date format of the code locale are used.

```
Dim MyDateTime As Date = #1/27/2001 5:04:23 PM#
Dim MyStr As String
' Returns current system time in the system-defined long time format.
MyStr = Format(Now(), "Long Time")
' Returns current system date in the system-defined long date format.
MyStr = Format(Now(), "Long Date")
' Also returns current system date in the system-defined long date
' format, using the single letter code for the format.
MyStr = Format(Now(), "D")
' Returns the value of MyDateTime in user-defined date/time formats.
MyStr = Format(MyDateTime, "h:m:s")   ' Returns "5:4:23".
MyStr = Format(MyDateTime, "hh:mm:ss tt")   ' Returns "05:04:23 PM".
MyStr = Format(MyDateTime, "dddd, MMM d yyyy")   ' Returns "Saturday,
   ' Jan 27 2001".
MyStr = Format(MyDateTime, "HH:mm:ss")   ' Returns "17:04:23"
MyStr = Format(23)   ' Returns "23".
' User-defined numeric formats.
MyStr = Format(5459.4, "##,##0.00")   ' Returns "5,459.40".
MyStr = Format(334.9, "###0.00")   ' Returns "334.90".
```

```
MyStr = Format(5, "0.00%")   ' Returns "500.00%".
```

Requirements

Namespace: Microsoft.VisualBasic

Module: Strings

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Different Formats for Different Numeric Values (Format Function) | Predefined Date/Time Formats (Format Function) | Predefined Numeric Formats (Format Function) | Str Function | Type Conversion Functions | User-Defined Date/Time Formats (Format Function) | User-Defined Numeric Formats (Format Function) | String.Format Method

Different Formats for Different Numeric Values (Format Function)

A user-defined format expression for numbers can have from one to three sections separated by semicolons. If the style argument of the **Format** function contains one of the Predefined Numeric Formats, only one section is allowed.

If you use	This is the result
One section only	The format expression applies to all values.
Two sections	The first section applies to positive values and zeros; the second applies to negative values.
Three sections	The first section applies to positive values, the second applies to negative values, and the third applies to zeros.

The following example has two sections: the first defines the format for positive values and zeros; the second section defines the format for negative values. Since the *Style* argument of the **Format** function takes a string, it is enclosed by quotation marks.

```
"$#,##0;($#,##0)"
```

If you include semicolons with nothing between them, the missing section is printed using the format of the positive value. For example, the following format displays positive and negative values using the format in the first section and displays `Zero` if the value is zero.

```
"$#,##0;;\Z\e\r\o"
```

Requirements

Namespace: Microsoft.VisualBasic

Module: Strings

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Format Function

Predefined Numeric Formats (Format Function)

The following table identifies the predefined numeric format names. These may be used by name as the style argument for the **Format** function:

Format name	Description
General Number, **G**, or **g**	Displays number with no thousand separator.
Currency, **C**, or **c**	Displays number with thousand separator, if appropriate; display two digits to the right of the decimal separator. Output is based on system locale settings.
Fixed, **F**, or **f**	Displays at least one digit to the left and two digits to the right of the decimal separator.
Standard, **N**, or **n**	Displays number with thousand separator, at least one digit to the left and two digits to the right of the decimal separator.
Percent	Displays number multiplied by 100 with a percent sign (%) appended immediately to the right; always displays two digits to the right of the decimal separator.
P, or **p**	Displays number with thousandths separator multiplied by 100 with a percent sign (%) appended to the right and separated by a single space; always displays two digits to the right of the decimal separator.
Scientific	Uses standard scientific notation, providing two significant digits.
E, or **e**	Uses standard scientific notation, providing six significant digits.
D, or **d**	Displays number as a string that contains the value of the number in Decimal (base 10) format. This option is supported for integral types **(Byte**, **Short**, **Integer**, **Long**) only.
X, or **x**	Displays number as a string that contains the value of the number in Hexadecimal (base 16) format. This option is supported for integral types (**Byte**, **Short**, **Integer**, **Long**) only.
Yes/No	Displays `No` if number is 0; otherwise, displays `Yes`.
True/False	Displays `False` if number is 0; otherwise, displays `True`.
On/Off	Displays `Off` if number is 0; otherwise, displays `On`.

Smart Device Developer Notes

The **Yes/No**, **True/False**, and **On/Off** formats are not supported.

Requirements

Namespace: Microsoft.VisualBasic

Module: Strings

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Format Function | Predefined Date/Time Formats (Format Function) | User-Defined Numeric Formats (Format Function)

Predefined Date/Time Formats (Format Function)

The following table identifies the predefined date and time format names. These may be used by name as the style argument for the **Format** function:

Format Name	Description
General Date, or **G**	Displays a date and/or time. For real numbers, display a date and time; for example, `4/3/93 05:34 PM`.If there is no fractional part, display only a date, for example, `4/3/93`. If there is no integer part, display time only, for example, `05:34 PM`. Date display is determined by your system's **LocaleID** value.
Long Date, or **D**	Displays a date according to your locale's long date format.
Medium Date	Displays a date using the medium date format appropriate for the language version of the host application.
Short Date, or **d**	Displays a date using your locale's short date format.
Long Time, or **T**	Displays a time using your locale's long time format; includes hours, minutes, seconds.
Medium Time	Displays time in 12-hour format using hours and minutes and the AM/PM designator.
Short Time, or **t**	Displays a time using the 24-hour format, for example, `17:45`.
f	Displays the long date and short time according to your locale's format.
F	Displays the long date and long time according to your locale's format.
g	Displays the short date and short time according to your locale's format.
M, **m**	Displays the month and the day of a date.
R, **r**	Formats the date and time as Greenwich Mean Time (GMT)
s	Formats the date and time as a sortable index.
u	Formats the date and time as a GMT sortable index.
U	Formats the date and time with the long date and long time as GMT.
Y, y	Formats the date as the year and month.

Requirements

Namespace: Microsoft.VisualBasic

Module: Strings

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Format Function | Predefined Numeric Formats (Format Function) | User-Defined Date/Time Formats (Format Function)

User-Defined Date/Time Formats (Format Function)

The following table shows characters you can use to create user-defined date/time formats. Unlike in previous versions of Visual Basic, these format characters are case-sensitive

Character	Description
(**:**)	Time separator. In some locales, other characters may be used to represent the time separator. The time separator separates hours, minutes, and seconds when time values are formatted. The actual character used as the time separator in formatted output is determined by your system's **LocaleID** value.
(**/**)	Date separator. In some locales, other characters may be used to represent the date separator. The date separator separates the day, month, and year when date values are formatted. The actual character used as the date separator in formatted output is determined by your locale.
(**%**)	Used to indicate that the following character should be read as a single-letter format without regard to any trailing letters. Also used to indicate that a single-letter format is read as a user-defined format. See below for further details
d	Displays the day as a number without a leading zero (for example, `1`). Use **%d** if this is the only character in your user-defined numeric format.
dd	Displays the day as a number with a leading zero (for example, `01`).
ddd	Displays the day as an abbreviation (for example, `Sun`).
dddd	Displays the day as a full name (for example, `Sunday`).
M	Displays the month as a number without a leading zero (for example, January is represented as `1`). Use **%M** if this is the only character in your user-defined numeric format.
MM	Displays the month as a number with a leading zero (for example, `01/12/01`).
MMM	Displays the month as an abbreviation (for example, `Jan`).
MMMM	Displays the month as a full month name (for example, `January`).
gg	Displays the period/era string (for example, `A.D.`)
h	Displays the hour as a number without leading zeros using the 12-hour clock (for example, `1:15:15 PM`). Use **%h** if this is the only character in your user-defined numeric format.
hh	Displays the hour as a number with leading zeros using the 12-hour clock (for example, `01:15:15 PM`).
H	Displays the hour as a number without leading zeros using the 24-hour clock (for example, `1:15:15`). Use **%H** if this is the only character in your user-defined numeric format.
HH	Displays the hour as a number with leading zeros using the 24-hour clock (for example, `01:15:15`).
m	Displays the minute as a number without leading zeros (for example, `12:1:15`). Use **%m** if this is the only character in your user-defined numeric format.
mm	Displays the minute as a number with leading zeros (for example, `12:01:15`).
s	Displays the second as a number without leading zeros (for example, `12:15:5`). Use **%s** if this is the only character in your user-defined numeric format.
ss	Displays the second as a number with leading zeros (for example, `12:15:05`).
F	Displays fractions of seconds. For example **ff** will display hundredths of seconds, whereas **ffff** will display ten-thousandths of seconds. You may use up to seven **f** symbols in your user-defined format. Use **%f** if this is the only character in your user-defined numeric format.
T	Uses the 12-hour clock and displays an uppercase `A` for any hour before noon; displays an uppercase `P` for any hour between noon and 11:59 P.M. Use **%t** if this is the only character in your user-defined numeric format.
tt	Uses the 12-hour clock and displays an uppercase `AM` with any hour before noon; displays an uppercase `PM` with any hour between noon and 11:59 P.M.
y	Displays the year number (0-9) without leading zeros. Use **%y** if this is the only character in your user-defined numeric format.
yy	Displays the year in two-digit numeric format with a leading zero, if applicable.
yyy	Displays the year in four digit numeric format.
yyyy	Displays the year in four digit numeric format.
z	Displays the timezone offset without a leading zero (for example, `-8`). Use **%z** if this is the only character in your user-defined numeric format.
zz	Displays the timezone offset with a leading zero (for example, `-08`)
zzz	Displays the full timezone offset (for example, `-08:00`)

Example

The following are examples of user-defined date and time formats for `December 7, 1958, 8:50 PM, 35 seconds`:

Format	Displays
M/d/yy	12/7/58
d-MMM	7-Dec
d-MMMM-yy	7-December-58
d MMMM	7 December
MMMM yy	December 58
hh:mm tt	08:50 PM
h:mm:ss t	8:50:35 P
H:mm	20:50
H:mm:ss	20:50:35
M/d/yyyy H:mm	12/7/1958 20:50

Requirements

Namespace: Microsoft.VisualBasic

Module: Strings

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Format Function | Predefined Date/Time Formats (Format Function) | User-Defined Numeric Formats (Format Function)

User-Defined Numeric Formats (Format Function)

The following table identifies characters you can use to create user-defined number formats. These may be used to build the style argument for the **Format** function:

Character	Description
None	Displays the number with no formatting.
(**0**)	Digit placeholder. Displays a digit or a zero. If the expression has a digit in the position where the zero appears in the format string, display it; otherwise, displays a zero in that position. If the number has fewer digits than there are zeros (on either side of the decimal) in the format expression, displays leading or trailing zeros. If the number has more digits to the right of the decimal separator than there are zeros to the right of the decimal separator in the format expression, rounds the number to as many decimal places as there are zeros. If the number has more digits to the left of the decimal separator than there are zeros to the left of the decimal separator in the format expression, displays the extra digits without modification.
(**#**)	Digit placeholder. Displays a digit or nothing. If the expression has a digit in the position where the **#** character appears in the format string, displays it; otherwise, displays nothing in that position. This symbol works like the **0** digit placeholder, except that leading and trailing zeros aren't displayed if the number has fewer digits than there are **#** characters on either side of the decimal separator in the format expression.
(**.**)	Decimal placeholder. The decimal placeholder determines how many digits are displayed to the left and right of the decimal separator. If the format expression contains only **#** characters to the left of this symbol; numbers smaller than 1 begin with a decimal separator. To display a leading zero displayed with fractional numbers, use zero as the first digit placeholder to the left of the decimal separator. In some locales, a comma is used as the decimal separator. The actual character used as a decimal placeholder in the formatted output depends on the number format recognized by your system. Thus, You should use the period as the decimal placeholder in your formats even if you are in a locale that uses a comma as a decimal placeholder. The formatted string will appear in the format correct for the locale.
(**%**)	Percent placeholder. Multiplies the expression by 100. The percent character (**%**) is inserted in the position where it appears in the format string.
(**,**)	Thousand separator. The thousand separator separates thousands from hundreds within a number that has four or more places to the left of the decimal separator. Standard use of the thousand separator is specified if the format contains a thousand separator surrounded by digit placeholders (**0** or **#**). A thousand separator immediately to the left of the decimal separator (whether or not a decimal is specified) or as the rightmost character in the string means "scale the number by dividing it by 1,000, rounding as needed." For example, you can use the format string `"##0,."` to represent 100 million as 100,000. Numbers smaller than 1,000 but greater or equal to 500 are displayed as `1`, and numbers smaller than 500 are displayed as `0`. Two adjacent thousand separators in this position scale by a factor of 1 million, and an additional factor of 1,000 for each additional separator. Multiple separators in any position other than immediately to the left of the decimal separator or the rightmost position in the string are treated simply as specifying the use of a thousand separator. In some locales, a period is used as a thousand separator. The actual character used as the thousand separator in the formatted output depends on the Number Format recognized by your system. Thus, You should use the comma as the thousand separator in your formats even if you are in a locale that uses a period as a thousand separator. The formatted string will appear in the format correct for the locale.
(**:**)	Time separator. In some locales, other characters may be used to represent the time separator. The time separator separates hours, minutes, and seconds when time values are formatted. The actual character used as the time separator in formatted output is determined by your system settings.
(**/**)	Date separator. In some locales, other characters may be used to represent the date separator. The date separator separates the day, month, and year when date values are formatted. The actual character used as the date separator in formatted output is determined by your system settings.

(**E- E+ e- e+**)	Scientific format. If the format expression contains at least one digit placeholder (**0** or **#**) to the left of **E-**, **E+**, **e-**, or **e+**, the number is displayed in scientific format and `E` or `e` is inserted between the number and its exponent. The number of digit placeholders to the left determines the number of digits in the exponent. Use **E-** or **e-** to place a minus sign next to negative exponents. Use **E+** or **e+** to place a minus sign next to negative exponents and a plus sign next to positive exponents. You must also include digit placeholders to the right of this symbol to get correct formatting.
- + $ ( )	Literal characters. These characters are displayed exactly as typed in the format string. To display a character other than one of those listed, precede it with a backslash (****) or enclose it in double quotation marks (" ").
(****)	Displays the next character in the format string. To display a character that has special meaning as a literal character, precede it with a backslash (****). The backslash itself isn't displayed. Using a backslash is the same as enclosing the next character in double quotation marks. To display a backslash, use two backslashes (****). Examples of characters that can't be displayed as literal characters are the date-formatting and time-formatting characters (**a**, **c**, **d**, **h**, **m**, **n**, **p**, **q**, **s**, **t**, **w**, **y**, **/**, and **:**), the numeric-formatting characters (**#**, **0**, **%**, **E**, **e**, comma, and period), and the string-formatting characters (**@**, **&**, **<**, **>**, and **!**).
(**"ABC"**)	Displays the string inside the double quotation marks (" "). To include a string in the style argument from within code, you must use **Chr(34)** to enclose the text (**34** is the character code for a quotation mark (").

Example

The following table contains some sample format expressions for numbers. (These examples all assume that your system's locale setting is English-U.S.) The first column contains the format strings for the *Style* argument of the **Format** function; the other columns contain the resulting output if the formatted data has the value given in the column headings.

Format (*Style*)	**"5" formatted as**	**"-5" formatted as**	**"0.5" formatted as**
`Zero-length string ("")`	`5`	`-5`	`0.5`
`0`	`5`	`-5`	`1`
`0.00`	`5.00`	`-5.00`	`0.50`
`#,##0`	`5`	`-5`	`1`
`$#,##0;($#,##0)`	`$5`	`($5)`	`$1`
`$#,##0.00;` `($#,##0.00)`	`$5.00`	`($5.00)`	`$0.50`
`0%`	`500%`	`-500%`	`50%`
`0.00%`	`500.00%`	`-500.00%`	`50.00%`
`0.00E+00`	`5.00E+00`	`-5.00E+00`	`5.00E-01`
`0.00E-00`	`5.00E00`	`-5.00E00`	`5.00E-01`

Requirements

Namespace: Microsoft.VisualBasic

Module: Strings

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Different Formats for Different Numeric Values (Format Function) | Format Function |
Predefined Numeric Formats (Format Function) | User-Defined Date/Time Formats (Format Function)

FormatCurrency Function

Returns an expression formatted as a currency value using the currency symbol defined in the system control panel.

```
Function FormatCurrency(
   ByVal Expression As Object,
   Optional ByVal NumDigitsAfterDecimal As Integer = -1,
   Optional ByVal IncludeLeadingDigit As TriState = TriState.UseDefault,
   Optional ByVal UseParensForNegativeNumbers As TriState = TriState.UseDefault,
   Optional ByVal GroupDigits As TriState = TriState.UseDefault
) As String
```

Parameters

Expression
Required. Expression to be formatted.

NumDigitsAfterDecimal
Optional. Numeric value indicating how many places are displayed to the right of the decimal. Default value is –1, which indicates that the computer's regional settings are used.

IncludeLeadingDigit
Optional. Tristate enumeration that indicates whether or not a leading zero is displayed for fractional values. See Settings for values.

UseParensForNegativeNumbers
Optional. Tristate enumeration that indicates whether or not to place negative values within parentheses. See Settings for values.

GroupDigits
Optional. Tristate enumeration that indicates whether or not numbers are grouped using the group delimiter specified in the computer's regional settings. See Settings for values.

Settings

The *IncludeLeadingDigit*, *UseParensForNegativeNumbers*, and *GroupDigits* arguments take the following Tristate enumeration values:

Value	Description
TriState.True	True
TriState.False	False
TriState.UseDefault	The computer's regional settings

Exceptions/Errors

Exception type	Error number	Condition
ArgumentException	5	Number of digits after decimal point is greater than 99.
InvalidCastException	13	Type is not numeric.

Remarks

When one or more optional arguments are omitted, the computer's matching regional-settings values are used instead.

The position of the currency symbol relative to the currency value is determined by the system's regional settings.

> **Note** All settings information comes from the locale of the application. By default, that will be the locale set in the control panel. However, it may be changed programmatically by using the .NET Framework, except leading zero, which comes from the **Number** tab.

Example

The following example illustrates the use of the **FormatCurrency** function.

```
Dim myDebt As Double = -4456.43
Dim myString As String
' Returns"($4,456.43)".
MyString = FormatCurrency(myDebt,,, TriState.True, TriState.True)
```

Requirements

Namespace: Microsoft.VisualBasic

Module: Strings

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

FormatDateTime Function | FormatNumber Function | FormatPercent Function | Tristate Enumeration | ArgumentException | InvalidCastException

FormatDateTime Function

Returns a string expression representing a date/time value.

```
Function FormatDateTime(
   ByVal Expression As DateTime,
   Optional ByVal NamedFormat As DateFormat = DateFormat.GeneralDate
) As String
```

Parameters

Expression
Required. **Date** expression to be formatted.
NamedFormat
Optional. Numeric value that indicates the date/time format used. If omitted, **DateFormat.GeneralDate** is used.

Settings

The *NamedFormat* argument has the following settings:

Constant	Description
DateFormat.GeneralDate	Display a date and/or time. If there is a date part, display it as a short date. If there is a time part, display it as a long time. If present, both parts are displayed.
DateFormat.LongDate	Display a date using the long date format specified in your computer's regional settings.
DateFormat.ShortDate	Display a date using the short date format specified in your computer's regional settings.
DateFormat.LongTime	Display a time using the time format specified in your computer's regional settings.
DateFormat.ShortTime	Display a time using the 24-hour format (hh:mm).

Exceptions/Errors

Exception type	Error number	Condition
ArgumentException	5	*NamedFormat* setting is invalid.

Remarks

The **Date** data type always contains both date and time information. For purposes of type conversion, Visual Basic .NET considers 1/1/1 (January 1 of the year 1) to be a neutral value for the date, and 00:00:00 (midnight) to be a neutral value for the time. If you format a **Date** value as a date/time string, **FormatDateTime** does not include neutral values in the resulting string. For example, if you convert #1/1/0001 9:30:00# to a string, the result is "9:30:00 AM"; the date information is suppressed. However, the date information is still present in the original **Date** value and can be recovered with functions such as **DatePart**.

> **Note** If you pass the *Expression* argument as a **String** literal, **FormatDateTime** interprets it according to the **CultureInfo.CurrentCulture** setting of your application. However, if you pass it as a **Date** literal, you should always use the format #mm/dd/yyyy#, because **FormatDateTime** always interprets a **Date** literal according to the English (US) culture. This is necessary because, if an application is developed and coded using **Date** literals from one culture, but is then executed on a platform with a different culture, the **Date** literals could be parsed incorrectly.

Example

This example demonstrates the use of the **FormatDateTime** function.

```
Dim myDate As DateTime = #3/12/1999#   ' English (US) format.
Dim myString As String
' FormatDateTime returns "Friday, March 12, 1999".
' The time information is neutral (00:00:00) and therefore suppressed.
myString = FormatDateTime(myDate, DateFormat.LongDate)
```

Requirements

Namespace: Microsoft.VisualBasic

Module: Strings

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

FormatCurrency Function | FormatNumber Function | FormatPercent Function | ArgumentException | DatePart Function | CultureInfo Class | CultureInfo.CurrentCulture Property

FormatNumber Function

Returns an expression formatted as a number.

```
Function FormatNumber(
   ByVal Expression As Object,
   Optional ByVal NumDigitsAfterDecimal As Integer = -1,
   Optional ByVal IncludeLeadingDigit As TriState = TriState.UseDefault,
   Optional ByVal UseParensForNegativeNumbers As TriState = TriState.UseDefault,
   Optional ByVal GroupDigits As TriState = TriState.UseDefault
) As String
```

Parameters

Expression
Required. Expression to be formatted.

NumDigitsAfterDecimal
Optional. Numeric value indicating how many places are displayed to the right of the decimal. The default value is –1, which indicates that the computer's regional settings are used.

IncludeLeadingDigit
Optional. Tristate constant that indicates whether a leading zero is displayed for fractional values. See Settings for values.

UseParensForNegativeNumbers
Optional. Tristate constant that indicates whether to place negative values within parentheses. See Settings for values.

GroupDigits
Optional. Tristate constant that indicates whether or not numbers are grouped using the group delimiter specified in the locale settings. See Settings for values.

Settings

The *IncludeLeadingDigit*, *UseParensForNegativeNumbers*, and *GroupDigits* arguments have the following settings:

Constant	Description
TriState.True	True
TriState.False	False
TriState.UseDefault	The computer's regional settings

Exceptions/Errors

Exception type	Error number	Condition
InvalidCastException	13	Type is not numeric.

Remarks

When one or more optional arguments are omitted, the values for omitted arguments are provided by the locale settings.

> **Note** All settings information comes from the locale of the application. By default, that will be the locale set in the control panel. However, it may be changed programmatically by using the .NET Framework.

Example

This example demonstrates the **FormatNumber** function.

```
Dim myNumber As Integer = 45600
Dim myString As String
' Returns "45,600.00".
myString = FormatNumber(myNumber, 2, , ,TriState.True)
```

Requirements

Namespace: Microsoft.VisualBasic

Module: Strings

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

FormatCurrency Function | FormatDateTime Function | FormatPercent Function | Tristate Enumeration | InvalidCastException

FormatPercent Function

Returns an expression formatted as a percentage (that is, multiplied by 100) with a trailing % character.

```
Function FormatPercent(
   ByVal Expression As Object,
   Optional ByVal NumDigitsAfterDecimal As Integer = -1,
   Optional ByVal IncludeLeadingDigit As TriState = TriState.UseDefault,
   Optional ByVal UseParensForNegativeNumbers As TriState = TriState.UseDefault,
   Optional ByVal GroupDigits As TriState = TriState.UseDefault
) As String
```

Parameters

Expression
Required. Expression to be formatted.

NumDigitsAfterDecimal
Optional. Numeric value indicating how many places to the right of the decimal are displayed. Default value is –1, which indicates that the locale settings are used.

IncludeLeadingDigit
Optional. Tristate constant that indicates whether or not a leading zero is displayed for fractional values. See Settings for values.

UseParensForNegativeNumbers
Optional. Tristate constant that indicates whether or not to place negative values within parentheses. See Settings for values.

GroupDigits
Optional. Tristate constant that indicates whether or not numbers are grouped using the group delimiter specified in the locale settings. See Settings for values.

Settings

The *IncludeLeadingDigit*, *UseParensForNegativeNumbers*, and *GroupDigits* arguments have the following settings:

Constant	Description
TriState.True	True
TriState.False	False
TriState.Default	The computer's regional settings

Exceptions/Errors

Exception type	Error number	Condition
InvalidCastException	13	Type is not numeric.

Remarks

When one or more optional arguments are omitted, the values for omitted arguments are provided by the locale settings.

> **Note** All settings information comes from the locale of the application. By default, that will be the locale set in the control panel. However, it may be changed programmatically by using the .NET Framework.

Example

This example illustrates the use of the **FormatPercent** function:

```
Dim myNumber As Single = 0.76
Dim myString As String
' Returns "76.00%".
myString = FormatPercent(myNumber)
```

Requirements

Namespace: Microsoft.VisualBasic

Module: Strings

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

FormatCurrency Function | FormatDateTime Function | FormatNumber Function | Tristate Enumeration | InvalidCastException

FreeFile Function

Returns an **Integer** value representing the next file number available for use by the **FileOpen** function.

```
Public Function FreeFile() As Integer
```

Exceptions/Errors

Exception type	Error number	Condition
IOException	67	More than 255 files are in use.

Remarks

Use **FreeFile** to supply a file number that is not already in use.

Example

This example uses the **FreeFile** function to return the next available file number. Five files are opened for output within the loop, and some sample data is written to each.

```
Dim count As Integer
Dim fileNumber As Integer
For count = 1 To 5
   fileNumber = FreeFile()
   FileOpen(fileNumber, "TEST" & count & ".TXT", OpenMode.Output)
   PrintLine(fileNumber, "This is a sample.")
   FileClose(fileNumber)
Next
```

Smart Device Developer Notes

This function is not supported.

Requirements

Namespace: Microsoft.VisualBasic

Module: FileSystem

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

FileOpen Function | IOException Class

FV Function

Returns a **Double** specifying the future value of an annuity based on periodic, fixed payments and a fixed interest rate.

```
Function FV( _
   ByVal Rate As Double, _
   ByVal NPer As Double, _
   ByVal Pmt As Double, _
   Optional ByVal PV As Double = 0, _
   Optional ByVal Due As DueDate = DueDate.EndOfPeriod _
) As Double
```

Parameters

Rate
Required. **Double** specifying interest rate per period. For example, if you get a car loan at an annual percentage rate (APR) of 10 percent and make monthly payments, the rate per period is 0.1/12, or 0.0083.

NPer
Required. **Double** specifying total number of payment periods in the annuity. For example, if you make monthly payments on a four-year car loan, your loan has a total of 4 * 12 (or 48) payment periods.

Pmt
Required. **Double** specifying payment to be made each period. Payments usually contain principal and interest that doesn't change over the life of the annuity.

PV
Optional. **Double** specifying present value (or lump sum) of a series of future payments. For example, when you borrow money to buy a car, the loan amount is the present value to the lender of the monthly car payments you will make. If omitted, 0 is assumed.

Due
Optional. Object of type `Microsoft.VisualBasic.DueDate` that specifies when payments are due. This argument must be either `DueDate.EndOfPeriod` if payments are due at the end of the payment period, or `DueDate.BegOfPeriod` if payments are due at the beginning of the period. If omitted, `DueDate.EndOfPeriod` is assumed.

Remarks

An annuity is a series of fixed cash payments made over a period of time. An annuity can be a loan (such as a home mortgage) or an investment (such as a monthly savings plan).

The *Rate* and *NPer* arguments must be calculated using payment periods expressed in the same units. For example, if *Rate* is calculated using months, *NPer* must also be calculated using months.

For all arguments, cash paid out (such as deposits to savings) is represented by negative numbers; cash received (such as dividend checks) is represented by positive numbers.

Example

This example uses the **FV** function to return the future value of an investment given the percentage rate that accrues per period (`APR / 12`), the total number of payments (`TotPmts`), the payment (`Payment`), the current value of the investment (`PVal`), and a number that indicates whether the payment is made at the beginning or end of the payment period (`PayType`). Note that because `Payment` represents cash paid out, it is a negative number.

```
Sub TestFV()
   Dim Fmt As String
   Dim TotPmts As Integer
   Dim Payment, APR, PVal, Fval As Double
   Dim PayType As DueDate
   Dim Response As MsgBoxResult
   Fmt = "###,###,##0.00"   ' Define money format.
   Payment = CDbl(InputBox("How much do you plan to save each month?"))
   APR = CDbl(InputBox("Enter the expected interest annual percentage rate."))
   If APR > 1 Then APR = APR / 100 ' Ensure proper form.
   TotPmts = CInt(InputBox("For how many months do you expect to save?"))
   Response = MsgBox("Do you make payments at the end of month?", MsgBoxStyle.YesNo)
```

```
    If Response = MsgBoxResult.No Then
       PayType = DueDate.BegOfPeriod
    Else
       PayType = DueDate.EndOfPeriod
    End If
    PVal = CDbl(InputBox("How much is in this savings account now?"))
    Fval = FV(APR / 12, TotPmts, -Payment, -PVal, PayType)
    MsgBox("Your savings will be worth " & Format(Fval, Fmt) & ".")
End Sub
```

Requirements

Namespace: Microsoft.VisualBasic

Module: Financial

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

DDB Function | IPmt Function | IRR Function | MIRR Function | NPer Function | NPV Function | Pmt Function | PPmt Function | PV Function | Rate Function | SLN Function | SYD Function

GetAllSettings Function

Returns a list of key settings and their respective values (originally created with **SaveSetting**) from an application's entry in the Windows registry.

```
Public Function GetAllSettings( _
   ByVal AppName As String, _
   ByVal Section As String _
) As String(,)
```

Parameters

AppName
Required. **String** expression containing the name of the application or project whose key settings are requested.

Section
Required. **String** expression containing the name of the section whose key settings are requested. **GetAllSettings** returns an object that contains a two-dimensional array of strings. The strings contain all the key settings in the specified section plus their corresponding values.

Exceptions/Errors

Exception type	Error number	Condition
ArgumentException	5	User is not logged in .

Remarks

GetAllSettings returns an uninitialized **Object** if either *AppName* or *Section* does not exist.

GetAllSettings requires that a user be logged on since it operates under the HKEY_LOCAL_USER registry key, which is not active until a user logs on interactively.

Registry settings that are to be accessed from a non-interactive process (such as mtx.exe) should be stored under either the HKEY_LOCAL_MACHINE\Software\ or the HKEY_USER\DEFAULT\Software registry keys.

Example

This example first uses the **SaveSetting** function to make entries in the Windows registry for the application specified as *AppName*, then uses the **GetAllSettings** function to display the settings. Note that application names and *Section* names can't be retrieved with **GetAllSettings**. Finally, the **DeleteSetting** function removes the application's entries.

```
' Object to hold 2-dimensional array returned by GetAllSettings.
' Integer to hold counter.
Dim MySettings(,) As String
Dim intSettings As Integer
' Place some settings in the registry.
SaveSetting("MyApp", "Startup", "Top", "75")
SaveSetting("MyApp", "Startup", "Left", "50")
' Retrieve the settings.
MySettings = GetAllSettings("MyApp", "Startup")
For intSettings = LBound(MySettings, 1) To UBound(MySettings, 1)
   Debug.WriteLine(MySettings(intSettings, 0))
   Debug.WriteLine(MySettings(intSettings, 1))
Next intSettings
DeleteSetting("MyApp")
```

Smart Device Developer Notes

This function is not supported.

Requirements

Namespace: Microsoft.VisualBasic

Module: Interaction

Assembly: Microsoft.Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

DeleteSetting Function | GetSetting Function | SaveSetting Function | ArgumentException Class

GetAttr Function

Returns a **FileAttribute** value representing the attributes of a file, directory, or folder.

```
Public Function GetAttr(ByVal PathName As String) As FileAttribute
```

Parameter

PathName
Required. **String** expression that specifies a file, directory, or folder name. *PathName* may include the directory or folder, and the drive.

Return Values

The value returned by **GetAttr** is the sum of the following enumeration values:

Value	Constant	Description
Normal	**vbNormal**	Normal.
ReadOnly	**vbReadOnly**	Read-only.
Hidden	**vbHidden**	Hidden.
System	**vbSystem**	System file.
Directory	**vbDirectory**	Directory or folder.
Archive	**vbArchive**	File has changed since last backup.
Alias	**vbAlias**	File has a different name.

Note These enumerations are specified by the Visual Basic language. The names can be used anywhere in your code in place of the actual values.

Exceptions/Errors

Exception type	Error number	Condition
IOException	52	*Pathname* is invalid or contains wildcards .
FileNotFoundException	53	Target file does not exist.

Remarks

To determine which attributes are set, use the **And** operator to perform a bitwise comparison of the value returned by the **GetAttr** function and the value of the individual file attribute you want. If the result is not zero, that attribute is set for the named file. For example, the return value of the following **And** expression is zero if the Archive attribute is not set:

```
Result = GetAttr(FName) And vbArchive
```

A nonzero value is returned if the **Archive** attribute is set.

Example

This example uses the **GetAttr** function to determine the attributes of a file and directory or folder.

```
Dim MyAttr As FileAttribute
' Assume file TESTFILE is normal and readonly.
MyAttr = GetAttr("C:\TESTFILE.txt")   ' Returns vbNormal.

' Test for normal.
If (MyAttr And FileAttribute.Normal) = FileAttribute.Normal Then
   MsgBox("This file is normal.")
End If

' Test for normal and readonly.
```

```
Dim normalReadonly As FileAttribute
normalReadonly = FileAttribute.Normal Or FileAttribute.ReadOnly
If (MyAttr And normalReadonly) = normalReadonly Then
   MsgBox("This file is normal and readonly.")
End If

' Assume MYDIR is a directory or folder.
MyAttr = GetAttr("C:\MYDIR")
If (MyAttr And FileAttribute.Directory) = FileAttribute.Directory Then
   MsgBox("MYDIR is a directory")
End If
```

Smart Device Developer Notes

This function is not supported.

Requirements

Namespace: Microsoft.VisualBasic

Module: FileSystem

Assembly: Microsoft Visual Basic .NET Runtime (in MicrosoftVisualBasic.dll)

See Also

And Operator | FileAttr Function | SetAttr Function | IOException Class | FileNotFoundException Class | FileAttribute Enumeration

GetChar Function

Returns a **Char** value representing the character from the specified index in the supplied string.

```
Public Shared Function GetChar( _
   ByVal Str As String, _
   ByVal Index As Integer _
) As Char
```

Parameters

Str
Required. Any valid **String** expression.
Index
Required. **Integer** expression. The (1-based) index of the character in *Str* to be returned.

Exceptions/Errors

Exception type	Error number	Condition
ArgumentException	5	*Str* is **Nothing**, *Index* < 1, or *Index* is greater than index of last character of *Str*.

Remarks

If *Index* is less than 1 or greater than the index of the last character in *Str*, an **ArgumentException** is thrown.

Example

This example shows how to use the **GetChar** function to return a character from a specified index in a **String**.

```
Dim myString As String = "ABCDE"
Dim myChar As Char
myChar = GetChar(myString, 4)   ' myChar = "D"
```

Requirements

Namespace: Microsoft.VisualBasic

Module: Strings

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Mid Function | ArgumentException Class

GetException Function

Returns the exception representing the error that occurred.

```
Overridable Public Function GetException() As Exception
```

Remarks

The **GetException** function is available only from the **Err** object class. It works with the **Exception** property of the **Err** object to display the error that has occurred.

Example

The following code displays the message assigned to the exception in the **Err** object:

```
On Error Resume Next
Dim myError As System.Exception
' Generate an overflow exception.
Err.Raise(6)
' Assigns the exception from the Err object to myError.
myError = Err.GetException()
' Displays the message associated with the exception.
MsgBox(myError.Message)
```

Requirements

Namespace: Microsoft.VisualBasic

Class: Err Object

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Err Object | Structured Exception Handling | Unstructured Exception Handling

GetObject Function

Returns a reference to an object provided by a COM component.

```
Public Function GetObject( _
   Optional ByVal PathName As String = Nothing, _
   Optional ByVal Class As String = Nothing _
) As Object
```

Parameters

PathName
Optional; **String**. The full path and name of the file containing the object to retrieve. If *pathname* is omitted, *class* is required.
Class
Optional; **String**. A string representing the class of the object.

The *Class* argument has the following syntax and parts:

```
appname.objecttype
```

Class Parameters

appname
Required; **String**. The name of the application providing the object.
objecttype
Required; **String**. The type or class of object to create.

Exceptions/Errors

Exception type	Error number	Condition
Exception	429	No object with the specified pathname exists.
FileNotFoundException	432	No object of the specified type exists.

Remarks

Use the **GetObject** function to load an instance of a COM component from a file. For example:

```
Dim CADObject As Object
CADObject = GetObject("C:\CAD\SCHEMA.CAD")
```

When this code is executed, the application associated with the specified *pathname* is started and the object in the specified file is activated.

If *PathName* is a zero-length string (""), **GetObject** returns a new object instance of the specified type. If the *PathName* argument is omitted, **GetObject** returns a currently active object of the specified type. If no object of the specified type exists, an error occurs.

Some applications allow you to activate a sub-object associated with a file. Add an exclamation point (**!**) to the end of the file name and follow it with a string that identifies the part of the file you want to activate. For information on how to create this string, see the documentation for the application that created the object.

For example, in a drawing application you might have multiple layers to a drawing stored in a file. You could use the following code to activate a layer within a drawing called `SCHEMA.CAD`:

```
LayerObject = GetObject("C:\CAD\SCHEMA.CAD!Layer3")
```

If you don't specify the object's *Class*, Automation determines the application to start and the object to activate, based on the file name you provide. Some files, however, may support more than one class of object. For example, a drawing might support three

different types of objects: an Application object, a Drawing object, and a Toolbar object, all of which are part of the same file. To specify which object in a file you want to activate, use the optional *Class* argument. For example:

```
Dim MyObject As Object
MyObject = GetObject("C:\DRAWINGS\SAMPLE.DRW", "FIGMENT.DRAWING")
```

In the example, `FIGMENT` is the name of a drawing application and `DRAWING` is one of the object types it supports.

Once an object is activated, you reference it in code using the object variable you defined. In the preceding example, you access properties and methods of the new object using the object variable `MyObject`. For example:

```
MyObject.Line (9, 90)
MyObject.InsertText (9, 100, "Hello, world.")
MyObject.SaveAs ("C:\DRAWINGS\SAMPLE.DRW")
```

> **Note** Use the **GetObject** function when there is a current instance of the object or if you want to create the object with a file already loaded. If there is no current instance, and you do not want the object started with a file loaded, use the **CreateObject** function. If an object has registered itself as an Active-X single-instance object, only one instance of the object is created, no matter how many times **CreateObject** is executed. With a single-instance object, **GetObject** always returns the same instance when called with the zero-length string ("") syntax, and it causes an error if the *PathName* argument is omitted. You cannot use **GetObject** to obtain a reference to a class created with Visual Basic.

Example

This example uses the **GetObject** function to get a reference to a specific Microsoft Excel worksheet (`MyXL`). It uses the worksheet's Application property to make Microsoft Excel visible, to close it, and so on. Using two API calls, the `DetectExcel` sub procedure looks for Microsoft Excel, and if it is running, enters it in the running object table. The first call to **GetObject** causes an error if Microsoft Excel isn't already running. In the example, the error causes the `ExcelWasNotRunning` flag to be set to `True`. The second call to **GetObject** specifies a file to open. If Microsoft Excel isn't already running, the second call starts it and returns a reference to the worksheet represented by the specified file, `test.xls`. The file must exist in the specified location; otherwise, the Visual Basic error Automation error is generated. Next the example code makes both Microsoft Excel and the window containing the specified worksheet visible.

Option Strict Off is required because this example uses late binding, where objects are assigned to generic object variables. You can specify **Option Strict On** and declare objects of specific object types if you add a reference to the Excel type library from the **COM** tab of the **Add Reference** dialog box of the **Project** menu in Visual Studio .NET.

```
' Add Option Strict Off to the top of your program.
Option Strict Off
 ' Declare necessary API routines:
   Declare Function FindWindow Lib "user32" Alias _
      "FindWindowA" (ByVal lpClassName As String, _
      ByVal lpWindowName As Long) As Long

   Declare Function SendMessage Lib "user32" Alias _
   "SendMessageA" (ByVal hWnd As Long, ByVal wMsg As Long, _
      ByVal wParam As Long, ByVal lParam As Long) As Long

Sub GetExcel()
   Dim MyXL As Object   ' Variable to hold reference
   ' to Microsoft Excel.
   Dim ExcelWasNotRunning As Boolean   ' Flag for final release.

   ' Test to see if there is a copy of Microsoft Excel already running.
   On Error Resume Next   ' Defer error trapping.
   ' Getobject function called without the first argument returns a
   ' reference to an instance of the application.
   ' If the application is not running, an error occurs.
   MyXL = GetObject(, "Excel.Application")
   If Err().Number <> 0 Then ExcelWasNotRunning = True
   Err().Clear() ' Clear Err object in case error occurred.

   ' Check for Microsoft Excel. If Microsoft Excel is running,
   ' enter it into the Running Object table.
```

```
    DetectExcel()

    ' Set the object variable to reference the file you want to see.
    MyXL = GetObject("c:\vb\TEST.XLS")

    ' Show Microsoft Excel through its Application property. Then
    ' show the actual window containing the file using the Windows
    ' collection of the MyXL object reference.
    MyXL.Application.Visible = True
    MyXL.Parent.Windows(1).Visible = True
    '  Do manipulations of your  file here.
    '
End Sub

Sub DetectExcel()
    ' Procedure dectects a running Excel and registers it.
    Const WM_USER = 1024
    Dim hWnd As Long
    ' If Excel is running this API call returns its handle.
    hWnd = FindWindow("XLMAIN", 0)
    If hWnd = 0 Then   ' 0 means Excel not running.
       Exit Sub
    Else
       ' Excel is running so use the SendMessage API
       ' function to enter it in the Running Object Table.
       SendMessage(hWnd, WM_USER + 18, 0, 0)
    End If
End Sub
```

When you call the `GetExcel` function, a check is made to see if Excel is already running. If it is not, then an instance of Excel is created.

Smart Device Developer Notes

This function is not supported.

Requirements

Namespace: Microsoft.VisualBasic

Module: Interaction

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

CreateObject Function | Declare Statement | Exception Class | FileNotFoundException Class

GetSetting Function

Returns a key setting value from an application's entry in the Windows registry.

```
Public Function GetSetting( _
   ByVal AppName As String, _
   ByVal Section As String, _
   ByVal Key As String, _
   Optional ByVal Default As String = "" _
) As String
```

Parameters

AppName
Required. **String** expression containing the name of the application or project whose key setting is requested.
Section
Required. **String** expression containing the name of the section in which the key setting is found.
Key
Required. **String** expression containing the name of the key setting to return.
Default
Optional. Expression containing the value to return if no value is set in the *Key* setting. If omitted, *Default* is assumed to be a zero-length string ("").

Exceptions/Errors

Exception type	Error number	Condition
ArgumentException	5	One or more arguments are not **String** expressions, or user is not logged in.

Remarks

If any of the items named in the **GetSetting** arguments do not exist, **GetSetting** returns a value of *Default*.

GetSetting requires that a user be logged on since it operates under the HKEY_LOCAL_USER registry key, which is not active until a user logs on interactively.

Registry settings that are to be accessed from a non-interactive process (such as mtx.exe) should be stored under either the HKEY_LOCAL_MACHINE\Software\ or the HKEY_USER\DEFAULT\Software registry keys.

GetSetting requires **Read** Registry Permission.

Example

This example first uses the **SaveSetting** function to make entries in the Windows registry for the application specified as *AppName*, and then uses the **GetSetting** function to display one of the settings. Because the *Default* argument is specified, some value is guaranteed to be returned. Note that *Section* names can't be retrieved with **GetSetting**. Finally, the **DeleteSetting** function removes all the application's entries.

```
' Place some settings in the registry.
SaveSetting("MyApp", "Startup", "Top", "75")
SaveSetting("MyApp", "Startup", "Left", "50")
Console.WriteLine(GetSetting("MyApp", "Startup", "Left", "25"))
DeleteSetting("MyApp")
```

Smart Device Developer Notes

This function is not supported.

Requirements

Namespace: Microsoft.VisualBasic

Module: Interaction

Assembly: Microsoft.Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

DeleteSetting Function | GetAllSettings Function | SaveSetting Function | ArgumentException Class | RegistryPermission Class

H-L Functions

The following table contains a listing of the Visual Basic .NET run-time member functions.

Hex	Hour	IIf	Input
InputBox	InputString	InStr	InStrRev
Int	IPmt	IRR	IsArray
IsDate	IsDBNull	IsError	IsNothing
IsNumeric	IsReference	Join	Kill
LBound	LCase	Left	Len
LineInput	Loc	Lock	LOF
Log	LSet	LTrim	

See Also

A-C Functions | D-G Functions | M-R Functions | S-Z Functions | Visual Basic Language and Run-Time Reference

Hex Function

Returns a string representing the hexadecimal value of a number.

```
Public Shared Function Hex( _
   ByVal Number As { Byte | Short | Integer | Long | Object } _
) As String
```

Parameter

Number
Required. Any valid numeric expression or **String** expression.

Exceptions/Errors

Exception type	Error number	Condition
ArgumentNullException	5	*Number* is not specified.
ArgumentException	5	*Number* is not a numeric type .

Remarks

If *Number* is not already a whole number, it is rounded to the nearest whole number before being evaluated.

If *Number* is	Hex returns
Empty	Zero (0)
Any numeric value	Up to sixteen hexadecimal characters

You can represent hexadecimal numbers directly by preceding numbers in the proper range with **&H**. For example, `&H10` represents decimal 16 in hexadecimal notation.

Example

This example uses the **Hex** function to return the hexadecimal value of a number.

```
Dim MyHex As String
MyHex = Hex(5)   ' Returns 5.
MyHex = Hex(10)   ' Returns A.
MyHex = Hex(459)   ' Returns 1CB.
```

Requirements

Namespace: Microsoft.VisualBasic

Module: Conversion

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Oct Function | Type Conversion Functions | ArgumentNullException Class

Hour Function

Returns an **Integer** value from 0 through 23 representing the hour of the day.

```
Public Function Hour(ByVal TimeValue As DateTime) As Integer
```

Parameter

TimeValue
Required. **Date** value from which you want to extract the hour.

You can also obtain the hour of the day by calling **DatePart** and specifying **DateInterval.Hour** for the *Interval* argument.

Example

This example uses the **Hour** function to obtain the hour from a specified time. In the development environment, the time literal is displayed in short time format using the locale settings of your code.

```
Dim MyTime As Date
Dim MyHour As Integer
MyTime = #4:35:17 PM#   ' Assign a time.
MyHour = Hour(MyTime)   ' MyHour contains 16.
```

Requirements

Namespace: Microsoft.VisualBasic

Module: DateAndTime

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Day Function | Minute Function | Now Property | Second Function | TimeOfDay Property | DatePart Function | System Namespace | DateTime Structure | ArgumentException Class | ArgumentOutOfRangeException Class

IIf Function

Returns one of two objects, depending on the evaluation of an expression.

```
Public Function IIf( _
   ByVal Expression As Boolean, _
   ByVal TruePart As Object, _
   ByVal FalsePart As Object _
) As Object
```

Parameters

Expression
Required. **Boolean**. The expression you want to evaluate.
TruePart
Required. **Object**. Returned if *Expression* evaluates to **True**.
FalsePart
Required. **Object**. Returned if *Expression* evaluates to **False**.

Remarks

> **Note** The expressions in the argument list can include function calls. As part of preparing the argument list for the call to **IIf**, the Visual Basic compiler calls every function in every expression. This means that you cannot rely on a particular function not being called if the other argument is selected by *Expression*.

Example

This example uses the **IIf** function to evaluate the `TestMe` parameter of the `CheckIt` procedure and returns the word "Large" if the amount is greater than 1000; otherwise, it returns the word "Small".

```
Function CheckIt (ByVal TestMe As Integer) As String
   CheckIt = IIf(TestMe > 1000, "Large", "Small")
End Function
```

Requirements

Namespace: Microsoft.VisualBasic

Module: Interaction

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Choose Function | If...Then...Else Statements | Select...Case Statements | Switch Function

Input Function

Reads data from an open sequential file and assigns the data to variables.

```
Public Sub Input( _
   FileNumber As Integer, _
   ByRef Value As Object _
)
```

Parameters

FileNumber
Required. Any valid file number.
Value
Required. Variable that is assigned the values read from the file — can't be an array or object variable.

Exceptions/Errors

Exception type	Error number	Condition
IOException	52	*FileNumber* does not exist.
IOException	54	File mode is invalid.

Remarks

Data read with **Input** is usually written to a file with **Write**. Use this function only with files opened in **Input** or **Binary** mode.

> **Security Note** When reading from files, do not make decisions about the contents of the file based on the file name extension. For example, a file named Form1.vb may not be a Visual Basic .NET source file.

When read, standard string or numeric data is assigned to variables without modification. The following table illustrates how other input data are treated:

Data	Value assigned to variable
Delimiting comma or blank line	Empty
#NULL#	**DBNull**
#TRUE# or #FALSE#	**True** or **False**
#yyyy-mm-dd hh:mm:ss#	The date and/or time represented by the expression
#ERROR *errornumber#*	*errornumber* (variable is an object tagged as an error)

If you reach the end of the file while you are inputting a data item, the input is terminated and an error occurs.

Note The **Input** function is not localized. For example, in the German version, if you input 3,14159, it will return only 3, since the comma is treated as a variable separator, rather than a decimal point.

> **Security Note** Reading from a file with the **Input** function requires **Read** access from the **FileIOPermissionAccess** enumeration. For more information, see FileIOPermissionAccess Enumeration.

Example

This example uses the **Input** function to read data from a file into two variables. This example assumes that `TESTFILE` is a file with a few lines of data written to it using the **Write** function; that is, each line contains a string in quotations and a number separated by a comma, for example, ("Hello", 234).

```
FileOpen(1, "TESTFILE", OpenMode.Output)
Write(1, "hello")
Write(1, 14)
FileClose(1)

Dim s As String
Dim i As Integer
FileOpen(1, "TESTFILE", OpenMode.Input)
```

```
Input(1, s)
Debug.WriteLine(s)
Input(1, i)
Debug.WriteLine(i)
FileClose(1)
```

Smart Device Developer Notes

This function is not supported.

Requirements

Namespace: Microsoft.VisualBasic

Module: FileSystem

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

InputString Function | FileOpen Function | Print, PrintLine Functions | Write, WriteLine Functions | File Access with Visual Basic Run-Time Functions| IOException Class

InputBox Function

Displays a prompt in a dialog box, waits for the user to input text or click a button, and then returns a string containing the contents of the text box.

```
Public Function InputBox( _
   ByVal Prompt As String, _
   Optional ByVal Title As String = "", _
   Optional ByVal DefaultResponse As String = "", _
   Optional ByVal XPos As Integer = -1, _
   Optional ByVal YPos As Integer = -1 _
) As String
```

Parameters

Prompt
Required. **String** expression displayed as the message in the dialog box. The maximum length of *Prompt* is approximately 1024 characters, depending on the width of the characters used. If *Prompt* consists of more than one line, you can separate the lines using a carriage return character (**Chr(**13**)**), a linefeed character (**Chr(**10**)**), or a carriage return–linefeed character combination (**Chr(**13**)** & **Chr(**10**)**) between each line.

Title
Optional. **String** expression displayed in the title bar of the dialog box. If you omit *Title*, the application name is placed in the title bar.

DefaultResponse
Optional. **String** expression displayed in the text box as the default response if no other input is provided. If you omit *DefaultResponse*, the displayed text box is empty.

XPos
Optional. Numeric expression that specifies, in twips, the distance of the left edge of the dialog box from the left edge of the screen. If you omit *XPos*, the dialog box is centered horizontally.

YPos
Optional. Numeric expression that specifies, in twips, the distance of the upper edge of the dialog box from the top of the screen. If you omit *YPos*, the dialog box is positioned vertically approximately one-third of the way down the screen.

Remarks

To specify more than the first argument, you must use the **InputBox** function in an expression. If you omit any positional arguments, you must retain the corresponding comma delimiter.

> **Note** The **InputBox** function requires UIPermission at the SafeTopLevelWindows level, which may affect its execution in partial trust situations. For more information, see Requesting Permissions and UIPermission Class.

Example

This example shows various ways to use the **InputBox** function to prompt the user to enter a value. If the x and y positions are omitted, the dialog box is automatically centered for the respective axes. The variable `MyValue` contains the value entered by the user if the user clicks OK or presses the ENTER key. If the user clicks Cancel, a zero-length string is returned.

```
Dim message, title, defaultValue As String
Dim myValue As Object
message = "Enter a value between 1 and 3"   ' Set prompt.
title = "InputBox Demo"   ' Set title.
DefaultValue = "1"   ' Set default value.

' Display message, title, and default value.
myValue = InputBox(message, title, defaultValue)

' Display dialog box at position 100, 100.
myValue = InputBox(message, title, defaultValue, 100, 100)
```

Requirements

Namespace: Microsoft.VisualBasic

Module: Interaction

Assembly: Microsoft.Visual Basic .NET Runtime (in MicrosoftVisualBasic.dll)

See Also

MsgBox Function | Chr, ChrW Functions

InputString Function

Returns **String** value containing characters from a file opened in **Input** or **Binary** mode.

```
InputString(_
   ByVal FileNumber As Integer, _
   ByVal CharCount As Integer _
) As String
```

Parameters

FileNumber
Required. Any valid file number.
CharCount
Required. Any valid numeric expression specifying the number of characters to read.

Exceptions/Errors

Exception type	Error number	Condition
IOException	52	*FileNumber* does not exist.
ArgumentException	5	*CharCount* < 0 or > 2^{14}.

Remarks

Data read with the **InputString** function is usually written to a file with **Print** or **FilePut**. Use this function only with files opened in **Input** or **Binary** mode.

Unlike the **Input** function, the **InputString** function returns all of the characters it reads, including commas, carriage returns, linefeeds, quotation marks, and leading spaces.

With files opened for **Binary** access, an attempt to read through the file using the **InputString** function until **EOF** returns **True** generates an error. Use the **LOF** and **Loc** functions instead of **EOF** when reading binary files with **InputString**, or use File**Get** when using the **EOF** function.

> **Security Note** When reading from files, do not make decisions about the contents of the file based on the file name extension. For example, a file named Form1.vb may not be a Visual Basic .NET source file.

Example

This example uses the **InputString** function to read one character at a time from a file and print it to the **Output** window. This example assumes that `MYFILE` is a text file with a few lines of sample data.

```
Dim oneChar As Char
FileOpen(1,  "MYFILE.TXT", OpenMode.Input) ' Open file.
While Not EOF(1)   ' Loop until end of file.
oneChar = (InputString(1, 1))   ' Get one character.
System.Console.Out.WriteLine(oneChar)   ' Print to the output window.
End While
FileClose(1)
```

Smart Device Developer Notes

This function is not supported.

Requirements

Namespace: Microsoft.VisualBasic

Module: FileSystem

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Input Function | IOException Class | ArgumentExceptionClass

InStr Function

Returns an integer specifying the start position of the first occurrence of one string within another.

```
Public Shared Function InStr(_
   Optional ByVal Start As Integer, _
   ByVal String1 As String, _
   ByVal String2 As String, _
) As Integer
```

-or-

```
Public Shared Function InStr(_
   Optional ByVal Start As Integer, _
   ByVal String1 As String, _
   ByVal String2 As String, _
   Optional ByVal Compare As Microsoft.VisualBasic.CompareMethod _
) As Integer
```

Parameters

Start
Optional. Numeric expression that sets the starting position for each search. If omitted, search begins at the first character position. The start index is 1 based.

String1
Required. **String** expression being searched.

String2
Required. **String** expression sought.

Compare
Optional. Specifies the type of string comparison. If *Compare* is omitted, the **Option Compare** setting determines the type of comparison. Specify a valid LCID (**LocaleID**) to use locale-specific rules in the comparison.

Settings

The *compare* argument settings are:

Constant	Value	Description
Binary	0	Performs a binary compariso n
Text	1	Performs a text comparison

Return Values

If	InStr returns
String1 is zero length or **Nothing**	0
String2 is zero length or **Nothing**	*start*
String2 is not found	0
String2 is found within *String1*	Position where match begins
Start > *String2*	0

Exceptions/Errors

Exception type	Error number	Condition
ArgumentException	5	*Start* < 1.

Note The **InStrB** function in previous versions of Visual Basic returns a number of bytes rather than a character position. It is used primarily for converting strings in double-byte character set (DBCS) applications. All Visual Basic .NET strings are in Unicode, and **InStrB** is no longer supported.

Example

This example uses the **InStr** function to return the position of the first occurrence of one string within another.

```
Dim SearchString, SearchChar As String
Dim MyPos As Integer

SearchString ="XXpXXpXXPXXP"   ' String to search in.
SearchChar = "P"   ' Search for "P".

' A textual comparison starting at position 4. Returns 6.
MyPos = InStr(4, SearchString, SearchChar, CompareMethod.Text)

' A binary comparison starting at position 1. Returns 9.
MyPos = InStr(1, SearchString, SearchChar, CompareMethod.Binary)

' Comparison is binary by default (last argument is omitted).
MyPos = InStr(SearchString, SearchChar)   ' Returns 9.

MyPos = InStr(1, SearchString, "W")   ' Returns 0.
```

Requirements

Namespace: Microsoft.VisualBasic

Module: Strings

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Strings | String Manipulation | InStrRev Function | Option Compare Statement | StrComp Function | ArgumentException Class | Programming Element Support Changes Summary

InStrRev Function

Returns the position of the first occurrence of one string within another, starting from the right side of the string.

```
Public Function InStrRev(
   ByVal StringCheck As String,
   ByVal StringMatch As String,
   Optional ByVal Start As Integer = -1,
   Optional ByVal Compare As CompareMethod = CompareMethod.Binary
) As Integer
```

Parameters

StringCheck
Required. String expression being searched.
StringMatch
Required. String expression being searched for.
Start
Optional. Numeric expression that sets the one-based starting position for each search, starting from the left side of the string. If *Start* is omitted, –1 is used, which means that the search begins at the last character position. Search then proceeds from right to left.
Compare
Optional. Numeric value indicating the kind of comparison to use when evaluating substrings. If omitted, a binary comparison is performed. See Settings for values.

Settings

The *Compare* argument can have the following values:

Constant	Description
Binary	Performs a binary comparison
Text	Performs a textual compariso n

Return Values

InStrRev returns the following values:

If	InStrRev returns
StringCheck is zero-length	0
StringMatch is zero-length	*start*
StringMatch is not found	0
StringMatch is found within *StringCheck*	Position at which the first match is found, starting with the left side of th e string.
Start is greater than length of *StringMatch*	0

Exceptions/Errors

Exception type	Error number	Condition
ArgumentException	5	*Start* = 0 or *Start* < -1 .

Remarks

Note that the syntax for the **InStrRev** function is not the same as the syntax for the **InStr** function.

Example

This example demonstrates the use of the **InStrRev** function:

```
Dim myString As String = "the quick brown fox jumps over the lazy dog"
Dim myNumber As Integer
MyNumber = InStrRev(myString, "the")   ' Returns 32.
MyNumber = InStrRev(myString, "the", 16)   ' Returns 1
```

Requirements

Namespace: Microsoft.VisualBasic

Module: Strings

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

InStr Function | Strings | String Manipulation

Int, Fix Functions

Return the integer portion of a number.

```
Public Shared Function Int(ByVal number As { Double | Integer | Long | Object | Short | Singl
e | Decimal }) As { Double | Integer | Long | Object | Short | Single | Decimal }
Public Shared Function Fix(ByVal number As { Double | Integer | Long | Object | Short | Singl
e | Decimal }) As { Double | Integer | Long | Object | Short | Single | Decimal }
```

Parameter

number
Required. A number of type **Double** or any valid numeric expression. If *number* contains Nothing, **Nothing** is returned.

Exceptions/Errors

Exception type	Error number	Condition
ArgumentNullException	5	*Number* is not specified.
ArgumentException	5	*Number* is not a numeric type .

Remarks

Both the **Int** and **Fix** functions remove the fractional part of *number* and return the resulting integer value.

The difference between **Int** and **Fix** functions is that if *number* is negative, **Int** returns the first negative integer less than or equal to *number*, whereas **Fix** returns the first negative integer greater than or equal to *number*. For example, **Int** converts -8.4 to -9, and **Fix** converts -8.4 to -8.

`Fix(number)` is equivalent to `Sign(number) * Int(Abs(number))`.

Example

This example illustrates how the **Int** and **Fix** functions return integer portions of numbers. In the case of a negative number argument, the **Int** function returns the first negative integer less than or equal to the number; the **Fix** function returns the first negative integer greater than or equal to the number. The following example requires you to specify **Option Strict Off** because implicit conversions from type **Double** to type **Integer** are not allowed under **Option Strict On**:

```
Option Strict Off
' Code will not function unless placed within a Sub or Function
Dim MyNumber As Integer
MyNumber = Int(99.8)   ' Returns 99.
MyNumber = Fix(99.8)   ' Returns 99.

MyNumber = Int(-99.8)   ' Returns -100.
MyNumber = Fix(-99.8)   ' Returns -99.

MyNumber = Int(-99.2)   ' Returns -100.
MyNumber = Fix(-99.2)   ' Returns -99.
```

You can use the **CInt** function to explicitly convert other data types to type **Integer** with **Option Strict Off**. However, **Cint** rounds to the nearest integer instead of truncating the fractional part of numbers. For example:

```
MyNumber = CInt(99.8)    ' Returns 100.
MyNumber = CInt(-99.8)   ' Returns -100.
MyNumber = CInt(-99.2)   ' Returns -99.
```

You can use the **CInt** function on the result of a call to **Fix** or **Int** to perform explicit conversion to integer without rounding. For example:

```
MyNumber = CInt(Fix(99.8))   ' Returns 99.
MyNumber = CInt(Int(99.8))   ' Returns 99.
```

Requirements

Namespace: Microsoft.VisualBasic

Module: Conversion

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

CInt Function | Integer Data Type | Math Summary | Math Functions | ArgumentNullException Class

IPmt Function

Returns a **Double** specifying the interest payment for a given period of an annuity based on periodic, fixed payments and a fixed interest rate.

```
Function IPmt( _
   ByVal Rate As Double, _
   ByVal Per As Double, _
   ByVal NPer As Double, _
   ByVal PV As Double, _
   Optional ByVal FV As Double = 0, _
   Optional ByVal Due As DueDate = DueDate.EndOfPeriod _
) As Double
```

Parameters

Rate
Required. **Double** specifying interest rate per period. For example, if you get a car loan at an annual percentage rate (APR) of 10 percent and make monthly payments, the rate per period is 0.1/12, or 0.0083.

Per
Required. **Double** specifying payment period in the range 1 through *NPer*.

NPer
Required. **Double** specifying total number of payment periods in the annuity. For example, if you make monthly payments on a four-year car loan, your loan has a total of 4 * 12 (or 48) payment periods.

PV
Required. **Double** specifying present value, or value today, of a series of future payments or receipts. For example, when you borrow money to buy a car, the loan amount is the present value to the lender of the monthly car payments you will make.

FV
Optional. **Double** specifying future value or cash balance you want after you've made the final payment. For example, the future value of a loan is $0 because that's its value after the final payment. However, if you want to save $50,000 over 18 years for your child's education, then $50,000 is the future value. If omitted, 0 is assumed.

Due
Optional. Object of type `Microsoft.VisualBasic.DueDate` that specifies when payments are due. This argument must be either `DueDate.EndOfPeriod` if payments are due at the end of the payment period, or `DueDate.BegOfPeriod` if payments are due at the beginning of the period. If omitted, `DueDate.EndOfPeriod` is assumed.

Exceptions/Errors

Exception type	Error number	Condition
ArgumentException	5	*Per* is invalid.

Remarks

An annuity is a series of fixed cash payments made over a period of time. An annuity can be a loan (such as a home mortgage) or an investment (such as a monthly savings plan).

The *Rate* and *NPer* arguments must be calculated using payment periods expressed in the same units. For example, if *Rate* is calculated using months, *NPer* must also be calculated using months.

For all arguments, cash paid out (such as deposits to savings) is represented by negative numbers; cash received (such as dividend checks) is represented by positive numbers.

Example

This example uses the **IPmt** function to calculate how much of a payment is interest when all the payments are of equal value. Given are the interest percentage rate per period (`APR / 12`), the payment period for which the interest portion is desired (`Period`), the total number of payments (`TotPmts`), the present value or principal of the loan (`PVal`), the future value of the loan (`FVal`), and a number that indicates whether the payment is due at the beginning or end of the payment period (`PayType`).

```
Sub TestIPMT()
   Dim Fmt, Msg As String
```

```
    Dim APR, PVal, Fval, Period, IntPmt, TotInt, TotPmts As Double
    Dim PayType As DueDate
    Dim Response As MsgBoxResult

    Fval = 0   ' Usually 0 for a loan.
    Fmt = "###,###,##0.00"   ' Define money format.
    PVal = CDbl(InputBox("How much do you want to borrow?"))
    APR = CDbl(InputBox("What is the annual percentage rate of your loan?"))
    If APR > 1 Then APR = APR / 100 ' Ensure proper form.
    TotPmts = CInt(InputBox("How many monthly payments?"))
    Response = MsgBox("Do you make payments at end of the month?", MsgBoxStyle.YesNo)
    If Response = MsgBoxResult.No Then
       PayType = DueDate.BegOfPeriod
    Else
       PayType = DueDate.EndOfPeriod
    End If
    For Period = 1 To TotPmts   ' Total all interest.
       IntPmt = IPmt(APR / 12, Period, TotPmts, -PVal, Fval, PayType)
       TotInt = TotInt + IntPmt
    Next Period
    Msg = "You'll pay a total of " & Format(TotInt, Fmt)
    Msg = Msg & " in interest for this loan."
    MsgBox(Msg)    ' Display results.
 End Sub
```

Requirements

Namespace: Microsoft.VisualBasic

Module: Financial

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

DDB Function | IRR Function | MIRR Function | NPer Function | NPV Function | Pmt Function | PPmt Function | PV Function | Rate Function | SLN Function | SYD Function | ArgumentException Class

IRR Function

Returns a **Double** specifying the internal rate of return for a series of periodic cash flows (payments and receipts).

```
Function IRR( _
    ByRef ValueArray() As Double, _
   Optional ByVal Guess As Double = 0.1 _
) As Double
```

Parameters

ValueArray()
Required. Array of **Double** specifying cash flow values. The array must contain at least one negative value (a payment) and one positive value (a receipt).

Guess
Optional. Object specifying value you estimate will be returned by **IRR**. If omitted, *Guess* is 0.1 (10 percent).

Exceptions/Errors

Exception type	Error number	Condition
ArgumentException	5	Array argument values are invalid.

Remarks

The internal rate of return is the interest rate received for an investment consisting of payments and receipts that occur at regular intervals.

The **IRR** function uses the order of values within the array to interpret the order of payments and receipts. Be sure to enter your payment and receipt values in the correct sequence. The cash flow for each period doesn't need to be fixed, as it is for an annuity.

IRR is calculated by iteration. Starting with the value of *Guess*, **IRR** cycles through the calculation until the result is accurate to within 0.00001 percent. If **IRR** can't find a result after 20 tries, it fails.

Example

In this example, the **IRR** function returns the internal rate of return for a series of 5 cash flows contained in the array `Values()`. The first array element is a negative cash flow representing business start-up costs. The remaining 4 cash flows represent positive cash flows for the subsequent 4 years. `Guess` is the estimated internal rate of return.

```
Sub TestIRR()
   Dim Guess, RetRate, Values(4) As Double
   Dim Fmt, Msg As String
   Guess = 0.1  ' Guess starts at 10 percent.
   Fmt = "#0.00"   ' Define percentage format.
   Values(0) = -70000   ' Business start-up costs.
   ' Positive cash flows reflecting income for four successive years.
   Values(1) = 22000 : Values(2) = 25000
   Values(3) = 28000 : Values(4) = 31000
   RetRate = IRR(Values, Guess) * 100   ' Calculate internal rate.
   Msg = "The internal rate of return for these five cash flows is "
   Msg = Msg & Format(RetRate, CStr(Fmt)) & " percent."
   MsgBox(Msg)    ' Display internal return rate.
End Sub
```

Requirements

Namespace: Microsoft.VisualBasic

Module: Financial

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

DDB Function | FV Function | IPmt Function | MIRR Function | NPer Function | NPV Function | Pmt Function | PPmt Function | PV Function | Rate Function | SLN Function | SYD Function | ArgumentException Class

IsArray Function

Returns a **Boolean** value indicating whether a variable points to an array.

```
Public Function IsArray(ByVal VarName As Object) As Boolean
```

Parameter

VarName
Required. **Object** variable.

Remarks

IsArray returns **True** if the variable points to an array; otherwise, it returns **False**. **IsArray** is especially useful with objects containing arrays.

Example

This example uses the **IsArray** function to check if several variables refer to an array.

```
Dim MyArray(4), YourArray(3) As Integer   ' Declare array variables.
Dim MyString As String
Dim MyCheck As Boolean
MyCheck = IsArray(MyArray)   ' Returns True.
MyCheck = IsArray(YourArray)   ' Returns True.
MyCheck = IsArray(MyString)   ' Returns False.
```

Requirements

Namespace: Microsoft.VisualBasic

Module: Information

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Declaring Array Variables | IsDate Function | IsDBNull Function | IsError Function | IsNothing Function | IsNumeric Function | IsReference Function | Object Data Type | TypeName Function

IsDate Function

Returns a **Boolean** value indicating whether an expression can be converted to a date.

```
Public Function IsDate(ByVal Expression As Object) As Boolean
```

Parameter

Expression
Required. **Object** expression.

Remarks

IsDate returns **True** if the expression is of type **Date** or is a string convertible to type **Date**; otherwise, it returns **False**.

Example

This example uses the **IsDate** function to determine if several variables can be converted to dates.

```
Dim MyDate, YourDate As Date
Dim NoDate As String
Dim MyCheck As Boolean
MyDate = "February 12, 1969"
YourDate = #2/12/1969#
NoDate = "Hello"
MyCheck = IsDate(MyDate)   ' Returns True.
MyCheck = IsDate(YourDate)   ' Returns True.
MyCheck = IsDate(NoDate)   ' Returns False.
```

Requirements

Namespace: Microsoft.VisualBasic

Module: Information

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

IsArray Function | IsDBNull Function | IsError Function | IsNothing Function | IsNumeric Function | IsReference Function | Object Data Type | Date Data Type | TypeName Function

IsDBNull Function

Returns a **Boolean** value indicating whether an expression evaluates to the **System.DBNull** class.

```
Public Function IsDBNull(ByVal Expression As Object) As Boolean
```

Parameter

Expression
Required. **Object** expression.

Remarks

IsDBNull returns **True** if the data type of *Expression* evaluates to the **DBNull** type; otherwise, **IsDBNull** returns **False**.

The **System.DBNull** value indicates that the **Object** represents missing or nonexistent data. **DBNull** is not the same as **Nothing**, which indicates that a variable has not yet been initialized. It is also not the same as a zero-length string (**""**), which is sometimes referred to as a null string.

Example

This example uses the **IsDBNull** function to determine if a variable evaluates to **DBNull**.

```
Dim MyVar As Object
Dim MyCheck As Boolean
MyCheck = IsDBNull(MyVar)   ' Returns False.
MyVar = ""
MyCheck = IsDBNull(MyVar)   ' Returns False.
MyVar = System.DBNull.Value
MyCheck = IsDBNull(MyVar)   ' Returns True.
MsgBox(MyCheck)
```

Requirements

Namespace: Microsoft.VisualBasic

Module: Information

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

IsArray Function | IsDate Function | IsError Function | IsNothing Function | IsNumeric Function | IsReference Function | Object Data Type | DBNull Class | TypeName Function

IsError Function

Returns a **Boolean** value indicating whether an expression is an exception type.

```
Public Function IsError(ByVal Expression As Object) As Boolean
```

Parameter

Expression
Required. **Object** expression.

Remarks

IsError returns **True** if the expression represents an **Object** variable that derives from the **Exception** class in the **System** namespace.

An exception that derives from **System.Exception** can be caught with the **Try...Catch...Finally** statements.

Example

This example uses the **IsError** function to check if an expression represents a system exception.

```
Dim ReturnVal As Object
Dim BadArg As String   ' Name of out-of-range argument.
Dim MyCheck As Boolean
' ...
ReturnVal = New System.ArgumentOutOfRangeException(BadArg)
' ...
MyCheck = IsError(ReturnVal)   ' Returns True.
```

Requirements

Namespace: Microsoft.VisualBasic

Module: Information

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

IsArray Function | IsDate Function | IsDBNull Function | IsNothing Function | IsNumeric Function | IsReference Function | Object Data Type | TypeName Function

IsNothing Function

Returns a **Boolean** value indicating whether an expression has no object assigned to it.

```
Public Function IsNothing(ByVal Expression As Object) As Boolean
```

Parameter

Expression
Required. **Object** expression.

Remarks

IsNothing returns **True** if the expression represents an **Object** variable that currently has no object assigned to it; otherwise, it returns **False**.

Example

This example uses the **IsNothing** function to determine if an object variable is associated with any object instance.

```
Dim MyVar As Object   ' No instance assigned to this variable yet.
Dim MyCheck As Boolean
' ...
MyCheck = IsNothing(MyVar)   ' Returns True.
' ...
MyVar = "ABCDEF"   ' Assign a string instance to the variable.
MyCheck = IsNothing(MyVar)   ' Returns False.
' ...
MyVar = Nothing   ' Disassociate the variable from any instance.
MyCheck = IsNothing(MyVar)   ' Returns True.
```

Requirements

Namespace: Microsoft.VisualBasic

Module: Information

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

IsArray Function | IsDate Function | IsDBNull Function | IsError Function | IsNumeric Function | IsReference Function | Object Data Type | TypeName Function | Nothing

IsNumeric Function

Returns a **Boolean** value indicating whether an expression can be evaluated as a number.

```
Public Function IsNumeric(ByVal Expression As Object) As Boolean
```

Parameter

Expression
Required. **Object** expression.

Remarks

IsNumeric returns **True** if the entire *Expression* is recognized as a number; otherwise, it returns **False**.

IsNumeric returns **True** if the data type of *Expression* is **Short**, **Integer**, **Long**, **Decimal**, **Single**, or **Short**. It also returns **True** if *Expression* is a **String** that can be successfully converted to a **Double**. It returns **False** if *Expression* is of data type **Date**.

Example

This example uses the **IsNumeric** function to determine if the contents of a variable can be evaluated as a number.

```
Dim MyVar As Object
Dim MyCheck As Boolean
' ...
MyVar = "53"   ' Assign value.
MyCheck = IsNumeric(MyVar)   ' Returns True.
' ...
MyVar = "459.95"   ' Assign value.
MyCheck = IsNumeric(MyVar)   ' Returns True.
' ...
MyVar = "45 Help"   ' Assign value.
MyCheck = IsNumeric(MyVar)   ' Returns False.
```

Requirements

Namespace: Microsoft.VisualBasic

Module: Information

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

IsArray Function | IsDate Function | IsDBNull Function | IsError Function | IsNothing Function | IsReference Function | Object Data Type | TypeName Function

IsReference Function

Returns a **Boolean** value indicating whether an expression evaluates to a reference type.

```
Public Function IsReference(ByVal Expression As Object) As Boolean
```

Parameter

Expression
Required. **Object** expression.

Remarks

IsReference returns **True** if the expression represents a reference type, such as a class instance, a **String** type, or an array of any type; otherwise, it returns **False**.

A reference type contains a pointer to data stored elsewhere in memory. A value type contains its own data.

Example

This example uses the **IsReference** function to check if several variables refer to reference types.

```
Dim MyArray(3) As Boolean
Dim MyString As String
Dim MyObject As Object
Dim MyNumber As Integer
MyArray(0) = IsReference(MyArray)   ' Returns True.
MyArray(1) = IsReference(MyString)   ' Returns True.
MyArray(2) = IsReference(MyObject)   ' Returns True.
MyArray(2) = IsReference(MyNumber)   ' Returns False.
```

Requirements

Namespace: Microsoft.VisualBasic

Module: Information

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

IsArray Function | IsDate Function | IsDBNull Function | IsError Function | IsNothing Function | IsNumeric Function | Object Data Type | TypeName Function

Join Function

Returns a string created by joining a number of substrings contained in an array.

```
Function Join(
   ByVal SourceArray() As { Object | String },
   Optional ByVal Delimiter As String = " "
) As String
```

Parameters

SourceArray()
Required. One-dimensional array containing substrings to be joined.

Delimiter
Optional. String used to separate the substrings in the returned string. If omitted, the space character (" ") is used. If *Delimiter* is a zero-length string (""), all items in the list are concatenated with no delimiters.

Exceptions/Errors

Exception type	Error number	Condition
ArgumentException	5	*SourceArray()* is not one dimensional.

Example

The following example demonstrates use of the **Join** function.

```
Dim myItem(2) As String
Dim myShoppingList As String
myItem(0) = "Pickle"
myItem(1) = "Pineapple"
myItem(2) = "Papaya"
' Returns "Pickle, Pineapple, Papaya"
myShoppingList = Join(myItem, ", ")
```

Requirements

Namespace: Microsoft.VisualBasic

Module: Strings

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Split Function | ArgumentException

Kill Function

Deletes files from a disk.

```
Public Sub Kill(ByVal PathName As String)
```

Parameter

PathName
Required. **String** expression that specifies one or more file names to be deleted. *PathName* may include the directory or folder, and the drive.

Exceptions/Errors

Exception type	Error number	Condition
IOException	55	Target file(s) open.
FileNotFoundException	53	Target file(s) not found.

Remarks

Kill supports the use of multiple-character (*****) and single-character (?) wildcards to specify multiple files.

Example

This example uses the **Kill** function to delete a file from a disk.

```
' Assume TESTFILE is a file containing some data.
Kill("TestFile")   ' Delete file.

' Delete all *.TXT files in current directory.
Kill("*.TXT")
```

Smart Device Developer Notes

This function is not supported.

Requirements

Namespace: Microsoft.VisualBasic

Module: FileSystem

Assembly: Microsoft Visual Basic .NET Runtime (in MicrosoftVisualBasic.dll)

See Also

RmDir Function | IOException Class | FileNotFoundException Class

LBound Function

Returns the lowest available subscript for the indicated dimension of an array.

```
Public Function LBound( _
   ByVal Array As System.Array, _
   Optional ByVal Rank As Integer = 1 _
) As Integer
```

Parameters

Array
Required. Array of any data type. The array in which you want to find the lowest possible subscript of a dimension.

Rank
Optional. **Integer**. The dimension for which the lowest possible subscript is to be returned. Use 1 for the first dimension, 2 for the second, and so on. If *Rank* is omitted, 1 is assumed.

Exceptions/Errors

Exception type	Error number	Condition
ArgumentNullException	9	*Array* is **Nothing.**
RankException	9	*Rank* < 1 or *Rank* is greater than the rank o f *Array*.

Remarks

Since array subscripts start at 0, the lowest available subscript for every dimension is always 0.

For an array with the following dimensions, **LBound** returns the values in the following table:

```
Dim A(100, 5, 4) As Byte
```

Call to LBound	Return value
`LBound(A, 1)`	0
`LBound(A, 2)`	0
`LBound(A, 3)`	0

Example

This example uses the **LBound** function to determine the lowest available subscript for the indicated dimension of an array.

```
Dim Lowest, MyArray(10, 15, 20), AnyArray(6) as Integer
Lowest = LBound(MyArray, 1)   ' Returns 0.
Lowest = LBound(MyArray, 3)   ' Returns 0.
Lowest = LBound(AnyArray)   ' Returns 0.
```

Requirements

Namespace: Microsoft.VisualBasic

Module: Information

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

UBound Function | Dim Statement | ReDim Statement | ArgumentException Class | RankException Class

LCase Function

Returns a string or character converted to lowercase.

```
Public Shared Function LCase(ByVal Value As Char) As Char
```

-or-

```
Public Shared Function LCase(ByVal Value As String) As String
```

Parameter

Value
Required. Any valid **String** or **Char** expression.

Remarks

Only uppercase letters are converted to lowercase; all lowercase letters and nonletter characters remain unchanged.

Example

This example uses the **LCase** function to return a lowercase version of a string.

```
Dim UpperCase, LowerCase As String
Uppercase = "Hello World 1234"   ' String to convert.
Lowercase = LCase(UpperCase)   ' Returns "hello world 1234".
```

Requirements

Namespace: Microsoft.VisualBasic

Module: Strings

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Strings | String Manipulation | UCase Function

Left Function

Returns a string containing a specified number of characters from the left side of a string.

```
Public Shared Function Left( _
   ByVal Str As String, _
   ByVal Length As Integer _
) As String
```

Parameters

Str
Required. **String** expression from which the leftmost characters are returned.

Length
Required. **Integer** expression. Numeric expression indicating how many characters to return. If 0, a zero-length string ("") is returned. If greater than or equal to the number of characters in *Str*, the entire string is returned.

Exceptions/Errors

Exception type	Error number	Condition
ArgumentException	5	*Length* < 0.

Remarks

To determine the number of characters in *Str*, use the **Len** function. If used in a Windows Form, or any other class that has a **Left** property, you must fully qualify the function with `Microsoft.VisualBasic.Left`.

> **Note** The **LeftB** function in previous versions of Visual Basic returns a string in bytes rather than characters. It is used primarily for converting strings in double-byte character set (DBCS) applications. All Visual Basic .NET strings are in Unicode, and **LeftB** is no longer supported.

Example

This example demonstrates the use of the **Left** function to return a substring of a given **String**. In a class that has a **Left** property, it may be necessary to fully qualify the **Left** function.

```
Dim myString As String = "Hello World!"
Dim subString As String
subString = Microsoft.VisualBasic.Left(myString, 5)   ' Returns "Hello"
```

Requirements

Namespace: Microsoft.VisualBasic

Module: Strings

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Right Function | Len Function | Mid Function | ArgumentException Class | Programming Element Support Changes Summary

Len Function

Returns an integer containing either the number of characters in a string or the number of bytes required to store a variable.

```
Public Shared Function Len( _
   ByVal Expression As { Boolean | Byte | Char | Double | Integer | Long |
   Object | Short | Single | String | DateTime | Decimal } _
) As Integer
```

Parameters

Expression
Any valid **String** expression or variable name. If *Expression* is of type **Object**, the **Len** function returns the size as it will be written to the file.

Remarks

With user-defined types and **Object** variables, the **Len** function returns the size as it will be written to the file. If an **Object** contains a **String**, it will return the length of the string. If an **Object** contains any other type, it will return the size of the object as it will be written to the file.

> **Note** The **Len** function may not be able to determine the actual number of storage bytes required when used with variable-length strings in user-defined data types.
>
> The **LenB** function in earlier versions of Visual Basic returns the number of bytes in a string rather than characters. It is used primarily for converting strings in double-byte character set (DBCS) applications. All Visual Basic .NET strings are in Unicode, and **LenB** is no longer supported.

Example

This example uses **Len** to return the number of characters in a string.

```
Dim MyString As String
Dim MyLen As Integer
MyString = "Hello World"   ' Initializes variable.
MyLen = Len(MyString)   ' Returns 11.
```

Smart Device Developer Notes

The **Len** function accepts only **String** and **Object** variables as parameters. If an **Object** contains a **String**, it will return the length of the **String**. If a parameter is a NULL **Object** reference, the function returns zero. If an **Object** contains any other type, an exception is thrown.

Requirements

Namespace: Microsoft.VisualBasic

Module: Strings

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Data Type Summary | InStr Function | Programming Element Support Changes Summary

LineInput Function

Reads a single line from an open sequential file and assigns it to a **String** variable.

```
Public Function LineInput(ByVal FileNumber As Integer) As String
```

Parameters

FileNumber
Required. Any valid file number.

Exceptions/Errors

Exception type	Error number	Condition
EndOfStreamException	62	End of file reached.
IOException	52	*FileNumber* does not exist.

Remarks

Data read with **LineInput** is usually written to a file with **Print**.

> **Security Note** When reading from files, do not make decisions about the contents of a file based on the file name extension. For example, a file named Form1.vb may not be a Visual Basic .NET source file.

The **LineInput** function reads from a file one character at a time until it encounters a carriage return (**Chr(**13**)**) or carriage return–linefeed (**Chr(**13**)** + **Chr(**10**)**) sequence. Carriage return–linefeed sequences are skipped rather than appended to the character string.

> **Security Note** Reading from a file with the **LineInput** function requires **Read** assess from the **FileIOPermissionAccess** enumeration. For more information, see FileIOPermissionAccess Enumeration.

Example

This example uses the **LineInput** function to read a line from a sequential file and assign it to a variable. This example assumes that `TESTFILE` is a text file with a few lines of sample data.

```
Dim TextLine As String
FileOpen(1, "TESTFILE", OpenMode.Input)   ' Open file.
While Not EOF(1)   ' Loop until end of file.
   TextLine = LineInput(1)   ' Read line into variable.
   Debug.WriteLine(TextLine)   ' Print to the console.
End While
FileClose(1)   ' Close file.
```

Smart Device Developer Notes

This function is not supported.

Requirements

Namespace: Microsoft.VisualBasic

Module: FileSystem

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Chr, ChrW Functions | Input Function | EndOfStreamException Class | IOException Class

Loc Function

Returns a **Long** value specifying the current read/write position within an open file.

```
Public Function Loc(ByVal FileNumber As Integer) As Long
```

Parameter

FileNumber
Required. Any valid **Integer** file number.

Exceptions/Errors

Exception type	Error number	Condition
IOException	52	*FileNumber* does not exist.
IOException	54	File mode is invalid.

Remarks

The following describes the return value for each file access mode:

Mode	Return value
Random	Number of the last record read from or written to the file.
Sequential	Current byte position in the file divided by 128. However, information returned by **Loc** for sequential files is neither used nor required.
Binary	Position of the last byte read or written.

Example

This example uses the **Loc** function to return the current read/write position within an open file. This example assumes that `MYFILE` is a text file with a few lines of sample data.

```
Dim location As Long
Dim oneLine As String
Dim oneChar As Char
FileOpen(1, "C:\MYFILE.TXT", OpenMode.Binary)
While location < EOF(1)
   Input(1, oneChar)
   location = Loc(1)
   Debug.WriteLine(location & ControlChars.CrLf)
End While
FileClose(1)
```

Smart Device Developer Notes

This function is not supported.

Requirements

Namespace: Microsoft.VisualBasic

Module: FileSystem

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

EOF Function | LOF Function | Seek Function | File Access with Visual Basic Run-Time Functions | IOException Class

Lock, Unlock Functions

Controls access by other processes to all or part of a file opened using the **Open** function.

```
Public Overloads Sub Lock(ByVal FileNumber As Integer)
. . .
Public Overloads Sub Unlock(ByVal FileNumber As Integer)
```

-or-

```
Public Overloads Sub Lock(_
   ByVal FileNumber As Integer, _
   ByVal FromRecord As Long _
)
. . .
Public Overloads Sub Unlock( _
   ByVal FileNumber As Integer, _
   ByVal FromRecord As Long _
)
```

-or-

```
Public Overloads Sub Lock( _
   ByVal FileNumber As Integer, _
   ByVal FromRecord As Long, _
   ByVal ToRecord As Long _
)
. . .
Public Overloads Sub Unlock( _
   ByVal FileNumber As Integer, _
   ByVal FromRecord As Long, _
   ByVal ToRecord As Long _
)
```

Parameters

FileNumber
Required. Any valid file number.
FromRecord
Optional. Number of the first record or byte to lock or unlock.
ToRecord
Optional. Number of the last record or byte to lock or unlock.

Exceptions/Errors

Exception type	Error number	Condition
IOException	52	*FileNumber* does not exist.
IOException	54	File mode is invalid.

Remarks

The **Lock** and **Unlock** functions are used in environments where several processes might need access to the same file.

Lock and **Unlock** functions are always used in pairs. The arguments to **Lock** and **Unlock** must match exactly.

If *FromRecord* and *ToRecord* are not supplied, the lock will be for the entire file. If *FromRecord* is specified but not *ToRecord*, the single record will be locked/unlocked.

If the file has been opened for sequential input or output, **Lock** and **Unlock** affect the entire file, regardless of the range specified by *FromRecord* and *ToRecord*.

Example

This example illustrates the use of the **Lock** and **Unlock** functions. This example assumes that `people.txt` is a file containing records of the structure `Person`.

```
Structure Person
   Dim Name As String
   Dim ID As Integer
End Structure

Sub PutInLockedFile(ByVal index As Integer, ByVal onePerson As Person)
   Try
      FileOpen(1, "c:\people.txt", OpenMode.Binary)
      Lock(1)
      FilePut(index, onePerson)
      Unlock(1)
      FileClose(1)
   Catch
      ' Error recovery code here.
   End Try
End Sub
```

Smart Device Developer Notes

This function is not supported.

Requirements

Namespace: Microsoft.VisualBasic

Module: FileSystem

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

FileOpen Function | IOException Class

LOF Function

Returns a **Long** representing the size, in bytes, of a file opened using the **FileOpen** function.

```
Public Function LOF(ByVal FileNumber As Integer) As Long
```

Parameter

FileNumber
Required. An **Integer** containing a valid file number.

Exceptions/Errors

Exception type	Error number	Condition
IOException	52	*FileNumber* does not exist.
IOException	54	File mode is invalid.

Remarks

Use the **FileLen** function to obtain the length of a file that is not open.

Example

This example uses the **LOF** function to determine the size of an open file. This example assumes that `TESTFILE` is a text file containing sample data.

```
Dim length As Integer
FileOpen(1, "C:\TESTFILE.TXT", OpenMode.Input) ' Open file.
length = LOF(1)   ' Get length of file.
Debug.WriteLine(length)
FileClose(1)   ' Close file.
```

Smart Device Developer Notes

This function is not supported.

Requirements

Namespace: Microsoft.VisualBasic

Module: FileSystem

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

EOF Function | FileLen Function | Loc Function | FileOpen Function | IOException Class

Math Functions

The math functions in Visual Basic 6 have been replaced by equivalent methods in the **System.Math** class of the .NET Framework.

Remarks

The .NET Framework math methods are functionally identical to their Visual Basic 6 counterparts, although some have slightly different names. For example, the .NET Framework equivalent of the Visual Basic 6 Atn function is **Atan**. The following table lists the Visual Basic 6 math function names and the equivalent .NET Framework methods.

Visual Basic 6 function	Visual Basic .NET method	Description
Abs	Math.Abs Method	Returns the absolute value of a specified number.
Atn	Math.Atan Method	Returns a **Double** value containing the angle whose tangent is the specified number.
Cos	Math.Cos Method	Returns a **Double** value containing the cosine of the specified angle.
Exp	Math.Exp Method	Returns a **Double** value containing e (the base of natural logarithms) raised to the specified power.
Log	Math.Log Method	Returns a **Double** value containing the logarithm of a specified number. This method is overloaded and can return either the natural (base e) logarithm of a specified number or the logarithm of a specified number in a specified base.
Round	Math.Round Method	Returns a **Double** value containing the number nearest the specified value. Additional round functions are available as methods of the intrinsic types such as Decimal.Round Method.
Sgn	Math.Sign Method	Returns an **Integer** value indicating the sign of a number.
Sin	Math.Sin Method	Returns a **Double** value specifying the sine of an angle.
Sqr	Math.Sqrt Method	Returns a **Double** value specifying the square root of a number.
Tan	Math.Tan Method	Returns a **Double** value containing the tangent of an angle.

In addition, the .NET Framework math class provides constants and other static methods for trigonometric, logarithmic, and other common mathematical functions. All of these can be used in a Visual Basic program.

To use these functions without qualification, import the **System.Math** namespace into your project by adding the following code to the top of the source code:

```
Imports System.Math
```

Requirements

Class: Math Class

Abs Example

This example uses the **Abs** method of the **Math** class to compute the absolute value of a number.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim MyNumber As Double
```

```
MyNumber = Abs(50.3)    ' Returns 50.3.
MyNumber = Abs(-50.3)   ' Returns 50.3.
```

Atan Example

This example uses the **Atan** method of the **Math** class to calculate the value of pi.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim pi As Double
pi = 4 * Atan(1)   ' Calculate the value of pi.
```

Cos Example

This example uses the **Cos** method of the **Math** class to return the cosine of an angle.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim MyAngle, MySecant As Double
MyAngle = 1.3   ' Define angle in radians.
MySecant = 1 / Cos(MyAngle)   ' Calculate secant.
```

Exp Example

This example uses the **Exp** method of the **Math** class to return *e* raised to a power.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim MyAngle, MyHSin As Double
' Define angle in radians.
MyAngle = 1.3
' Calculate hyperbolic sine.
MyHSin = (Exp(MyAngle) - Exp(-1 * MyAngle)) / 2
```

Log Example

This example uses the **Log** method of the **Math** class to return the natural logarithm of a number.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim MyAngle, MyLog As Double
' Define angle in radians.
MyAngle = 1.3
' Calculate inverse hyperbolic sine.
MyLog = Log(MyAngle + Sqrt(MyAngle * MyAngle + 1))
```

Round Example

This example uses the **Round** method of the **Math** class to round a number to the nearest integer.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim MyVar1 As Double = 2.8
Dim MyVar2 As Double
MyVar2 =Round(MyVar1)   ' Returns 3.
```

Sign Example

This example uses the **Sign** method of the **Math** class to determine the sign of a number.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim MyVar1, MyVar2, MyVar3 As Double
Dim MySign As Integer
MyVar1 = 12
MyVar2 = -2.4
MyVar3 = 0
MySign = Sign(MyVar1)   ' Returns 1.
MySign = Sign(MyVar2)   ' Returns -1.
MySign = Sign(MyVar3)   ' Returns 0.
```

Sin Example

This example uses the **Sin** method of the **Math** class to return the sine of an angle.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim MyAngle, MyCosecant As Double
MyAngle = 1.3   ' Define angle in radians.
MyCosecant = 1 / Sin(MyAngle)   ' Calculate cosecant.
```

Sqrt Example

This example uses the **Sqrt** method of the **Math** class to calculate the square root of a number.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim MySqr As Double
MySqr = Sqrt(4)    ' Returns 2.
MySqr = Sqrt(23)   ' Returns 4.79583152331272.
MySqr = Sqrt(0)    ' Returns 0.
MySqr = Sqrt(-4)   ' Returns NaN (not a number).
```

Tan Example

This example uses the **Tan** method of the **Math** class to return the tangent of an angle.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim MyAngle, MyCotangent As Double
MyAngle = 1.3   ' Define angle in radians.
MyCotangent = 1 / Tan(MyAngle)   ' Calculate cotangent.
```

Requirements

Namespace: System.Math

Assembly: mscorlib (in mscorlib.dll)

See Also

Rnd Function | Randomize Statement | Derived Math Functions | Double.NaN Field

LSet Function

Returns a left-aligned string containing the specified string adjusted to the specified length.

```
Public Shared Function LSet( _
   ByVal Source As String, _
   ByVal Length As Integer _
) As String
```

Parameters

Source
Required. **String** expression. Name of string variable.
Length
Required. **Integer** expression. Length of returned string.

Remarks

If the specified string is longer than the specified length, the returned string is shortened to the specified length. If the specified string is shorter than the specified length, spaces are added to the right end of the returned string to produce the appropriate length.

Example

This example demonstrates the use of the **LSet** function.

```
Dim myString As String = "Left"
Dim lString As String
LString = LSet(myString, 10)   ' Returns "Left      "
LString = LSet(myString, 2)    ' Returns "Le"
LString = LSet(myString, 4)    ' Returns "Left"
```

Requirements

Namespace: Microsoft.VisualBasic

Module: Strings

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Data Type Summary | RSet Function | Efficient Use of Data Types

LTrim, RTrim, and Trim Functions

Returns a string containing a copy of a specified string with no leading spaces (**LTrim**), no trailing spaces (**RTrim**), or no leading or trailing spaces (**Trim**).

```
Public Shared Function LTrim(ByVal Str As String) As String
Public Shared Function RTrim(ByVal Str As String) As String
Public Shared Function Trim(ByVal Str As String) As String
```

Parameter

Str
Required. Any valid **String** expression.

Example

This example uses the **LTrim** function to strip leading spaces and the **RTrim** function to strip trailing spaces from a string variable. It uses the **Trim** function to strip both types of spaces.

```
Dim MyString, TrimString As String
MyString = "  <-Trim->  "   ' Initializes string.
TrimString = LTrim(MyString)   ' TrimString = "<-Trim->  ".
TrimString = RTrim(MyString)   ' TrimString = "  <-Trim->".
TrimString = LTrim(RTrim(MyString))   ' TrimString = "<-Trim->".
' Using the Trim function alone achieves the same result.
TrimString = Trim(MyString)   ' TrimString = "<-Trim->".
```

Requirements

Namespace: Microsoft.VisualBasic

Module: Strings

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Left Function | Right Function

M-R Functions

The following table contains a listing of the Visual Basic .NET run-time member functions.

Mid	Minute	MIRR	MkDir
Month	MonthName	MsgBox	NPer
NPV	Oct	Partition	Pmt
PPmt	Print	PrintLine	PV
QBColor	Rate	Rename	Replace
Reset	RGB	Right	RmDir
Rnd	Round	RSet	RTrim

See Also

A-C Functions | D-G Functions | H-L Functions | S-Z Functions | Visual Basic Language and Run-Time Reference

Mid Function

Returns a string containing a specified number of characters from a string.

```
Public Shared Function Mid( _
   ByVal Str As String, _
   ByVal Start As Integer, _
   Optional ByVal Length As Integer _
) As String
```

Parameters

Str
Required. **String** expression from which characters are returned.

Start
Required. **Integer** expression. Character position in *Str* at which the part to be taken starts. If *Start* is greater than the number of characters in *Str*, the **Mid** function returns a zero-length string (""). *Start* is one based.

Length
Optional. **Integer** expression. Number of characters to return. If omitted or if there are fewer than *Length* characters in the text (including the character at position *Start*), all characters from the start position to the end of the string are returned.

Exceptions/Errors

Exception type	Error number	Condition
ArgumentException	5	*Start* <= 0 or *Length* < 0 .

Remarks

To determine the number of characters in *Str*, use the **Len** function.

> **Note** The **MidB** function in previous versions of Visual Basic returns a string in bytes rather than characters. It is used primarily for converting strings in double-byte character set (DBCS) applications. All Visual Basic .NET strings are in Unicode, and **MidB** is no longer supported.

Example

This example uses the **Mid** function to return a specified number of characters from a string.

```
Dim MyString, FirstWord, LastWord, MidWords As String
MyString = "Mid Function Demo"   ' Creates text string.
FirstWord = Mid(MyString, 1, 3)   ' Returns "Mid".
LastWord = Mid(MyString, 14, 4)   ' Returns "Demo".
MidWords = Mid(MyString, 5)   ' Returns "Function Demo".
```

Requirements

Namespace: Microsoft.VisualBasic

Module: Strings

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Strings | String Manipulation | Left Function | Len Function | LTrim, RTrim, and Trim Functions | Mid Statement | Right Function | ArgumentException Class | Programming Element Support Changes Summary

Minute Function

Returns an **Integer** value from 0 through 59 representing the minute of the hour.

```
Public Function Minute(ByVal TimeValue As DateTime) As Integer
```

Parameter

TimeValue
Required. **Date** value from which you want to extract the minute.

You can also obtain the minute of the hour by calling **DatePart** and specifying **DateInterval.Minute** for the *Interval* argument.

Example

This example uses the **Minute** function to obtain the minute of the hour from a specified time. In the development environment, the time literal is displayed in short time format using the locale settings of your code.

```
Dim MyTime As Date
Dim MyMinute As Integer
MyTime = #4:35:17 PM#   ' Assign a time.
MyMinute = Minute(MyTime)   ' MyMinute contains 35.
```

Requirements

Namespace: Microsoft.VisualBasic

Module: DateAndTime

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Day Function | Hour Function | Now Property | Second Function | TimeOfDay Property | DatePart Function System Namespace | DateTime Structure | ArgumentException Class | ArgumentOutOfRangeException Class

MIRR Function

Returns a **Double** specifying the modified internal rate of return for a series of periodic cash flows (payments and receipts).

```
Function MIRR( _
   ByRef ValueArray() As Double, _
   ByVal FinanceRate As Double, _
   ByVal ReinvestRate As Double _
) As Double
```

Parameters

ValueArray()
Required. Array of **Double** specifying cash flow values. The array must contain at least one negative value (a payment) and one positive value (a receipt).

FinanceRate
Required. **Double** specifying interest rate paid as the cost of financing.

ReinvestRate
Required. **Double** specifying interest rate received on gains from cash reinvestment.

Exceptions/Errors

Exception type	Error number	Condition
ArgumentException	5	Rank of *ValueArray* does not equal 1 .
ArgumentException	5	*FinanceRate* = -1
ArgumentException	5	*ReinvestRate* = -1
DivideByZeroException	11	Division by zero has occurred.

Remarks

The modified internal rate of return is the internal rate of return when payments and receipts are financed at different rates. The **MIRR** function takes into account both the cost of the investment (*FinanceRate*) and the interest rate received on reinvestment of cash (*ReinvestRate*).

The *FinanceRate* and *ReinvestRatearguments* are percentages expressed as decimal values. For example, 12 percent is expressed as 0.12.

The **MIRR** function uses the order of values within the array to interpret the order of payments and receipts. Be sure to enter your payment and receipt values in the correct sequence.

Example

This example uses the **MIRR** function to return the modified internal rate of return for a series of cash flows contained in the array `Values()`. `LoanAPR` represents the financing interest, and `InvAPR` represents the interest rate received on reinvestment.

```
Sub TestMIRR()
   Dim LoanAPR, InvAPR, RetRate, Values(4) As Double
   Dim Msg, Fmt As String
   LoanAPR = 0.1  ' Loan rate.
   InvAPR = 0.12  ' Reinvestment rate.
   Fmt = "#0.00"   ' Define money format.
   Values(0) = -70000   ' Business start-up costs.
   ' Positive cash flows reflecting income for four successive years.
   Values(1) = 22000 : Values(2) = 25000
   Values(3) = 28000 : Values(4) = 31000
   RetRate = MIRR(Values, LoanAPR, InvAPR)   ' Calculate internal rate.
   Msg = "The modified internal rate of return for these five cash flows is "
   Msg = Msg & Format(Math.Abs(RetRate) * 100, CStr(Fmt)) & "%."
   MsgBox(Msg)    ' Display internal return rate.
End Sub
```

Requirements

Namespace: Microsoft.VisualBasic

Module: Financial

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

DDB Function | FV Function | IPmt Function | IRR Function | NPer Function | NPV Function | Pmt Function | PPmt Function | PV Function | Rate Function | SLN Function | SYD Function | ArgumentException Class | DivideByZeroException Class

MkDir Function

Creates a new directory or folder.

```
Public Sub MkDir(ByVal Path As String)
```

Parameter

Path
Required. **String** expression that identifies the directory or folder to be created. The *Path* may include the drive. If no drive is specified, **MkDir** creates the new directory or folder on the current drive.

Exceptions/Errors

Exception type	Error number	Condition
ArgumentException	52	*Path* is not specified or is empty .
IOException	75	Directory already exists.

Example

This example uses the **MkDir** function to create a directory or folder. If the drive is not specified, the new directory or folder is created on the current drive.

```
MkDir("C:\MYDIR")   ' Make new directory or folder.
```

Smart Device Developer Notes

This function is not supported.

Requirements

Namespace: Microsoft.VisualBasic

Module: FileSystem

Assembly: Microsoft Visual Basic .NET Runtime (in MicrosoftVisualBasic.dll)

See Also

ChDir Function | CurDir Function | RmDir Function | ArgumentException Class| IOException Class

Month Function

Returns an **Integer** value from 1 through 12 representing the month of the year.

```
Public Function Month(ByVal DateValue As DateTime) As Integer
```

Parameter

DateValue
Required. **Date** value from which you want to extract the month.

You can also obtain the month of the year by calling **DatePart** and specifying **DateInterval.Month** for the *Interval* argument.

Example

This example uses the **Month** function to obtain the month from a specified date. In the development environment, the date literal is displayed in short date format using the locale settings of your code.

```
Dim MyDate As Date
Dim MyMonth As Integer
MyDate = #2/12/1969#   ' Assign a date.
MyMonth = Month(MyDate)   ' MyMonth contains 2.
```

Requirements

Namespace: Microsoft.VisualBasic

Module: DateAndTime

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Day Function | Now Property | Weekday Function | Year Function | DatePart Function | System Namespace | DateTime Structure | ArgumentException Class | ArgumentOutOfRangeException Class

MonthName Function

Returns a **String** value containing the name of the specified month.

```
Public Function MonthName( _
   ByVal Month As Integer, _
   Optional ByVal Abbreviate As Boolean = False _
) As String
```

Parameters

Month
Required. **Integer**. The numeric designation of the month, from 1 through 13; 1 indicates January and 12 indicates December. You can use the value 13 with a 13-month calendar. If your system is using a 12-month calendar and *Month* is 13, **MonthName** returns an empty string.

Abbreviate
Optional. **Boolean** value that indicates if the month name is to be abbreviated. If omitted, the default is **False**, which means the month name is not abbreviated.

Exceptions/Errors

Exception type	Error number	Condition
ArgumentException	5	*Month* is less than 1 or greater than 13 .

Remarks

The string returned by **MonthName** depends not only on the input arguments, but also on the Regional Options settings specified in the Windows Control Panel.

If *Month* is less than 1 or more than 13, an **ArgumentException** error occurs.

> **Note** **MonthName** uses the current calendar setting from the **CurrentCulture** property of the **CultureInfo** class in the **System.Globalization** namespace. The default **CurrentCulture** values are determined by **Control Panel** settings.

Example

This example uses the **MonthName** function to determine the name of the month, by the integer given. The Boolean value will determine whether the full name (**False**) or the abbreviated name (**True**) will be displayed.

```
Dim MyMonth As Integer
Dim Name As String
MyMonth = 4
Name = MonthName(MyMonth, True)   ' "True" returns an abbreviated name.
Msgbox(Name)   ' Name contains "Apr".
```

Requirements

Namespace: Microsoft.VisualBasic

Module: DateAndTime

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

WeekdayName Function | System Namespace | System.Globalization Namespace | DateTime Structure | ArgumentException Class | CultureInfo Class

MsgBox Function

Displays a message in a dialog box, waits for the user to click a button, and then returns an integer indicating which button the user clicked.

```
Public Function MsgBox( _
   ByVal Prompt As Object, _
   Optional ByVal Buttons As MsgBoxStyle = MsgBoxStyle.OKOnly, _
   Optional ByVal Title As Object = Nothing _
) As MsgBoxResult
```

Parameters

Prompt
Required. **String** expression displayed as the message in the dialog box. The maximum length of *Prompt* is approximately 1024 characters, depending on the width of the characters used. If *Prompt* consists of more than one line, you can separate the lines using a carriage return character (**Chr(**13**)**), a linefeed character (**Chr(**10**)**), or a carriage return/linefeed character combination (**Chr(**13**)** & **Chr(**10**)**) between each line.

Buttons
Optional. Numeric expression that is the sum of values specifying the number and type of buttons to display, the icon style to use, the identity of the default button, and the modality of the message box. If you omit *Buttons*, the default value is zero.

Title
Optional. **String** expression displayed in the title bar of the dialog box. If you omit *Title*, the application name is placed in the title bar.

Settings

The **MsgBoxStyle** enumeration values are listed in the following table.

Enumeration	Value	Description
OKOnly	0	Displays OK button only.
OKCancel	1	Displays OK and Cancel buttons.
AbortRetryIgnore	2	Displays Abort, Retry, and Ignore buttons.
YesNoCancel	3	Displays Yes, No, and Cancel buttons.
YesNo	4	Displays Yes and No buttons.
RetryCancel	5	Displays Retry and Cancel buttons.
Critical	16	Displays Critical Message icon.
Question	32	Displays Warning Query icon.
Exclamation	48	Displays Warning Message icon.
Information	64	Displays Information Message icon.
DefaultButton1	0	First button is default.
DefaultButton2	256	Second button is default.
DefaultButton3	512	Third button is default.
ApplicationModal	0	Application is modal. The user must respond to the message box b efore continuing work in the current application.
SystemModal	4096	System is modal. All applications are suspended until the user resp onds to the message box.
MsgBoxSetForeground	65536	Specifies the message box window as the foreground window.
MsgBoxRight	524288	Text is right-aligned.
MsgBoxRtlReading	1048576	Specifies text should appear as right-to-left reading on Hebrew an d Arabic systems.

The first group of values (0–5) describes the number and type of buttons displayed in the dialog box; the second group (16, 32, 48, 64) describes the icon style; the third group (0, 256, 512) determines which button is the default; the fourth group (0, 4096) determines the modality of the message box, and the fifth group specifies whether or not the message box window is the foreground window, along with the alignment and direction of the text. When adding numbers to create a final value for the *Buttons* argument, use only one number from each group.

Return Values

Constant	Value
OK	1
Cancel	2
Abort	3
Retry	4
Ignore	5
Yes	6
No	7

Exceptions/Errors

Exception type	Error number	Condition
ArgumentException	5	*Prompt* is not a **String** expression, or *Title* is invalid.
InvalidOperationException	5	Process is not running in User Interactive mode.
InvalidEnumArgumentException	5	One or more parameters not a member of **MsgBoxResult** or **MsgBoxStyle** enumerations.

Remarks

If the dialog box displays a **Cancel** button, pressing the ESC key has the same effect as clicking **Cancel**. If the dialog box contains a **Help** button, context-sensitive Help is provided for the dialog box. However, no value is returned until one of the other buttons is clicked.

> **Note** To specify more than the first argument, you must use the **MsgBox** function in an expression. If you omit any positional arguments, you must retain the corresponding comma delimiter.
>
> **Note** The **MsgBox** function requires UIPermission at the SafeTopLevelWindows level, which may affect its execution in partial trust situations. For more information, see Requesting Permissions and UIPermission Class.

Example

This example uses the **MsgBox** function to display a critical-error message in a dialog box with Yes and No buttons. The No button is specified as the default response. This is done by combining the **MsgBox** constant values into one numeric expression. In this case, adding 4 (the Yes/No button combination) and 16 (the Critical Message window) and 256 (the second button as default button) gives a total of 276. The value returned by the **MsgBox** function depends on the button chosen by the user: Yes returns a value of 6; No returns a value of 7.

```
Dim msg As String
Dim title As String
Dim style As MsgBoxStyle
Dim response As MsgBoxResult
msg = "Do you want to continue?"   ' Define message.
style = MsgBoxStyle.DefaultButton2 Or _
   MsgBoxStyle.Critical Or MsgBoxStyle.YesNo
title = "MsgBox Demonstration"   ' Define title.
' Display message.
response = MsgBox(msg, style, title)
If response = MsgBoxResult.Yes Then   ' User chose Yes.
   ' Perform some action.
Else
   ' Perform some other action.
End If
```

Requirements

Namespace: Microsoft.VisualBasic

Module: Interaction

Assembly: Microsoft.Visual Basic .NET Runtime (in MicrosoftVisualBasic.dll)

See Also

InputBox Function

NPer Function

Returns a **Double** specifying the number of periods for an annuity based on periodic, fixed payments and a fixed interest rate.

```
Function NPer( _
    ByVal Rate As Double, _
   ByVal Pmt As Double, _
   ByVal PV As Double, _
   Optional ByVal FV As Double = 0, _
   Optional ByVal Due As DueDate = DueDate.EndOfPeriod _
) As Double
```

Parameters

Rate
Required. **Double** specifying interest rate per period. For example, if you get a car loan at an annual percentage rate (APR) of 10 percent and make monthly payments, the rate per period is 0.1/12, or 0.0083.

Pmt
Required. **Double** specifying payment to be made each period. Payments usually contain principal and interest that doesn't change over the life of the annuity.

PV
Required. **Double** specifying present value, or value today, of a series of future payments or receipts. For example, when you borrow money to buy a car, the loan amount is the present value to the lender of the monthly car payments you will make.

FV
Optional. **Double** specifying future value or cash balance you want after you've made the final payment. For example, the future value of a loan is $0 because that's its value after the final payment. However, if you want to save $50,000 over 18 years for your child's education, then $50,000 is the future value. If omitted, 0 is assumed.

Due
Optional. Object of type `Microsoft.VisualBasic.DueDate` that specifies when payments are due. This argument must be either `DueDate.EndOfPeriod` if payments are due at the end of the payment period, or `DueDate.BegOfPeriod` if payments are due at the beginning of the period. If omitted, `DueDate.EndOfPeriod` is assumed.

Exceptions/Errors

Exception type	Error number	Condition
ArgumentException	5	*Rate* <= -1.
ArgumentException	5	*Rate* = 0 and *Pmt* = 0

Remarks

An annuity is a series of fixed cash payments made over a period of time. An annuity can be a loan (such as a home mortgage) or an investment (such as a monthly savings plan).

For all arguments, cash paid out (such as deposits to savings) is represented by negative numbers; cash received (such as dividend checks) is represented by positive numbers.

Example

This example uses the **NPer** function to return the number of periods during which payments must be made to pay off a loan whose value is contained in `PVal`. Also provided are the interest percentage rate per period (`APR / 12`), the payment (`Payment`), the future value of the loan (`FVal`), and a number that indicates whether the payment is due at the beginning or end of the payment period (`PayType`).

```
Sub TestNPer()
   Dim FVal, TotPmts As Double
   Dim PVal, APR, Payment As Double
   Dim PayType As DueDate
   Dim Response As MsgBoxResult
   FVal = 0   ' Usually 0 for a loan.
   PVal = CDbl(InputBox("How much do you want to borrow?"))
```

```
    APR = CDbl(InputBox("What is the annual percentage rate of your loan?"))
    If APR > 1 Then APR = APR / 100 ' Ensure proper form.
    Payment = CDbl(InputBox("How much do you want to pay each month?"))
    Response = MsgBox("Do you make payments at the end of month?", MsgBoxStyle.YesNo)
    If Response = MsgBoxResult.No Then
       PayType = DueDate.BegOfPeriod
    Else
       PayType = DueDate.EndOfPeriod
    End If
    TotPmts = NPer(APR / 12, -Payment, PVal, FVal, PayType)
    If Int(TotPmts) <> TotPmts Then TotPmts = Int(TotPmts) + 1
       MsgBox("It will take you " & TotPmts & " months to pay off your loan.")
End Sub
```

Requirements

Namespace: Microsoft.VisualBasic

Module: Financial

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

DDB Function | FV Function | IPmt Function | IRR Function | MIRR Function | NPV Function | Pmt Function | PPmt Function | PV Function | Rate Function | SLN Function | SYD Function | ArgumentException Class

NPV Function

Returns a **Double** specifying the net present value of an investment based on a series of periodic cash flows (payments and receipts) and a discount rate.

```
Function NPV( _
   ByVal Rate As Double, _
   ByRef ValueArray() As Double _
) As Double
```

Parameters

Rate
Required. **Double** specifying discount rate over the length of the period, expressed as a decimal.

ValueArray()
Required. Array of **Double** specifying cash flow values. The array must contain at least one negative value (a payment) and one positive value (a receipt).

Exceptions/Errors

Exception type	Error number	Condition
ArgumentException	5	*ValueArray* is **Nothing,** rank of *ValueArray* <> 1, or *Rate* = -1

Remarks

The net present value of an investment is the current value of a future series of payments and receipts.

The **NPV** function uses the order of values within the array to interpret the order of payments and receipts. Be sure to enter your payment and receipt values in the correct sequence.

The **NPV** investment begins one period before the date of the first cash flow value and ends with the last cash flow value in the array.

The net present value calculation is based on future cash flows. If your first cash flow occurs at the beginning of the first period, the first value must be added to the value returned by **NPV** and must not be included in the cash flow values of *ValueArray()*.

The **NPV** function is similar to the **PV** function (present value) except that the **PV** function allows cash flows to begin either at the end or the beginning of a period. Unlike the variable **NPV** cash flow values, **PV** cash flows must be fixed throughout the investment.

Example

This example uses the **NPV** function to return the net present value for a series of cash flows contained in the array `Values()`. `RetRate` represents the fixed internal rate of return.

```
Sub TestNPV()
   Dim Guess, RetRate, NetPVal, values(4) As Double
   Dim Fmt, Msg As String
   Fmt = "###,##0.00"   ' Define money format.
   Guess = 0.1  ' Guess starts at 10 percent.
   RetRate = 0.0625  ' Set fixed internal rate.
   values(0) = -70000   ' Business start-up costs.
   ' Positive cash flows reflecting income for four successive years.
   values(1) = 22000 : values(2) = 25000
   values(3) = 28000 : values(4) = 31000
   NetPVal = NPV(RetRate, values)   ' Calculate net present value.
   Msg = "The net present value of these cash flows is "
   Msg = Msg & Format(NetPVal, Fmt) & "."
   MsgBox(Msg)    ' Display net present value.
End Sub
```

Requirements

Namespace: Microsoft.VisualBasic

Module: Financial

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

DDB Function | FV Function | IPmt Function | IRR Function | MIRR Function | NPer Function | Pmt Function | PPmt Function | PV Function | Rate Function | SLN Function | SYD Function | ArgumentException Class

Oct Function

Returns a string representing the octal value of a number.

```
Public Shared Function Oct( _
   ByVal Number As { Byte | Short | Integer | Long | Object } _
) As String
```

Parameter

Number
Required. Any valid numeric expression or **String** expression.

Exceptions/Errors

Exception type	Error number	Condition
ArgumentNullException	5	*Number* is not specified.
ArgumentException	5	*Number* is not a numeric type.

Remarks

If *Number* is not already a whole number, it is rounded to the nearest whole number before being evaluated.

If *Number* is	Oct returns
Empty	Zero (0)
Any other number	Up to 22 octal characters

You can represent octal numbers directly by preceding numbers in the proper range with **&O**. For example, `&O10` is the octal notation for decimal 8.

Example

This example uses the **Oct** function to return the octal value of a number.

```
Dim MyOct As String
MyOct = Oct(4)    ' Returns 4.
MyOct = Oct(8)   ' Returns 10.
MyOct = Oct(459)   ' Returns 713.
```

Requirements

Namespace: Microsoft.VisualBasic

Module: Conversion

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Hex Function | Type Conversion Functions | ArgumentNullException Class

Partition Function

Returns a string representing the calculated range that contains a number.

```
Public Function Partition( _
   ByVal Number As Long, _
   ByVal Start As Long, _
   ByVal Stop As Long, _
   ByVal Interval As Long _
) As String
```

Parameters

Number
Required. **Long**. Whole number that you want to locate within one of the calculated ranges.
Start
Required. **Long**. Whole number that indicates the start of the set of calculated ranges. *Start* cannot be less than 0.
Stop
Required. **Long**. Whole number that indicates the end of the set of calculated ranges. *Stop* cannot be less than or equal to *Start*.
Interval
Required. **Long**. Whole number that indicates the size of each range calculated between *Start* and *Stop*. *Interval* cannot be less than 1.

Exceptions/Errors

Exception type	Error number	Condition
ArgumentException	5	*Start* < 0, *Stop* <= *Start*, or *Interval* < 1 .

Remarks

The **Partition** function calculates a set of numeric ranges, each containing the number of values specified by *Interval*. The first range begins at *Start*, and the last range ends at *Stop*. The **Partition** function then identifies which range contains *Number* and returns a string describing that range. The range is represented in the string as "*lowervalue*:*uppervalue*", where the low end of the range (*lowervalue*) is separated from the high end (*uppervalue*) by a colon (:).

If necessary, the **Partition** function inserts leading spaces before *lowervalue* and *uppervalue* so that they both have the same number of characters as the string representation of the value (*Stop* + 1). This ensures that if you use the output of the **Partition** function with several values of *Number*, the resulting text will be handled properly during any subsequent sort operation.

The following table shows some sample strings for ranges calculated using three sets of *Start*, *Stop*, and *Interval*. The "First range" and "Last range" columns show the lowest and highest ranges possible given the values of *Start* and *Stop*. The "Before first range" and "After last range" columns show the strings returned for values of *Number* less than *Start* and greater than *Stop*, respectively.

Start	***Stop***	***Interval***	**Before first range**	**First range**	**Last range**	**After last range**
0	99	5	" : -1"	" 0: 4"	" 95: 99"	"100: "
20	199	10	" : 19"	" 20: 29"	"190:199"	"200: "
100	1010	20	" : 99"	" 100: 119"	"1000:1010"	"1011: "

In the preceding table, the third line shows the result when *Start* and *Stop* define a set of numbers that cannot be evenly divided by *Interval*. The last range ends at *Stop*, making it only 11 numbers long, even though *Interval* is 20.

If *Interval* is 1, the range is "*Number*:*Number*", regardless of the *Start* and *Stop* arguments. For example, if *Number* is 267, *Stop* is 1000, and *Interval* is 1, **Partition** returns " 267: 267".

Partition is useful in database queries. You can create a select query that shows how many orders occur within various value ranges, for example with invoice values from 1 to 1000, 1001 to 2000, and so on.

Example

This example assumes you have an **Orders** table that contains a `Freight` field. It creates a SELECT procedure that counts the number of orders for which freight cost is within each of several ranges. The **Partition** function first establishes these ranges, and

then the SQL **Count** function counts the number of orders in each range. In this example, the arguments to the **Partition** function are *Start* = 0, *Stop* = 500, *Interval* = 50. The first range is therefore 0:49, and so on up to 450:499 and 500:500.

```
SELECT DISTINCTROW Partition([Freight],0, 500, 50) As Range,
Count(Orders.Freight) As Count
FROM Orders
GROUP BY Partition([Freight],0,500,50);
```

Requirements

Namespace: Microsoft.VisualBasic

Module: Interaction

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

SQL

Pmt Function

Returns a **Double** specifying the payment for an annuity based on periodic, fixed payments and a fixed interest rate.

```
Function Pmt( _
   ByVal Rate As Double, _
   ByVal NPer As Double, _
   ByVal PV As Double, _
   Optional ByVal FV As Double = 0, _
   Optional ByVal Due As DueDate = DueDate.EndOfPeriod _
) As Double
```

Parameters

Rate
Required. **Double** specifying interest rate per period. For example, if you get a car loan at an annual percentage rate (APR) of 10 percent and make monthly payments, the rate per period is 0.1/12, or 0.0083.

NPer
Required. **Double** specifying total number of payment periods in the annuity. For example, if you make monthly payments on a four-year car loan, your loan has a total of 4 * 12 (or 48) payment periods.

PV
Required. **Double** specifying present value (or lump sum) that a series of payments to be paid in the future is worth now. For example, when you borrow money to buy a car, the loan amount is the present value to the lender of the monthly car payments you will make.

FV
Optional. **Double** specifying future value or cash balance you want after you've made the final payment. For example, the future value of a loan is $0 because that's its value after the final payment. However, if you want to save $50,000 over 18 years for your child's education, then $50,000 is the future value. If omitted, 0 is assumed.

Due
Optional. Object of type `Microsoft.VisualBasic.DueDate` that specifies when payments are due. This argument must be either `DueDate.EndOfPeriod` if payments are due at the end of the payment period, or `DueDate.BegOfPeriod` if payments are due at the beginning of the period. If omitted, `DueDate.EndOfPeriod` is assumed.

Exceptions/Errors

Exception type	Error number	Condition
ArgumentException	5	*NPer* = 0.

Remarks

An annuity is a series of fixed cash payments made over a period of time. An annuity can be a loan (such as a home mortgage) or an investment (such as a monthly savings plan).

The *Rate* and *NPer* arguments must be calculated using payment periods expressed in the same units. For example, if *Rate* is calculated using months, *NPer* must also be calculated using months.

For all arguments, cash paid out (such as deposits to savings) is represented by negative numbers; cash received (such as dividend checks) is represented by positive numbers.

Example

This example uses the **Pmt** function to return the monthly payment for a loan over a fixed period. Given are the interest percentage rate per period (`APR / 12`), the total number of payments (`TotPmts`), the present value or principal of the loan (`PVal`), the future value of the loan (`FVal`), and a number that indicates whether the payment is due at the beginning or end of the payment period `(PayType)`.

```
Sub TestPMT()
   Dim PVal, APR, FVal, Payment, TotPmts As Double
   Dim PayType As DueDate
   Dim Fmt As String
   Dim Response As MsgBoxResult
```

```
    Fmt = "###,###,##0.00"   ' Define money format.
    FVal = 0   ' Usually 0 for a loan.
    PVal = CDbl(InputBox("How much do you want to borrow?"))
    APR = CDbl(InputBox("What is the annual percentage rate of your loan?"))
    If APR > 1 Then APR = APR / 100 ' Ensure proper form.
    TotPmts = CDbl(InputBox("How many monthly payments will you make?"))
    Response = MsgBox("Do you make payments at the end of month?", MsgBoxStyle.YesNo)
    If Response = MsgBoxResult.No Then
       PayType = DueDate.BegOfPeriod
    Else
       PayType = DueDate.BegOfPeriod
    End If
    Payment = Pmt(APR / 12, TotPmts, -PVal, FVal, PayType)
    MsgBox("Your payment will be " & Format(Payment, Fmt) & " per month.")
 End Sub
```

Requirements

Namespace: Microsoft.VisualBasic

Module: Financial

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

DDB Function | FV Function | IPmt Function | IRR Function | MIRR Function | NPer Function | NPV Function | PPmt Function | PV Function | Rate Function | SLN Function | SYD Function | ArgumentException Class

PPmt Function

Returns a **Double** specifying the principal payment for a given period of an annuity based on periodic, fixed payments and a fixed interest rate.

```
Function PPmt( _
    ByVal Rate As Double, _
   ByVal Per As Double, _
   ByVal NPer As Double, _
   ByVal PV As Double, _
   Optional ByVal FV As Double = 0, _
   Optional ByVal Due As DueDate = DueDate.EndOfPeriod _
) As Double
```

Parameters

Rate
Required. **Double** specifying interest rate per period. For example, if you get a car loan at an annual percentage rate (APR) of 10 percent and make monthly payments, the rate per period is 0.1/12, or 0.0083.

Per
Required. **Double** specifying payment period in the range 1 through *NPer*.

NPer
Required. **Double** specifying total number of payment periods in the annuity. For example, if you make monthly payments on a four-year car loan, your loan has a total of 4 * 12 (or 48) payment periods.

PV
Required. **Double** specifying present value, or value today, of a series of future payments or receipts. For example, when you borrow money to buy a car, the loan amount is the present value to the lender of the monthly car payments you will make.

FV
Optional. **Double** specifying future value or cash balance you want after you've made the final payment. For example, the future value of a loan is $0 because that's its value after the final payment. However, if you want to save $50,000 over 18 years for your child's education, then $50,000 is the future value. If omitted, 0 is assumed.

Due
Optional. Object of type `Microsoft.VisualBasic.DueDate` that specifies when payments are due. This argument must be either `DueDate.EndOfPeriod` if payments are due at the end of the payment period, or `DueDate.BegOfPeriod` if payments are due at the beginning of the period. If omitted, `DueDate.EndOfPeriod` is assumed.

Exceptions/Errors

Exception type	Error number	Condition
ArgumentException	5	*Per* value invalid: must be > 1 and < *NPer* .

Remarks

An annuity is a series of fixed cash payments made over a period of time. An annuity can be a loan (such as a home mortgage) or an investment (such as a monthly savings plan).

The *Rate* and *NPer* arguments must be calculated using payment periods expressed in the same units. For example, if *Rate* is calculated using months, *NPer* must also be calculated using months.

For all arguments, cash paid out (such as deposits to savings) is represented by negative numbers; cash received (such as dividend checks) is represented by positive numbers.

Example

This example uses the **PPmt** function to calculate how much of a payment for a specific period is principal when all the payments are of equal value. Given are the interest percentage rate per period (`APR / 12`), the payment period for which the principal portion is desired (`Period`), the total number of payments (`TotPmts`), the present value or principal of the loan (`PVal`), the future value of the loan (`FVal`), and a number that indicates whether the payment is due at the beginning or end of the payment period (`PayType`).

```
Sub TestPPMT()
   Dim PVal, APR, TotPmts, MakeChart, FVal, Payment, Period, P, I As Double
   Dim PayType As DueDate
   Dim Msg, Fmt As String
   Dim Response As MsgBoxResult

   Fmt = "###,###,##0.00"   ' Define money format.
   FVal = 0   ' Usually 0 for a loan.
   PVal = CDbl(InputBox("How much do you want to borrow?"))
   APR = CDbl(InputBox("What is the annual percentage rate of your loan?"))
   If APR > 1 Then APR = APR / 100 ' Ensure proper form.
   TotPmts = CDbl(InputBox("How many monthly payments do you have to make?"))
   Response = MsgBox("Do you make payments at the end of month?", MsgBoxStyle.YesNo)
   If Response = MsgBoxResult.No Then
      PayType = DueDate.BegOfPeriod
   Else
      PayType = DueDate.EndOfPeriod
   End If
   Payment = Math.Abs(-Pmt(APR / 12, TotPmts, PVal, FVal, PayType))
   Msg = "Your monthly payment is " & Format(Payment, Fmt) & ". "
   Msg = Msg & "Would you like a breakdown of your principal and "
   Msg = Msg & "interest per period?"
   Response = MsgBox(Msg, MsgBoxStyle.YesNo)   ' See if chart is desired.
   If Response <> MsgBoxResult.No Then
      If TotPmts > 12 Then MsgBox("Only first year will be shown.")
      Msg = "Month  Payment  Principal  Interest" & vbNewLine
      For Period = 1 To TotPmts
         If Period > 12 Then Exit For ' Show only first 12.
         P = PPmt(APR / 12, Period, TotPmts, -PVal, FVal, PayType)
         P = (Int((P + 0.005) * 100) / 100)  ' Round principal.
         I = Payment - P
         I = (Int((I + 0.005) * 100) / 100)  ' Round interest.
         Msg = Msg & Period & vbTab & Format(Payment, Fmt)
         Msg = Msg & vbTab & Format(P, Fmt) & vbTab & Format(I, Fmt) & vbNewLine
      Next Period
      MsgBox(Msg)    ' Display amortization table.
   End If
End Sub
```

Requirements

Namespace: Microsoft.VisualBasic

Module: Financial

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

DDB Function | FV Function | IPmt Function | IRR Function | MIRR Function | NPer Function | NPV Function | Pmt Function | PV Function | Rate Function | SLN Function | SYD Function | ArgumentException Class

Print, PrintLine Functions

Writes display-formatted data to a sequential file.

```
Public Sub Print( _
   ByVal FileNumber As Integer, _
   ByVal ParamArray Output() As Object _
)
```

-or-

```
Public Sub PrintLine( _
   ByVal FileNumber As Integer, _
   ByVal ParamArray Output() As Object _
)
```

Parameters

FileNumber
Required. Any valid file number.
Output
Optional. Zero or more comma-delimited expressions to write to a file.

Output argument settings

The *Output* argument settings are:

Setting	Description
SPC(*n*)	Used to insert space characters in the output, where *n* is the number of space characters to insert.
TAB(*n*)	Used to position the insertion point to an absolute column number, where *n* is the column number. Use **TAB** with no argument to position the insertion point at the beginning of the next print zone.
expression	Numeric expressions or string expressions to print.

Exceptions/Errors

Exception type	Error number	Condition
IOException	54	File mode is invalid.
IOException	52	*FileNumber* does not exist.

Remarks

Print will not include a linefeed at the end of a line; **PrintLine,** however, will include a linefeed.

Data written with **Print** is usually read from a file with **LineInput** or **Input**.

If you omit *Output* for **PrintLine**, a blank line is printed to the file; for **Print,** nothing is output. Multiple expressions separated with a comma will be aligned on tab boundaries, but mixing commas and **TAB** may result in inconsistent results.

For **Boolean** data, either `True` or `False` is printed. The **True** and **False** keywords are not translated, regardless of the locale.

Date data is written to the file using the standard short date format recognized by your system. When either the date or the time component is missing or zero, only the part provided is written to the file.

Nothing is written to the file if *Output* data is empty. However, if *Output* list data is **DBNull**, `Null` is written to the file.

For **Error** data, the output appears as `Error errorcode`. The **Error** keyword is not translated regardless of the locale.

All data written to the file using **Print** is internationally aware; that is, the data is properly formatted using the appropriate decimal separator. If the user wishes to output data for use by multiple locales, then **Write** should be used.

Writing to a file with the **Print** or **PrintLine** functions requires **Write** access from the **FileIOPermissionAccess** enumeration. For more information, see FileIOPermissionAccess Enumeration.

Example

This example uses the **Print** and **PrintLine** functions to write data to a file.

```
FileOpen(1, "c:\trash.txt", OpenMode.Output)   ' Open file for output.
Print(1, "This is a test.")   ' Print text to file.
PrintLine(1)   ' Print blank line to file.
PrintLine(1, "Zone 1", TAB(), "Zone 2")   ' Print in two print zones.
PrintLine(1, "Hello", "World")   ' Separate strings with a tab.
PrintLine(1, SPC(5), "5 leading spaces ")   ' Print five leading spaces.
PrintLine(1, TAB(10), "Hello")   ' Print word at column 10.

' Assign Boolean, Date, and Error values.
Dim aBool As Boolean
Dim aDate As DateTime
aBool = False
aDate = DateTime.Parse("February 12, 1969")

' Dates and booleans are translated using locale settings of your system.
PrintLine(1, aBool, " is a Boolean value")
PrintLine(1, aDate, " is a date")
FileClose(1)   ' Close file.
```

Smart Device Developer Notes

This function is not supported.

Requirements

Namespace: Microsoft.VisualBasic

Module: FileSystem

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

FileOpen Function | SPC Function | TAB Function | Write, WriteLine Functions | File Access with Visual Basic Run-Time Functions | IOException Class

PV Function

Returns a **Double** specifying the present value of an annuity based on periodic, fixed payments to be paid in the future and a fixed interest rate.

```
Function PV( _
   ByVal Rate As Double, _
   ByVal NPer As Double, _
   ByVal Pmt As Double, _
   Optional ByVal FV As Double = 0, _
   Optional ByVal Due As DueDate = DueDate.EndOfPeriod _
) As Double
```

Parameters

Rate
Required. **Double** specifying interest rate per period. For example, if you get a car loan at an annual percentage rate (APR) of 10 percent and make monthly payments, the rate per period is 0.1/12, or 0.0083.

NPer
Required. **Double** specifying total number of payment periods in the annuity. For example, if you make monthly payments on a four-year car loan, your loan has a total of 4 * 12 (or 48) payment periods.

Pmt
Required. **Double** specifying payment to be made each period. Payments usually contain principal and interest that doesn't change over the life of the annuity.

FV
Optional. **Double** specifying future value or cash balance you want after you've made the final payment. For example, the future value of a loan is $0 because that's its value after the final payment. However, if you want to save $50,000 over 18 years for your child's education, then $50,000 is the future value. If omitted, 0 is assumed.

Due
Optional. Object of type `Microsoft.VisualBasic.DueDate` that specifies when payments are due. This argument must be either `DueDate.EndOfPeriod` if payments are due at the end of the payment period, or `DueDate.BegOfPeriod` if payments are due at the beginning of the period. If omitted, `DueDate.EndOfPeriod` is assumed.

Remarks

An annuity is a series of fixed cash payments made over a period of time. An annuity can be a loan (such as a home mortgage) or an investment (such as a monthly savings plan).

The *Rate* and *NPer* arguments must be calculated using payment periods expressed in the same units. For example, if *Rate* is calculated using months, *NPer* must also be calculated using months.

For all arguments, cash paid out (such as deposits to savings) is represented by negative numbers; cash received (such as dividend checks) is represented by positive numbers.

Example

In this example, the **PV** function returns the present value of an $1,000,000 annuity that will provide $50,000 a year for the next 20 years. Provided are the expected annual percentage rate (`APR`), the total number of payments (`TotPmts`), the amount of each payment (`YrIncome`), the total future value of the investment (`FVal`), and a number that indicates whether each payment is made at the beginning or end of the payment period (`PayType`). Note that `YrIncome` is a negative number because it represents cash paid out from the annuity each year.

```
Sub TestPV()
   Dim Fmt As String = "###,##0.00"    ' Define money format.
   Dim APR, TotPmts, YrIncome, FVal, PVal As Double
   Dim PayType As DueDate
   APR = 0.0825  ' Annual percentage rate.
   TotPmts = 20   ' Total number of payments.
   YrIncome = 50000   ' Yearly income.
   FVal = 1000000   ' Future value.
   PayType = DueDate.BegOfPeriod   ' Payment at beginning of month.
   PVal = PV(APR, TotPmts, -YrIncome, FVal, PayType)
```

```
    MsgBox("The present value is " & Format(PVal, Fmt) & ".")
End Sub
```

Requirements

Namespace: Microsoft.VisualBasic

Module: Financial

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

DDB Function | FV Function | IPmt Function | IRR Function | MIRR Function | NPer Function | NPV Function | Pmt Function | PPmt Function | Rate Function | SLN Function | SYD Function

QBColor Function

Returns an **Integer** value representing the RGB color code corresponding to the specified color number.

```
QBColor(Color)
```

Parameter

Color
Required. A whole number in the range 0–15.

Settings

The *color* argument has these settings:

Number	Color	Number	Color
0	Black	8	Gray
1	Blue	9	Light blue
2	Green	10	Light green
3	Cyan	11	Light cyan
4	Red	12	Light red
5	Magenta	13	Light magenta
6	Yellow	14	Light yellow
7	White	15	Bright white

Exceptions/Errors

Exception type	Error number	Condition
ArgumentException	5	*Color* is outside of range 0 to 15, inclusive.

Remarks

Starting with the least-significant byte, the returned value specifies the red, green, and blue values used to set the appropriate color in the RGB system used by the Visual Basic language.

Example

This example uses the **QBColor** function to change to the color indicated by `colorInteger`. **QBColor** accepts integer values between 0 and 15.

```
Dim colorInteger As Integer
' Use 4 for red.
colorInteger = QBColor(4)
```

Requirements

Namespace: Microsoft.VisualBasic

Module: Information

Assembly: Microsoft Visual Basic .NET Runtime (in MicrosoftVisualBasic.dll)

See Also

RGB Function | ArgumentException Class

Rate Function

Returns a **Double** specifying the interest rate per period for an annuity.

```
Function Rate( _
   ByVal NPer As Double, _
   ByVal Pmt As Double, _
   ByVal PV As Double, _
   Optional ByVal FV As Double = 0, _
   Optional ByVal Due As DueDate = DueDate.EndOfPeriod, _
   Optional ByVal Guess As Double = 0.1 _
) As Double
```

Parameters

NPer
Required. **Double** specifying total number of payment periods in the annuity. For example, if you make monthly payments on a four-year car loan, your loan has a total of 4 * 12 (or 48) payment periods.

Pmt
Required. **Double** specifying payment to be made each period. Payments usually contain principal and interest that doesn't change over the life of the annuity.

PV
Required. **Double** specifying present value, or value today, of a series of future payments or receipts. For example, when you borrow money to buy a car, the loan amount is the present value to the lender of the monthly car payments you will make.

FV
Optional. **Double** specifying future value or cash balance you want after you make the final payment. For example, the future value of a loan is $0 because that's its value after the final payment. However, if you want to save $50,000 over 18 years for your child's education, then $50,000 is the future value. If omitted, 0 is assumed.

Due
Optional. Object of type `Microsoft.VisualBasic.DueDate` that specifies when payments are due. This argument must be either `DueDate.EndOfPeriod` if payments are due at the end of the payment period, or `DueDate.BegOfPeriod` if payments are due at the beginning of the period. If omitted, `DueDate.EndOfPeriod` is assumed.

Guess
Optional. **Double** specifying value you estimate will be returned by **Rate**. If omitted, *Guess* is 0.1 (10 percent).

Exceptions/Errors

Exception type	Error number	Condition
ArgumentException	5	*NPer* <= 0.

Remarks

An annuity is a series of fixed cash payments made over a period of time. An annuity can be a loan (such as a home mortgage) or an investment (such as a monthly savings plan).

For all arguments, cash paid out (such as deposits to savings) is represented by negative numbers; cash received (such as dividend checks) is represented by positive numbers.

Rate is calculated by iteration. Starting with the value of *Guess*, **Rate** cycles through the calculation until the result is accurate to within 0.00001 percent. If **Rate** can't find a result after 20 tries, it fails. If your guess is 10 percent and **Rate** fails, try a different value for *Guess*.

Example

This example uses the **Rate** function to calculate the interest rate of a loan given the total number of payments (`TotPmts`), the amount of the loan payment (`Payment`), the present value or principal of the loan (`PVal`), the future value of the loan (`FVal`), a number that indicates whether the payment is due at the beginning or end of the payment period (`PayType`), and an approximation of the expected interest rate (`Guess`).

```
Sub TestRate()
   Dim PVal, Payment, TotPmts, FVal, Guess, APR As Double
```

```
    Dim PayType As DueDate
    Dim Fmt As String = "##0.00"   ' Define percentage format.
    Dim Response As MsgBoxResult
    FVal = 0   ' Usually 0 for a loan.
    Guess = 0.1  ' Guess of 10 percent.
    PVal = CDbl(InputBox("How much did you borrow?"))
    Payment = CDbl(InputBox("What's your monthly payment?"))
    TotPmts = CDbl(InputBox("How many monthly payments do you have to make?"))
    Response = MsgBox("Do you make payments at the end of the month?", MsgBoxStyle.YesNo)
    If Response = MsgBoxResult.No Then
       PayType = DueDate.BegOfPeriod
    Else
       PayType = DueDate.EndOfPeriod
    End If
    APR = (Rate(TotPmts, -Payment, PVal, FVal, PayType, Guess) * 12) * 100
    MsgBox("Your interest rate is " & Format(CInt(APR), Fmt) & " percent.")
 End Sub
```

Requirements

Namespace: Microsoft.VisualBasic

Module: Financial

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

DDB Function | FV Function | IPmt Function | IRR Function | MIRR Function | NPer Function | NPV Function | Pmt Function | PPmt Function | PV Function | SLN Function | SYD Function | ArgumentException Class

Rename Function

Renames a disk file, directory, or folder.

```
Public Sub Rename( _
   ByVal OldPath As String, _
   ByVal NewPath As String _
)
```

Parameters

OldPath
Required. **String** expression that specifies the existing file name and location. *OldPath* may include the directory or folder, and drive, of the file.

NewPath
Required. **String** expression that specifies the new file name and location. *NewPath* may include directory or folder, and drive of the destination location. The file name specified by *NewPath* can't already exist.

Exceptions/Errors

Exception type	Error number	Condition
ArgumentException	5	Pathname is invalid .
FileNotFoundException	53	*OldPath* file does not exist.
IOException	58	*NewPath* file already exists.
IOException	75	Access is invalid.
IOException	74	Cannot rename to different device .

Remarks

The **Rename** function renames a file and moves it to a different directory or folder, if necessary. The **Rename** function can move a file across drives, but it can only rename an existing directory or folder when both *NewPath* and *OldPath* are located on the same drive. Name cannot create a new file, directory, or folder.

Using the **Rename** function on an open file produces an error. You must close an open file before renaming it. **Rename** arguments cannot include multiple-character (*) and single-character (?) wildcards.

Example

This example uses the **Rename** function to rename a file. For purposes of this example, assume that the directories or folders that are specified already exist.

```
Dim OldName, NewName As String
OldName = "OLDFILE"
NewName = "NEWFILE" ' Define file names.
Rename(OldName, NewName)   ' Rename file.

OldName = "C:\MYDIR\OLDFILE"
NewName = "C:\YOURDIR\NEWFILE"
Rename(OldName, NewName)   ' Move and rename file.
```

Smart Device Developer Notes

This function is not supported.

Requirements

Namespace: Microsoft.VisualBasic

Module: FileSystem

Assembly: Microsoft Visual Basic .NET Runtime (in MicrosoftVisualBasic.dll)

See Also

Kill Function | ArgumentException Class | FileNotFoundException Class | IOException Class

Replace Function

Returns a string in which a specified substring has been replaced with another substring a specified number of times.

```
Public Function Replace(
   ByVal Expression As String,
   ByVal Find As String,
   ByVal Replacement As String,
   Optional ByVal Start As Integer = 1,
   Optional ByVal Count As Integer = -1,
   Optional ByVal Compare As CompareMethod = CompareMethod.Binary
) As String
```

Parameters

Expression
Required. String expression containing substring to replace.
Find
Required. Substring being searched for.
Replacement
Required. Replacement substring.
Start
Optional. Position within *Expression* where substring search is to begin. If omitted, 1 is assumed.
Count
Optional. Number of substring substitutions to perform. If omitted, the default value is –1, which means make all possible substitutions.
Compare
Optional. Numeric value indicating the kind of comparison to use when evaluating substrings. See Settings for values.

Settings

The *Compare* argument can have the following values:

Constant	Description
Binary	Performs a binary comparison
Text	Performs a textual compariso n

Return Values

Replace returns the following values:

If	Replace returns
Expression is zero-length	Zero-length string ("")
Find is zero-length	Copy of *Expression*
Replace is zero-length	Copy of *Expression* with no occurrences of *Find*
Start is greater than length of *expression*	Zero-length string
Count is 0	Copy of *Expression*

Exceptions/Errors

Exception type	Error number	Condition
ArgumentException	5	*Count* < -1 or *Start* <= 0 .

Remarks

The return value of the **Replace** function is a string that begins at the position specified by *Start* and concludes at the end of the *Expression* string, with the substitutions made as specified by the *Find* and *Replace* values.

Example

This example demonstrates the **Replace** function.

```
Dim myString As String = "Shopping List"
Dim aString As String
' Returns "Shipping List".
aString = Replace(myString, "o", "i")
```

Requirements

Namespace: Microsoft.VisualBasic

Module: Strings

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Filter Function

Reset Function

Closes all disk files opened using the **FileOpen** function.

```
Public Sub Reset()
```

Remarks

The **Reset** function closes all active files opened by the **FileOpen** function and has the same function as **FileClose()** without any parameters.

Example

This example uses the **Reset** function to close all open files and write the contents of all file buffers to disk. Note the use of the **Object** variable `FileNumber` as both a string and a number.

```
' Open 5 files named TEST1, TEST2, etc.
Dim fileNumber As Integer
For fileNumber = 1 To 5    ' Open 5 files.
   FileOpen(fileNumber, "TEST" & fileNumber, OpenMode.Output)
   PrintLine(fileNumber, "Hello World")
Next fileNumber
Reset()   ' Close files and write contents to disk.
```

Smart Device Developer Notes

This function is not supported.

Requirements

Namespace: Microsoft.VisualBasic

Module: FileSystem

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

FileClose Function | End Statement | FileOpen Function | File Access with Visual Basic Run-Time Functions

RGB Function

Returns an **Integer** value representing an RGB color value from a set of red, green and blue color components.

```
RGB( _
   Red As Integer, _
   Green As Integer, _
   Blue As Integer _
)
```

Parameters

Red
Required. **Integer** in the range 0–255, inclusive, that represents the intensity of the red component of the color.
Green
Required. **Integer** in the range 0–255, inclusive, that represents the intensity of the green component of the color.
Blue
Required. **Integer** in the range 0–255, inclusive, that represents the intensity of the blue component of the color.

Exceptions/Errors

Exception type	Error number	Condition
ArgumentException	5	*Green*, *Blue*, or *Red* outside of range 0 and 255, inclusive.

Remarks

Application methods and properties that accept a color specification expect that specification to be a number representing an RGB color value. An RGB color value specifies the relative intensity of red, green, and blue to cause a specific color to be displayed.

If the value for any argument to **RGB** is greater than 255, 255 is used.

The following table lists some standard colors and the red, green, and blue values they include:

Color	Red value	Green value	Blue value
Black	0	0	0
Blue	0	0	255
Green	0	255	0
Cyan	0	255	255
Red	255	0	0
Magenta	255	0	255
Yellow	255	255	0
White	255	255	255

Example

This example shows how the **RGB** function is used to return a whole number representing an **RGB** color value.

```
Dim red, rgbValue As Integer
Dim i As Integer = 75
red = RGB(255, 0, 0)   ' Return the value for red.
rgbValue = RGB(i, 64 + i, 128 + i)    ' Same as RGB(75, 139, 203).
```

Requirements

Namespace: Microsoft.VisualBasic

Module: Information

Assembly: Microsoft Visual Basic .NET Runtime (in MicrosoftVisualBasic.dll)

See Also

QBColor Function | ArgumentException Class

Right Function

Returns a string containing a specified number of characters from the right side of a string.

```
Public Shared Function Right( _
   ByVal Str As String, _
   ByVal Length As Integer _
) As String
```

Parameters

Str
Required. **String** expression from which the rightmost characters are returned.

Length
Required. **Integer**. Numeric expression indicating how many characters to return. If 0, a zero-length string ("") is returned. If greater than or equal to the number of characters in *Str*, the entire string is returned.

Exceptions/Errors

Exception type	Error number	Condition
ArgumentException	5	*Length* < 0.

Remarks

To determine the number of characters in *Str*, use the **Len** function. If used in a Windows Form, or any other class that has a **Right** property, you must fully qualify the function with `Microsoft.VisualBasic.Right`.

> **Note** The **RightB** function in earlier versions of Visual Basic returns a string in bytes, rather than characters. It is used primarily for converting strings in double-byte character set (DBCS) applications. All Visual Basic .NET strings are in Unicode, and **RightB** is no longer supported.

Example

This example demonstrates the use of the **Right** function to return a substring of a given **String**. In a class that has a **Right** property, it may be necessary to fully qualify the **Right** function.

```
Dim myString As String = "Hello World!"
Dim subString As String
subString = Microsoft.VisualBasic.Right(myString, 6)   ' Returns "World!"
```

Requirements

Namespace: Microsoft.VisualBasic

Module: Strings

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Left Function | Len Function | Mid Function | ArgumentException Class | Programming Element Support Changes Summary

RmDir Function

Removes an existing directory or folder.

```
Public Sub RmDir(ByVal Path As String)
```

Parameter

Path
Required. **String** expression that identifies the directory or folder to be removed. *Path* may include the drive. If no drive is specified, **RmDir** removes the directory or folder on the current drive.

Exceptions/Errors

Exception type	Error number	Condition
ArgumentException	52	*Path* is not specified or is empty .
IOException	75	Target directory contains files.
FileNotFoundException	76	Directory does not exist.

Remarks

An error occurs if you try to use **RmDir** on a directory or folder containing files. Use the **Kill** function to delete all files before attempting to remove a directory or folder.

Example

This example uses the **RmDir** function to remove an existing directory or folder.

```
' Assume that MYDIR is an empty directory or folder.
RmDir ("MYDIR")   ' Remove MYDIR.
```

Smart Device Developer Notes

This function is not supported.

Requirements

Namespace: Microsoft.VisualBasic

Module: FileSystem

Assembly: Microsoft Visual Basic .NET Runtime (in MicrosoftVisualBasic.dll)

See Also

ChDir Function | CurDir Function | Kill Function | MkDir Function | ArgumentException Class | IOException Class | FileNotFoundException Class

Rnd Function

Returns a random number of type **Single**.

```
Public Shared Function Rnd[(number)] As Single
```

Parameter

number
Optional. A **Single** value or any valid **Single** expression.

Return Values

If *number* is	Rnd generates
Less than zero	The same number every time, using *number* as the seed.
Greater than zero	The next random number in the sequence.
Equal to zero	The most recently generated number.
Not supplied	The next random number in the sequence.

Remarks

The **Rnd** function returns a value less than 1, but greater than or equal to zero.

The value of *number* determines how **Rnd** generates a random number:

For any given initial seed, the same number sequence is generated because each successive call to the **Rnd** function uses the previously generated number as a seed for the next number in the sequence.

Before calling **Rnd,** use the **Randomize** statement without an argument to initialize the random-number generator with a seed based on the system timer.

To produce random integers in a given range, use this formula:

```
CInt(Int((upperbound - lowerbound + 1) * Rnd() + lowerbound))
```

Here, *upperbound* is the highest number in the range, and *lowerbound* is the lowest number in the range.

> **Note** To repeat sequences of random numbers, call **Rnd** with a negative argument immediately before using **Randomize** with a numeric argument. Using **Randomize** with the same value for *number* does not repeat the previous sequence.
>
> **Security Note** Because the **Random** statement and the **Rnd** function start with a seed value and generate numbers that fall within a finite range, the results may be predictable by someone who knows the algorithm used to generate them. Consequently, the **Random** statement and the **Rnd** function should not be used to generate random numbers for use in cryptography.

Example

This example uses the **Rnd** function to generate a random integer value from 1 to 6.

```
Dim MyValue As Integer
MyValue = CInt(Int((6 * Rnd()) + 1)) ' Generate random value between 1 and 6.
```

Requirements

Namespace: Microsoft.VisualBasic

Module: VBMath

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Math Summary | Randomize Statement

Math Functions

The math functions in Visual Basic 6 have been replaced by equivalent methods in the **System.Math** class of the .NET Framework.

Remarks

The .NET Framework math methods are functionally identical to their Visual Basic 6 counterparts, although some have slightly different names. For example, the .NET Framework equivalent of the Visual Basic 6 Atn function is **Atan**. The following table lists the Visual Basic 6 math function names and the equivalent .NET Framework methods.

Visual Basic 6 function	Visual Basic .NET method	Description
Abs	Math.Abs Method	Returns the absolute value of a specified number.
Atn	Math.Atan Method	Returns a **Double** value containing the angle whose tangent is the specified number.
Cos	Math.Cos Method	Returns a **Double** value containing the cosine of the specified angle.
Exp	Math.Exp Method	Returns a **Double** value containing e (the base of natural logarithms) raised to the specified power.
Log	Math.Log Method	Returns a **Double** value containing the logarithm of a specified number. This method is overloaded and can return either the natural (base e) logarithm of a specified number or the logarithm of a specified number in a specified base.
Round	Math.Round Method	Returns a **Double** value containing the number nearest the specified value. Additional round functions are available as methods of the intrinsic types such as Decimal.Round Method.
Sgn	Math.Sign Method	Returns an **Integer** value indicating the sign of a number.
Sin	Math.Sin Method	Returns a **Double** value specifying the sine of an angle.
Sqr	Math.Sqrt Method	Returns a **Double** value specifying the square root of a number.
Tan	Math.Tan Method	Returns a **Double** value containing the tangent of an angle.

In addition, the .NET Framework math class provides constants and other static methods for trigonometric, logarithmic, and other common mathematical functions. All of these can be used in a Visual Basic program.

To use these functions without qualification, import the **System.Math** namespace into your project by adding the following code to the top of the source code:

```
Imports System.Math
```

Requirements

Class: Math Class

Abs Example

This example uses the **Abs** method of the **Math** class to compute the absolute value of a number.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim MyNumber As Double
```

```
MyNumber = Abs(50.3)    ' Returns 50.3.
MyNumber = Abs(-50.3)   ' Returns 50.3.
```

Atan Example

This example uses the **Atan** method of the **Math** class to calculate the value of pi.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim pi As Double
pi = 4 * Atan(1)   ' Calculate the value of pi.
```

Cos Example

This example uses the **Cos** method of the **Math** class to return the cosine of an angle.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim MyAngle, MySecant As Double
MyAngle = 1.3   ' Define angle in radians.
MySecant = 1 / Cos(MyAngle)   ' Calculate secant.
```

Exp Example

This example uses the **Exp** method of the **Math** class to return *e* raised to a power.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim MyAngle, MyHSin As Double
' Define angle in radians.
MyAngle = 1.3
' Calculate hyperbolic sine.
MyHSin = (Exp(MyAngle) - Exp(-1 * MyAngle)) / 2
```

Log Example

This example uses the **Log** method of the **Math** class to return the natural logarithm of a number.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim MyAngle, MyLog As Double
' Define angle in radians.
MyAngle = 1.3
' Calculate inverse hyperbolic sine.
MyLog = Log(MyAngle + Sqrt(MyAngle * MyAngle + 1))
```

Round Example

This example uses the **Round** method of the **Math** class to round a number to the nearest integer.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim MyVar1 As Double = 2.8
Dim MyVar2 As Double
MyVar2 =Round(MyVar1)   ' Returns 3.
```

Sign Example

This example uses the **Sign** method of the **Math** class to determine the sign of a number.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim MyVar1, MyVar2, MyVar3 As Double
Dim MySign As Integer
MyVar1 = 12
MyVar2 = -2.4
MyVar3 = 0
MySign = Sign(MyVar1)   ' Returns 1.
MySign = Sign(MyVar2)   ' Returns -1.
MySign = Sign(MyVar3)   ' Returns 0.
```

Sin Example

This example uses the **Sin** method of the **Math** class to return the sine of an angle.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim MyAngle, MyCosecant As Double
MyAngle = 1.3   ' Define angle in radians.
MyCosecant = 1 / Sin(MyAngle)   ' Calculate cosecant.
```

Sqrt Example

This example uses the **Sqrt** method of the **Math** class to calculate the square root of a number.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim MySqr As Double
MySqr = Sqrt(4)    ' Returns 2.
MySqr = Sqrt(23)   ' Returns 4.79583152331272.
MySqr = Sqrt(0)    ' Returns 0.
MySqr = Sqrt(-4)   ' Returns NaN (not a number).
```

Tan Example

This example uses the **Tan** method of the **Math** class to return the tangent of an angle.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim MyAngle, MyCotangent As Double
MyAngle = 1.3   ' Define angle in radians.
MyCotangent = 1 / Tan(MyAngle)   ' Calculate cotangent.
```

Requirements

Namespace: System.Math

Assembly: mscorlib (in mscorlib.dll)

See Also

Rnd Function | Randomize Statement | Derived Math Functions | Double.NaN Field

RSet Function

Returns a right-aligned string containing the specified string adjusted to the specified length.

```
Public Shared Function RSet( _
   ByVal Source As String, _
   ByVal Length As Integer _
) As String
```

Parameters

Source
Required. **String** expression. Name of string variable.
Length
Required. **Integer** expression. Length of returned string.

Remarks

If *Source* is longer than *Length*, **RSet** places only the leftmost characters, up to the length of *Source,* in the returned string. If the specified string is shorter than the specified length, spaces are added to the left end of the string to produce the appropriate length. If the specified string is longer than the specified length, it is shortened to the specified length.

Example

This example demonstrates the use of the **RSet** function.

```
Dim myString As String = "Right"
Dim rString As String
rString = RSet(myString, 11)   ' Returns "      Right"
```

Requirements

Namespace: Microsoft.VisualBasic

Module: Strings

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Data Type Summary | LSet Function | Efficient Use of Data Types

LTrim, RTrim, and Trim Functions

Returns a string containing a copy of a specified string with no leading spaces (**LTrim**), no trailing spaces (**RTrim**), or no leading or trailing spaces (**Trim**).

```
Public Shared Function LTrim(ByVal Str As String) As String
Public Shared Function RTrim(ByVal Str As String) As String
Public Shared Function Trim(ByVal Str As String) As String
```

Parameter

Str
Required. Any valid **String** expression.

Example

This example uses the **LTrim** function to strip leading spaces and the **RTrim** function to strip trailing spaces from a string variable. It uses the **Trim** function to strip both types of spaces.

```
Dim MyString, TrimString As String
MyString = "  <-Trim->  "   ' Initializes string.
TrimString = LTrim(MyString)   ' TrimString = "<-Trim->  ".
TrimString = RTrim(MyString)   ' TrimString = "  <-Trim->".
TrimString = LTrim(RTrim(MyString))   ' TrimString = "<-Trim->".
' Using the Trim function alone achieves the same result.
TrimString = Trim(MyString)   ' TrimString = "<-Trim->".
```

Requirements

Namespace: Microsoft.VisualBasic

Module: Strings

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Left Function | Right Function

S-Z Functions

The following table contains a listing of the Visual Basic .NET run-time member functions.

SaveSetting	Second	Seek	SetAttr
Sgn	Shell	Sin	SLN
Space	SPC	Split	Sqr
Str	StrComp	StrConv	StrDup
StrReverse	Switch	SYD	SystemTypeName
TAB	Tan	TimeSerial	TimeValue
Trim	TypeName	UBound	UCase
Unlock	Val	VarType	VbTypeName
Weekday	WeekdayName	Write	WriteLine
Year			

See Also

A-C Functions | D-G Functions | H-L Functions | M-R Functions | Visual Basic Language and Run-Time Reference

SaveSetting Function

Saves or creates an application entry in the Windows registry.

```
Public Sub SaveSetting( _
   ByVal AppName As String, _
   ByVal Section As String, _
   ByVal Key As String, _
   ByVal Setting As String _
)
```

Parameters

AppName
Required. **String** expression containing the name of the application or project to which the setting applies.
Section
Required. **String** expression containing the name of the section in which the key setting is being saved.
Key
Required. **String** expression containing the name of the key setting being saved.
Setting
Required. Expression containing the value to which *Key* is being set.

Exceptions/Errors

Exception type	Error number	Condition
ArgumentException	5	Key registry could not be created, or user is not logged in.

Remarks

The **SaveSetting** function adds the key to HKEY_CURRENT_USER\Software\VB and VBA Program Settings.

If the key setting can't be saved for any reason, an error occurs.

SaveSetting requires that a user be logged on since it operates under the HKEY_LOCAL_USER registry key, which is not active until a user logs on interactively.

Registry settings that are to be accessed from a non-interactive process (such as mtx.exe) should be stored under either the HKEY_LOCAL_MACHINE\Software\ or the HKEY_USER\DEFAULT\Software registry keys.

GetSetting requires **Write** and **Create** Registry Permission.

Example

The following example first uses the **SaveSetting** function to make entries in the Windows registry for the `MyApp` application, and then uses the **DeleteSetting** function to remove them.

```
' Place some settings in the registry.
SaveSetting("MyApp", "Startup", "Top", "75")
SaveSetting("MyApp","Startup", "Left", "50")
' Remove Startup section and all its settings from registry.
DeleteSetting ("MyApp", "Startup")
' Remove MyApp from the registry.
DeleteSetting ("MyApp")
```

Smart Device Developer Notes

This function is not supported.

Requirements

Namespace: Microsoft.VisualBasic

Module: Interaction

Assembly: Microsoft.Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

DeleteSetting Function | GetAllSettings Function | GetSetting Function | RegistryPermission Class

Second Function

Returns an **Integer** value from 0 through 59 representing the second of the minute.

```
Public Function Second(ByVal TimeValue As DateTime) As Integer
```

Parameter

TimeValue
Required. **Date** value from which you want to extract the second.

You can also obtain the second of the minute by calling **DatePart** and specifying **DateInterval.Second** for the *Interval* argument.

Example

This example uses the **Second** function to obtain the second of the minute from a specified time. In the development environment, the time literal is displayed in short time format using the locale settings of your code.

```
Dim MyTime As Date
Dim MySecond As Integer
MyTime = #4:35:17 PM#   ' Assign a time.
MySecond = Second(MyTime)   ' MySecond contains 17.
```

Requirements

Namespace: Microsoft.VisualBasic

Module: DateAndTime

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Day Function | Hour Function | Minute Function | Now Property | TimeOfDay Property | DatePart Function | System Namespace | DateTime Structure | ArgumentException Class | ArgumentOutOfRangeException Class

Seek Function

Returns a **Long** specifying the current read/write position within a file opened using the **FileOpen** function or sets the position for the next read/write operation within a file opened using the **FileOpen** function.

```
Public Overloads Function Seek( _
   ByVal FileNumber As Integer _
) As Long
```

-or-

```
Public Overloads Sub Seek( _
   ByVal FileNumber As Integer, _
   ByVal Position As Long _
)
```

Parameter

FileNumber
Required. An **Integer** containing a valid file number.
Position
Required. Number in the range 1 – 2,147,483,647, inclusive, that indicates where the next read/write operation should occur.

Exceptions/Errors

Exception type	Error number	Condition
IOException	52	*FileNumber* does not exist.
IOException	54	File mode is invalid.

Remarks

Seek returns a value between 1 and 2,147,483,647 (equivalent to 2^31 – 1), inclusive.

The following describes the return values for each file access mode:

Mode	Return Value
Random	Number of the next record read or written
Binary, **Input, Output**, **Append**	Byte position at which the next operation takes place. The first byte in a file is at position 1, the seco nd byte is at position 2, and so on.

Example

This example uses the **Seek** function to return the current file position. The example assumes `TESTFILE` is a file containing records of the structure `Record`.

```
Structure Record   ' Define user-defined type.
   ID As Integer
   Name As String
End Structure
```

For files opened in **Random** mode, **Seek** returns the number of next record.

```
Dim MyRecord As Record
FileOpen(1, "TESTFILE", OpenMode.Random)
Do While Not EOF(1)
   Debug.WriteLine(Seek(1))   ' Print record number.
   FileGet(1, MyRecord)   ' Read next record.
Loop
FileClose(1)
```

For files opened in modes other than **Random** mode, **Seek** returns the byte position at which the next operation takes place. Assume `TESTFILE` is a file containing a few lines of text.

```
' Print character position at beginning of each line.
Dim TextLine As String
FileOpen(1, "TESTFILE", OpenMode.Input)   ' Open file for reading.
While Not EOF(1)
   TextLine = LineInput(1)   ' Read next line.
   Debug.WriteLine(Seek(1))   ' Position of next line.
End While
FileClose(1)
```

This example uses the **Seek** function to set the position for the next read or write within a file. This example assumes `people.txt` is a file containing records of the structure `Record`.

```
Structure Record
   Dim Name As String
   Dim ID As Integer
End Structure
```

For files opened in **Random** mode, **Seek** sets the next record.

```
Public Sub SeekAPerson(ByVal index As Integer)
   Try
      FileOpen(1, "c:\people.txt", OpenMode.Random)
      Dim onePerson As Record
      Seek(1, index)
      FileGet(1, onePerson)
      FileClose(1)
      Console.WriteLine(onePerson.Name & " " & onePerson.ID)
   Catch
      ' Error recovery code here.
   End Try
End Sub
```

For files opened in modes other than **Random** mode, **Seek** sets the byte position at which the next operation takes place. Assume `TESTFILE` is a file containing a few lines of text.

```
Dim someText As String
FileOpen(1, "TESTFILE", OpenMode.Input)   ' Open file for output.
Seek(1, 3)   ' Move to the third character.
Input(1, someText)
Console.WriteLine(someText)
FileClose(1)
```

Smart Device Developer Notes

This function is not supported.

Requirements

Namespace: Microsoft.VisualBasic

Module: FileSystem

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

FileGet Function | Loc Function | FileOpen Function | FilePut Function | IOException Class

SetAttr Function

Sets attribute information for a file.

```
Public Sub SetAttr( _
   ByVal PathName As String, _
   ByVal Attributes As FileAttribute _
)
```

Parameters

PathName
Required. **String** expression that specifies a file name. *PathName* may include directory or folder, and drive.
Attributes
Required. Constant or numeric expression, whose sum specifies file attributes.

Settings

The *Attributes* argument enumeration values are:

Value	Constant	Description
Normal	**vbNormal**	Normal (default).
ReadOnly	**vbReadOnly**	Read-only.
Hidden	**vbHidden**	Hidden.
System	**vbSystem**	System file.
Volume	**vbVolume**	Volume label
Directory	**vbDirectory**	Directory or folder.
Archive	**vbArchive**	File has changed since last backup.
Alias	**vbAlias**	File has a different name.

Note These enumerations are specified by the Visual Basic language. The names can be used anywhere in your code in place of the actual values.

Exceptions/Errors

Exception type	Error number	Condition
ArgumentException	52	*PathName* invalid or does not exist .
ArgumentException	5	*Attribute* type is invalid.

Remarks

A run-time error occurs if you try to set the attributes of an open file.

Example

This example uses the **SetAttr** function to set attributes for a file.

```
SetAttr("TESTFILE", vbHidden)   ' Set hidden attribute.
SetAttr("TESTFILE", vbHidden Or vbReadOnly)   ' Set hidden and read-only
   ' attributes.
```

Smart Device Developer Notes

This function is not supported.

Requirements

Namespace: Microsoft.VisualBasic

Module: FileSystem

Assembly: Microsoft Visual Basic .NET Runtime (in MicrosoftVisualBasic.dll)

See Also

FileAttr Function | GetAttr Function | ArgumentException Class | FileAttribute Enumeration

Math Functions

The math functions in Visual Basic 6 have been replaced by equivalent methods in the **System.Math** class of the .NET Framework.

Remarks

The .NET Framework math methods are functionally identical to their Visual Basic 6 counterparts, although some have slightly different names. For example, the .NET Framework equivalent of the Visual Basic 6 Atn function is **Atan**. The following table lists the Visual Basic 6 math function names and the equivalent .NET Framework methods.

Visual Basic 6 function	Visual Basic .NET method	Description
Abs	Math.Abs Method	Returns the absolute value of a specified number.
Atn	Math.Atan Method	Returns a **Double** value containing the angle whose tangent is the specified number.
Cos	Math.Cos Method	Returns a **Double** value containing the cosine of the specified angle.
Exp	Math.Exp Method	Returns a **Double** value containing e (the base of natural logarithms) raised to the specified power.
Log	Math.Log Method	Returns a **Double** value containing the logarithm of a specified number. This method is overloaded and can return either the natural (base e) logarithm of a specified number or the logarithm of a specified number in a specified base.
Round	Math.Round Method	Returns a **Double** value containing the number nearest the specified value. Additional round functions are available as methods of the intrinsic types such as Decimal.Round Method.
Sgn	Math.Sign Method	Returns an **Integer** value indicating the sign of a number.
Sin	Math.Sin Method	Returns a **Double** value specifying the sine of an angle.
Sqr	Math.Sqrt Method	Returns a **Double** value specifying the square root of a number.
Tan	Math.Tan Method	Returns a **Double** value containing the tangent of an angle.

In addition, the .NET Framework math class provides constants and other static methods for trigonometric, logarithmic, and other common mathematical functions. All of these can be used in a Visual Basic program.

To use these functions without qualification, import the **System.Math** namespace into your project by adding the following code to the top of the source code:

```
Imports System.Math
```

Requirements

Class: Math Class

Abs Example

This example uses the **Abs** method of the **Math** class to compute the absolute value of a number.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim MyNumber As Double
```

```
MyNumber = Abs(50.3)    ' Returns 50.3.
MyNumber = Abs(-50.3)   ' Returns 50.3.
```

Atan Example

This example uses the **Atan** method of the **Math** class to calculate the value of pi.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim pi As Double
pi = 4 * Atan(1)   ' Calculate the value of pi.
```

Cos Example

This example uses the **Cos** method of the **Math** class to return the cosine of an angle.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim MyAngle, MySecant As Double
MyAngle = 1.3   ' Define angle in radians.
MySecant = 1 / Cos(MyAngle)   ' Calculate secant.
```

Exp Example

This example uses the **Exp** method of the **Math** class to return *e* raised to a power.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim MyAngle, MyHSin As Double
' Define angle in radians.
MyAngle = 1.3
' Calculate hyperbolic sine.
MyHSin = (Exp(MyAngle) - Exp(-1 * MyAngle)) / 2
```

Log Example

This example uses the **Log** method of the **Math** class to return the natural logarithm of a number.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim MyAngle, MyLog As Double
' Define angle in radians.
MyAngle = 1.3
' Calculate inverse hyperbolic sine.
MyLog = Log(MyAngle + Sqrt(MyAngle * MyAngle + 1))
```

Round Example

This example uses the **Round** method of the **Math** class to round a number to the nearest integer.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim MyVar1 As Double = 2.8
Dim MyVar2 As Double
MyVar2 =Round(MyVar1)   ' Returns 3.
```

Sign Example

This example uses the **Sign** method of the **Math** class to determine the sign of a number.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim MyVar1, MyVar2, MyVar3 As Double
Dim MySign As Integer
MyVar1 = 12
MyVar2 = -2.4
MyVar3 = 0
MySign = Sign(MyVar1)   ' Returns 1.
MySign = Sign(MyVar2)   ' Returns -1.
MySign = Sign(MyVar3)   ' Returns 0.
```

Sin Example

This example uses the **Sin** method of the **Math** class to return the sine of an angle.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim MyAngle, MyCosecant As Double
MyAngle = 1.3   ' Define angle in radians.
MyCosecant = 1 / Sin(MyAngle)   ' Calculate cosecant.
```

Sqrt Example

This example uses the **Sqrt** method of the **Math** class to calculate the square root of a number.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim MySqr As Double
MySqr = Sqrt(4)    ' Returns 2.
MySqr = Sqrt(23)   ' Returns 4.79583152331272.
MySqr = Sqrt(0)    ' Returns 0.
MySqr = Sqrt(-4)   ' Returns NaN (not a number).
```

Tan Example

This example uses the **Tan** method of the **Math** class to return the tangent of an angle.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim MyAngle, MyCotangent As Double
MyAngle = 1.3   ' Define angle in radians.
MyCotangent = 1 / Tan(MyAngle)   ' Calculate cotangent.
```

Requirements

Namespace: System.Math

Assembly: mscorlib (in mscorlib.dll)

See Also

Rnd Function | Randomize Statement | Derived Math Functions | Double.NaN Field

Shell Function

Runs an executable program and returns an integer containing the program's process ID if it is still running.

```
Public Function Shell( _
   ByVal Pathname As String, _
   Optional ByVal Style As AppWinStyle = AppWinStyle.MinimizedFocus, _
   Optional ByVal Wait As Boolean = False, _
   Optional ByVal Timeout As Integer = -1 _
) As Integer
```

Parameters

Pathname
Required. **String**. Name of the program to execute, together with any required arguments and command-line switches. *Pathname* can also include the drive and the directory path or folder.

Style
Optional. **AppWinStyle**. A value chosen from the **AppWinStyle** enumeration corresponding to the style of the window in which the program is to be run. If *Style* is omitted, **Shell** uses **AppWinStyle.MinimizedFocus**, which starts the program minimized and with focus.

The *Style* argument can have one of the following settings:

Enumeration value	Description
AppWinStyle.Hide	The window is hidden and focus is given to the hidden window.
AppWinStyle.NormalFocus	The window is given focus and displayed in its most recent size and position.
AppWinStyle.MinimizedFocus	The window is given focus and displayed as an icon.
AppWinStyle.MaximizedFocus	The window is given focus and displayed using the entire screen.
AppWinStyle.NormalNoFocus	The window is set to its most recent size and position. The currently active window remains in focus.
AppWinStyle.MinimizedNoFocus	The window is displayed as an icon. The currently active window remains in focus.

Wait
Optional. **Boolean**. A value indicating whether the **Shell** function should wait for completion of the program. If *Wait* is omitted, **Shell** uses **False**.

Timeout
Optional. **Integer**. The number of milliseconds to wait for completion if *Wait* is **True**. If *Timeout* is omitted, **Shell** uses -1, which means there is no timeout and **Shell** does not return until the program completes. Therefore, if you omit *Timeout* or set it to -1, it is possible that **Shell** might never return control to your program.

Exceptions/Errors

Exception type	Error number	Condition
ArgumentException	5	*Style* is not within range 0 through 9, inclusive.
FileNotFoundException	53	**Shell** cannot start the named program.

Remarks

The return value of the **Shell** function depends on whether the program named in *Pathname* is still executing when **Shell** returns. If you set *Wait* to **True** and the program finishes before the timeout expires, **Shell** returns zero. If the timeout expires, or if you omit *Wait* or set it to **False**, **Shell** returns the process ID of the program. The process ID is a unique number that identifies the running program.

If the **Shell** function cannot start the named program, a **System.IO.FileNotFoundException** error occurs. This can happen, for example, when you attempt to run a 16-bit program, such as `command.com`, from an application using **System.Windows.Forms**. For a workaround, you can run a 32-bit program that calls the desired 16-bit program. In the case of `command.com`, you can run `cmd.exe` as an alternative.

By default, the **Shell** function runs the program asynchronously. This means that a program started with the **Shell** function might not finish executing before the statements following the **Shell** function are executed. If you want to wait for the program to finish before you continue, set *Wait* to **True**.

You should always enclose the entire path and file specification in quotes, as the following example shows:

```
ID = Shell("""C:\Program Files\MyFile.exe"" -a -q", , True, 100000)
```

Each pair of adjacent double quotes (`""`) within the string literal is interpreted as one double quote character in the string. Therefore, the preceding example presents the following string to the **Shell** function:

```
"C:\Program Files\MyFile.exe" -a -q
```

If you did not have the path enclosed in quotes, Windows would look for a file called `Program.exe` in the C:\ directory, instead of `MyFile.exe` in the C:\Program Files directory.

> **Security Note** If you do not enclose the path and file specification in quotes, there is a security risk if the file name or a path node contains spaces. In the preceding example, the path node `\Program Files` includes a space. If the specification were not inside quotes and a program named `Program.exe` had been installed in C:\, for example by illicit tampering, Windows would execute it instead of `MyFile.exe`.
>
> **Security Note** The **Shell** function requires unmanaged code permission, which might affect its execution in partial trust situations. For more information, see SecurityPermission Class and Code Access Permissions.

Example

This example uses the **Shell** function to run an application specified by the user. Specifying **AppWinStyle.NormalFocus** as the second argument opens the application in normal size and gives it the focus.

```
Dim ProcID As Integer
   ' Run Calculator.
ProcID = Shell("C:\Windows\system32\calc.exe", AppWinStyle.NormalFocus)
' The preceding path is for Windows XP;
' The Windows 2000 path is C:\WINNT\system32\calc.exe.
```

Smart Device Developer Notes

This function is not supported.

Requirements

Namespace: Microsoft.VisualBasic

Module: Interaction

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

AppActivate Function | ArgumentException Class | FileNotFoundException Class

Math Functions

The math functions in Visual Basic 6 have been replaced by equivalent methods in the **System.Math** class of the .NET Framework.

Remarks

The .NET Framework math methods are functionally identical to their Visual Basic 6 counterparts, although some have slightly different names. For example, the .NET Framework equivalent of the Visual Basic 6 Atn function is **Atan**. The following table lists the Visual Basic 6 math function names and the equivalent .NET Framework methods.

Visual Basic 6 function	Visual Basic .NET method	Description
Abs	Math.Abs Method	Returns the absolute value of a specified number.
Atn	Math.Atan Method	Returns a **Double** value containing the angle whose tangent is the specified number.
Cos	Math.Cos Method	Returns a **Double** value containing the cosine of the specified angle.
Exp	Math.Exp Method	Returns a **Double** value containing e (the base of natural logarithms) raised to the specified power.
Log	Math.Log Method	Returns a **Double** value containing the logarithm of a specified number. This method is overloaded and can return either the natural (base e) logarithm of a specified number or the logarithm of a specified number in a specified base.
Round	Math.Round Method	Returns a **Double** value containing the number nearest the specified value. Additional round functions are available as methods of the intrinsic types such as Decimal.Round Method.
Sgn	Math.Sign Method	Returns an **Integer** value indicating the sign of a number.
Sin	Math.Sin Method	Returns a **Double** value specifying the sine of an angle.
Sqr	Math.Sqrt Method	Returns a **Double** value specifying the square root of a number.
Tan	Math.Tan Method	Returns a **Double** value containing the tangent of an angle.

In addition, the .NET Framework math class provides constants and other static methods for trigonometric, logarithmic, and other common mathematical functions. All of these can be used in a Visual Basic program.

To use these functions without qualification, import the **System.Math** namespace into your project by adding the following code to the top of the source code:

```
Imports System.Math
```

Requirements

Class: Math Class

Abs Example

This example uses the **Abs** method of the **Math** class to compute the absolute value of a number.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim MyNumber As Double
```

```
MyNumber = Abs(50.3)    ' Returns 50.3.
MyNumber = Abs(-50.3)   ' Returns 50.3.
```

Atan Example

This example uses the **Atan** method of the **Math** class to calculate the value of pi.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim pi As Double
pi = 4 * Atan(1)   ' Calculate the value of pi.
```

Cos Example

This example uses the **Cos** method of the **Math** class to return the cosine of an angle.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim MyAngle, MySecant As Double
MyAngle = 1.3   ' Define angle in radians.
MySecant = 1 / Cos(MyAngle)   ' Calculate secant.
```

Exp Example

This example uses the **Exp** method of the **Math** class to return *e* raised to a power.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim MyAngle, MyHSin As Double
' Define angle in radians.
MyAngle = 1.3
' Calculate hyperbolic sine.
MyHSin = (Exp(MyAngle) - Exp(-1 * MyAngle)) / 2
```

Log Example

This example uses the **Log** method of the **Math** class to return the natural logarithm of a number.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim MyAngle, MyLog As Double
' Define angle in radians.
MyAngle = 1.3
' Calculate inverse hyperbolic sine.
MyLog = Log(MyAngle + Sqrt(MyAngle * MyAngle + 1))
```

Round Example

This example uses the **Round** method of the **Math** class to round a number to the nearest integer.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim MyVar1 As Double = 2.8
Dim MyVar2 As Double
MyVar2 =Round(MyVar1)   ' Returns 3.
```

Sign Example

This example uses the **Sign** method of the **Math** class to determine the sign of a number.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim MyVar1, MyVar2, MyVar3 As Double
Dim MySign As Integer
MyVar1 = 12
MyVar2 = -2.4
MyVar3 = 0
MySign = Sign(MyVar1)   ' Returns 1.
MySign = Sign(MyVar2)   ' Returns -1.
MySign = Sign(MyVar3)   ' Returns 0.
```

Sin Example

This example uses the **Sin** method of the **Math** class to return the sine of an angle.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim MyAngle, MyCosecant As Double
MyAngle = 1.3   ' Define angle in radians.
MyCosecant = 1 / Sin(MyAngle)   ' Calculate cosecant.
```

Sqrt Example

This example uses the **Sqrt** method of the **Math** class to calculate the square root of a number.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim MySqr As Double
MySqr = Sqrt(4)    ' Returns 2.
MySqr = Sqrt(23)   ' Returns 4.79583152331272.
MySqr = Sqrt(0)    ' Returns 0.
MySqr = Sqrt(-4)   ' Returns NaN (not a number).
```

Tan Example

This example uses the **Tan** method of the **Math** class to return the tangent of an angle.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim MyAngle, MyCotangent As Double
MyAngle = 1.3   ' Define angle in radians.
MyCotangent = 1 / Tan(MyAngle)   ' Calculate cotangent.
```

Requirements

Namespace: System.Math

Assembly: mscorlib (in mscorlib.dll)

See Also

Rnd Function | Randomize Statement | Derived Math Functions | Double.NaN Field

SLN Function

Returns a **Double** specifying the straight-line depreciation of an asset for a single period.

```
Function SLN( _
   ByVal Cost As Double, _
   ByVal Salvage As Double, _
   ByVal Life As Double _
) As Double
```

Parameters

Cost
Required. **Double** specifying initial cost of the asset.
Salvage
Required. **Double** specifying value of the asset at the end of its useful life.
Life
Required. **Double** specifying length of the useful life of the asset.

Exceptions/Errors

Exception type	Error number	Condition
ArgumentException	5	*Life* = 0.

Remarks

The depreciation period must be expressed in the same unit as the *Life* argument. All arguments must be positive numbers.

Example

This example uses the **SLN** function to return the straight-line depreciation of an asset for a single period given the asset's initial cost (`InitCost`), the salvage value at the end of the asset's useful life (`SalvageVal`), and the total life of the asset in years (`LifeTime`).

```
Sub TestSLN()
   Dim InitCost, SalvageVal, MonthLife As Double
   Dim LifeTime, Pdepr As Double
   Dim Fmt As String = "###,##0.00"   ' Define money format.
   Const YEARMONTHS As Integer = 12  ' Number of months in a year.

   InitCost = CDbl(InputBox("What's the initial cost of the asset?"))
   SalvageVal = CDbl(InputBox("What's the asset's value at the end of its useful life?"))
   MonthLife = CDbl(InputBox("What's the asset's useful life in months?"))
   Do While MonthLife < YEARMONTHS   ' Ensure period is >= 1 year.
      MsgBox("Asset life must be a year or more.")
      MonthLife = CDbl(InputBox("What's the asset's useful life in months?"))
   Loop
   LifeTime = MonthLife / YEARMONTHS   ' Convert months to years.
   If LifeTime <> Int(MonthLife / YEARMONTHS) Then
      LifeTime = Int(LifeTime + 1)   ' Round up to nearest year.
   End If
   Pdepr = SLN(InitCost, SalvageVal, LifeTime)
   MsgBox("The depreciation is " & Format(Pdepr, Fmt) & " per year.")
End Sub
```

Requirements

Namespace: Microsoft.VisualBasic

Module: Financial

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

DDB Function | FV Function | IPmt Function | IRR Function | MIRR Function | NPer Function | NPV Function | Pmt Function | PPmt Function | PV Function | Rate Function | SYD Function | ArgumentException Class

Space Function

Returns a string consisting of the specified number of spaces.

```
Public Shared Function Space(ByVal Number As Integer) As String
```

Parameter

Number
Required. **Integer** expression. The number of spaces you want in the string.

Exceptions/Errors

Exception type	Error number	Condition
ArgumentException	5	*Number* < 0.

Remarks

The **Space** function is useful for formatting output and clearing data in fixed-length strings.

Example

This example uses the **Space** function to return a string consisting of a specified number of spaces.

```
Dim MyString As String
' Returns a string with 10 spaces.
MyString = Space(10)
' Inserts 10 spaces between two strings.
MyString = "Hello" & Space(10) & "World"
```

Requirements

Namespace: Microsoft.VisualBasic

Module: Strings

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

SPC Function | ArgumentException Class

SPC Function

Used with the **Print** or **PrintLine** functions to position output.

```
Public Function SPC(ByVal Count As Short) As SPCInfo
```

Parameters

Count
Required. The number of spaces to insert before displaying or printing the next expression in a list.

Remarks

If *Count* is less than the output line width, the next print position immediately follows the number of spaces printed. If *Count* is greater than the output line width, **SPC** calculates the next print position using the formula:

```
currentprintposition + (Count Mod width)
```

For example, if the current print position is 24, the output line width is 80, and you specify **SPC(**90**)**, the next print will start at position 34 (current print position + the remainder of 90/80). If the difference between the current print position and the output line width is less than *Count* (or *Count* **Mod** *width*), the **SPC** function skips to the beginning of the next line and generates spaces equal to *Count* – (*width* – *currentprintposition*).

Note Make sure your tabular columns are wide enough to accommodate wide letters.

Example

This example uses the **SPC** function to position output in a file and in the **Output** window.

```
' The SPC function can be used with the Print function.
FileOpen(1, "TESTFILE", OpenMode.Output)   ' Open file for output.
Print(1, "10 spaces between here", SPC(10), "and here.")
FileClose(1)   ' Close file.
```

The following statement causes the text to be printed in the **Output** window (using the **WriteLine** method), preceded by 30 spaces.

```
Debug.WriteLine(SPC(30), "Thirty spaces later...")
```

Smart Device Developer Notes

This function is not supported.

Requirements

Namespace: Microsoft.VisualBasic

Module: FileSystem

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Mod Operator | Print, PrintLine Functions | Space Function | TAB Function | FileWidth Function

Split Function

Returns a zero-based, one-dimensional array containing a specified number of substrings.

```
Function Split(
   ByVal Expression As String,
   Optional ByVal Delimiter As String = " ",
   Optional ByVal Limit As Integer = -1,
   Optional ByVal Compare As CompareMethod = CompareMethod.Binary
) As String()
```

Parameters

Expression
Required. **String** expression containing substrings and delimiters. If *Expression* is a zero-length string ("""), *the* **Split** function returns an array of length one, containing an empty string.

Delimiter
Optional. Single character used to identify substring limits. If *Delimiter* is omitted, the space character (" ") is assumed to be the delimiter. If *Delimiter* is a zero-length string, a single-element array containing the entire *Expression* string is returned.

Limit
Optional. Number of substrings to be returned; the default, –1, indicates that all substrings are returned.

Compare
Optional. Numeric value indicating the comparison to use when evaluating substrings. See Settings for values.

Settings

The *Compare* argument can have the following values:

Constant	Description
Binary	Performs a binary comparison
Text	Performs a textual compariso n

Example

The following example demonstrates the **Split** function:

```
Dim myString As String = "Look at these!"
' Returns ["Look", "at", "these!"]
Dim myArray() As String = Split(myString)
```

Requirements

Namespace: Microsoft.VisualBasic

Module: Strings

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Join Function

Math Functions

The math functions in Visual Basic 6 have been replaced by equivalent methods in the **System.Math** class of the .NET Framework.

Remarks

The .NET Framework math methods are functionally identical to their Visual Basic 6 counterparts, although some have slightly different names. For example, the .NET Framework equivalent of the Visual Basic 6 Atn function is **Atan**. The following table lists the Visual Basic 6 math function names and the equivalent .NET Framework methods.

Visual Basic 6 function	Visual Basic .NET method	Description
Abs	Math.Abs Method	Returns the absolute value of a specified number.
Atn	Math.Atan Method	Returns a **Double** value containing the angle whose tangent is the specified number.
Cos	Math.Cos Method	Returns a **Double** value containing the cosine of the specified angle.
Exp	Math.Exp Method	Returns a **Double** value containing e (the base of natural logarithms) raised to the specified power.
Log	Math.Log Method	Returns a **Double** value containing the logarithm of a specified number. This method is overloaded and can return either the natural (base e) logarithm of a specified number or the logarithm of a specified number in a specified base.
Round	Math.Round Method	Returns a **Double** value containing the number nearest the specified value. Additional round functions are available as methods of the intrinsic types such as Decimal.Round Method.
Sgn	Math.Sign Method	Returns an **Integer** value indicating the sign of a number.
Sin	Math.Sin Method	Returns a **Double** value specifying the sine of an angle.
Sqr	Math.Sqrt Method	Returns a **Double** value specifying the square root of a number.
Tan	Math.Tan Method	Returns a **Double** value containing the tangent of an angle.

In addition, the .NET Framework math class provides constants and other static methods for trigonometric, logarithmic, and other common mathematical functions. All of these can be used in a Visual Basic program.

To use these functions without qualification, import the **System.Math** namespace into your project by adding the following code to the top of the source code:

```
Imports System.Math
```

Requirements

Class: Math Class

Abs Example

This example uses the **Abs** method of the **Math** class to compute the absolute value of a number.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim MyNumber As Double
```

```
MyNumber = Abs(50.3)    ' Returns 50.3.
MyNumber = Abs(-50.3)   ' Returns 50.3.
```

Atan Example

This example uses the **Atan** method of the **Math** class to calculate the value of pi.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim pi As Double
pi = 4 * Atan(1)   ' Calculate the value of pi.
```

Cos Example

This example uses the **Cos** method of the **Math** class to return the cosine of an angle.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim MyAngle, MySecant As Double
MyAngle = 1.3   ' Define angle in radians.
MySecant = 1 / Cos(MyAngle)   ' Calculate secant.
```

Exp Example

This example uses the **Exp** method of the **Math** class to return *e* raised to a power.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim MyAngle, MyHSin As Double
' Define angle in radians.
MyAngle = 1.3
' Calculate hyperbolic sine.
MyHSin = (Exp(MyAngle) - Exp(-1 * MyAngle)) / 2
```

Log Example

This example uses the **Log** method of the **Math** class to return the natural logarithm of a number.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim MyAngle, MyLog As Double
' Define angle in radians.
MyAngle = 1.3
' Calculate inverse hyperbolic sine.
MyLog = Log(MyAngle + Sqrt(MyAngle * MyAngle + 1))
```

Round Example

This example uses the **Round** method of the **Math** class to round a number to the nearest integer.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim MyVar1 As Double = 2.8
Dim MyVar2 As Double
MyVar2 =Round(MyVar1)   ' Returns 3.
```

Sign Example

This example uses the **Sign** method of the **Math** class to determine the sign of a number.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim MyVar1, MyVar2, MyVar3 As Double
Dim MySign As Integer
MyVar1 = 12
MyVar2 = -2.4
MyVar3 = 0
MySign = Sign(MyVar1)   ' Returns 1.
MySign = Sign(MyVar2)   ' Returns -1.
MySign = Sign(MyVar3)   ' Returns 0.
```

Sin Example

This example uses the **Sin** method of the **Math** class to return the sine of an angle.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim MyAngle, MyCosecant As Double
MyAngle = 1.3   ' Define angle in radians.
MyCosecant = 1 / Sin(MyAngle)   ' Calculate cosecant.
```

Sqrt Example

This example uses the **Sqrt** method of the **Math** class to calculate the square root of a number.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim MySqr As Double
MySqr = Sqrt(4)    ' Returns 2.
MySqr = Sqrt(23)   ' Returns 4.79583152331272.
MySqr = Sqrt(0)    ' Returns 0.
MySqr = Sqrt(-4)   ' Returns NaN (not a number).
```

Tan Example

This example uses the **Tan** method of the **Math** class to return the tangent of an angle.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim MyAngle, MyCotangent As Double
MyAngle = 1.3   ' Define angle in radians.
MyCotangent = 1 / Tan(MyAngle)   ' Calculate cotangent.
```

Requirements

Namespace: System.Math

Assembly: mscorlib (in mscorlib.dll)

See Also

Rnd Function | Randomize Statement | Derived Math Functions | Double.NaN Field

Str Function

Returns a **String** representation of a number.

```
Public Shared Function Str(ByVal Number As Object) As String
```

Parameter

Number
Required. An **Object** containing any valid numeric expression.

Exceptions/Errors

Exception type	Error number	Condition
ArgumentNullException	5	*Number* is not specified.
InvalidCastException	5	*Number* is not a numeric type .

Remarks

When numbers are converted to strings, a leading space is always reserved for the sign of *Number*. If *Number* is positive, the returned string contains a leading space, and the plus sign is implied. A negative number will include the minus sign (-) and no leading space.

Use the **Format** function to convert numeric values you want formatted as dates, times, or currency or in other user-defined formats. Unlike the **Str** function, the **Format** function doesn't include a leading space for the sign of *Number*.

> **Note** The **Str** function recognizes only the period (**.**) as a valid decimal separator. If different decimal separators are used (for example, in international applications), use the **CStr** function to convert a number to a string.

Example

This example uses the **Str** function to return a **String** representation of a number. When a positive number is converted to a string, a leading space is always reserved for its sign.

```
Dim MyString As String
MyString = Str(459)   ' Returns " 459".
MyString = Str(-459.65)   ' Returns "-459.65".
MyString = Str(459.001)   ' Returns " 459.001".
```

Requirements

Namespace: Microsoft.VisualBasic

Module: Conversion

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Format Function | Type Conversion Functions | Val Function | ArgumentNullException Class | InvalidCastException Class

StrComp Function

Returns -1, 0, or 1, based on the result of a string comparison.

```
Public Shared Function StrComp( _
   ByVal String1 As String, _
   ByVal String2 As String, _
   <Microsoft.VisualBasic.OptionCompareAttribute> _
   Optional ByVal Compare As Microsoft.VisualBasic.CompareMethod _
) As Integer
```

Parameters

String1
Required. Any valid **String** expression.
String2
Required. Any valid **String** expression.
Compare
Optional. Specifies the type of string comparison. If *compare* is omitted, the **Option Compare** setting determines the type of comparison.

Settings

The *Compare* argument settings are:

Constant	Description
Binary	Performs a binary comparison, based on a sort order derived from the internal binary representations of the characters.
Text	Performs a text comparison, based on a case-insensitive text sort order determined by your system's **LocaleID** value.

Return Values

The **StrComp** function has the following return values:

If	StrComp returns
String1 sorts ahead of *String2*	-1
String1 is equal to *String2*	0
String1 sorts after *String2*	1

Exceptions/Errors

Exception type	Error number	Condition
ArgumentException	5	*Compare* value is invalid.

Remarks

The strings are compared by alphanumeric sort values beginning with the first character. For further information on binary comparisons, textual comparisons, and sort order, see Option Compare Statement.

Example

This example uses the **StrComp** function to return the results of a string comparison. If the third argument is omitted, the comparison type defined in the option compare statement or project defaults is performed.

```
Dim MyStr1, MyStr2 As String
Dim MyComp As Integer
MyStr1 = "ABCD"
MyStr2 = "abcd"   ' Defines variables.
```

```
' The two strings sort equally. Returns 0
MyComp = StrComp(MyStr1, MyStr2, CompareMethod.Text)
' MyStr1 sorts after MyStr2. Returns -1.
MyComp = StrComp(MyStr1, MyStr2, CompareMethod.Binary)
' MyStr2 sorts before MyStr1. Returns 1.
MyComp = StrComp(MyStr2, MyStr1)
```

Requirements

Namespace: Microsoft.VisualBasic

Module: Strings

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Strings | String Manipulation | InStr Function | ArgumentException Class

StrConv Function

Returns a string converted as specified.

```
Public Shared Function StrConv( _
   ByVal Str As String, _
   ByVal Conversion As Microsoft.VisualBasic.VbStrConv, _
   Optional ByVal LocaleID As Integer,
) As String
```

Parameters

Str
Required. **String** expression to be converted.
Conversion
Required. **Microsoft.VisualBasic.VbStrConv** member. The enumeration value specifying the type of conversion to perform.
LocaleID
Optional. The **LocaleID** value, if different from the system **LocaleID** value. (The system **LocaleID** value is the default.)

Settings

The *Conversion* argument settings are:

Enumeration member	Description
VbStrConv.None	Performs no conversion
VbStrConv.LinguisticCasing	Uses linguistic rules for casing, rather than File System (default). Valid with UpperCase and LowerCase only.
VbStrConv.UpperCase	Converts the string to uppercase characters.
VbStrConv.LowerCase	Converts the string to lowercase characters.
VbStrConv.ProperCase	Converts the first letter of every word in string to uppercase.
VbStrConv.Wide*	Converts narrow (half-width) characters in the string to wide (full-width) c haracters.
VbStrConv.Narrow*	Converts wide (full-width) characters in the string to narrow (half-width) c haracters.
VbStrConv.Katakana**	Converts Hiragana characters in the string to Katakana characters.
VbStrConv.Hiragana**	Converts Katakana characters in the string to Hiragana characters.
VbStrConv.SimplifiedChinese*	Converts Traditional Chinese characters to Simplified Chinese.
VbStrConv.TraditionalChinese*	Converts Simplified Chinese characters to Traditional Chinese.

* Applies to Asian locales.
** Applies to Japan only.

> **Note** These constants are specified in the .NET common language runtime. As a result, they may be used anywhere in your code in place of the actual values. Most can be combined, for example, `UpperCase + Wide`, except when they are mutually exclusive, for example, `VbStrConv.Wide + VbStrConv.Narrow`.

The following are valid word separators for proper casing: Null (**Chr$(0)**), horizontal tab (**Chr$(9)**), linefeed (**Chr$(10)**), vertical tab (**Chr$(11)**), form feed (**Chr$(12)**), carriage return (**Chr$(13)**), space (single-byte character set) (**Chr$(32)**). The actual value for a space, which can be full or half width, varies by country/region for East Asian cultures.

Exceptions/Errors

Exception type	Error number	Condition
ArgumentException	5	Unsupported *LocaleID, Conversion* < 0 or > 2048, or unsupported conversion for specif ied locale.

Remarks

The constants **VbStrConv.Wide**, **VbStrConv.Narrow**, **VbStrConv.Simplified Chinese**, **VbStrConv.Traditional Chinese**, **VbStrConv.Katakana**, and **VbStrConv.Hiragana** can cause run-time errors when used in locales where they do not apply, but not always: the constants **VbStrConv.Katakana** and **VbStrConv.Hiragana** can be used in a non-Japanese system with the Japanese Language Pack installed. In addition, use of the constants **VbStrConv.Wide** and **VbStrConv.Narrow** is supported on any system with an East Asian language installed.

Example

This example converts text into all lowercase letters.

```
Dim sText, sNewText As String
sText = "Hello World"
sNewText = StrConv(sText, VbStrConv.LowerCase)
Debug.WriteLine (sNewText)   ' Outputs "hello world".
```

Smart Device Developer Notes

This function is not supported.

Requirements

Namespace: Microsoft.VisualBasic

Module: Strings

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Chr, ChrW Functions | String Data Type | Type Conversion Functions | ArgumentException Class

StrDup Function

Returns a string or object consisting of the specified character repeated the specified number of times.

```
Public Shared Function StrDup( _
   ByVal Number As Integer, _
   ByVal Character As { Char | String } _
) As String
```

-or-

```
Public Shared Function StrDup( _
   ByVal Number As Integer, _
   ByVal Character As Object _
) As Object
```

Parameters

Number
Required. **Integer** expression. The length to the string to be returned.

Character
Required. Any valid **Char**, **String**, or **Object** expression. Only the first character of the expression will be used. If Character is of type Object, it must contain either a **Char** or a **String** value.

Exceptions/Errors

Exception type	Error number	Condition
ArgumentException	5	*Number* < 0 or *Character* type is invalid .
ArgumentNullException	5	*Character* is **Nothing.**

Remarks

This function returns a **String** made up of repeated characters. The character that makes up the string is the first character in the *Character* argument, and it is duplicated *Number* number of times.

Example

This example uses the **StrDup** function to return a string of duplicated characters.

```
Dim aString As String = "Wow! What a string!"
Dim aObject As New Object()
Dim myString As String
aObject = "This is a String contained within an Object"
myString = StrDup(5, "P")   ' Returns "PPPPP"
myString = StrDup(10, aString)   ' Returns "WWWWWWWWWW"
myString = StrDup(6, aObject)   ' Returns "TTTTTT"
```

Requirements

Namespace: Microsoft.VisualBasic

Module: Strings

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

SPC Function | ArgumentException Class | ArgumentNullException Class

StrReverse Function

Returns a string in which the character order of a specified string is reversed.

```
Public Function StrReverse(ByVal Expression As String) As String
```

Parameters

Expression
Required. String expression whose characters are to be reversed. If *Expression* is a zero-length string (""), a zero-length string is returned.

Example

```
Dim myString As String = "ABCDEFG"
Dim revString As String
' Returns "GFEDCBA"
revString = StrReverse(myString)
```

Requirements

Namespace: Microsoft.VisualBasic

Module: Strings

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

InStrRev Function

Switch Function

Evaluates a list of expressions and returns an **Object** value of an expression associated with the first expression in the list that is **True**.

```
Public Function Switch( _
   ByVal ParamArray VarExpr() As Object _
) As Object
```

Parameter

VarExpr()
Required. **Object** parameter array. Must have an even number of elements. You can supply a list of **Object** variables or expressions separated by commas, or a single-dimensional array of **Object** elements.

Exceptions/Errors

Exception type	Error number	Condition
ArgumentException	5	Number of arguments is odd.

Remarks

The **Switch** function argument *VarExpr* consists of paired expressions and values. The **Switch** function evaluates the expressions from lowest to highest subscript in *VarExpr*, and returns the value associated with the first expression that evaluates to **True**. For example, if *VarExpr(0)* is **True**, **Switch** returns *VarExpr(1)*, and if *VarExpr(0)* is **False** but *VarExpr(2)* is **True**, **Switch** returns *VarExpr(3)*, and so on.

If you do not supply the *VarExpr* argument, **Switch** returns **Nothing**. If the number of elements in *VarExpr* is not divisible by two, an **ArgumentException** error occurs.

> **Note** The expressions in the argument list can include function calls. As part of preparing the argument list for the call to **Switch**, the Visual Basic compiler calls every function in every expression. This means that you cannot rely on a particular function not being called if an expression earlier in the argument list is **True**.

Example

This example uses the **Switch** function to return the name of a language that matches the name of a city. It requires that **Option Strict** be **Off**.

```
Function MatchUp (CityName As String) As String
   Return Microsoft.VisualBasic.Switch(CityName = "London", "English", _
      CityName = "Rome", "Italian", CityName = "Paris", "French")
End Function
```

Since the **System.Diagnostics** namespace contains a class called **Switch**, a call to the **Switch** function must be qualified with the **Microsoft.VisualBasic** namespace.

Requirements

Namespace: Microsoft.VisualBasic

Module: Interaction

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Choose Function | IIf Function | Select...Case Statements | Switch Class

SYD Function

Returns a **Double** specifying the sum-of-years digits depreciation of an asset for a specified period.

```
Function SYD( _
   ByVal Cost As Double, _
   ByVal Salvage As Double, _
   ByVal Life As Double, _
   ByVal Period As Double _
) As Double
```

Parameters

Cost
Required. **Double** specifying initial cost of the asset.
Salvage
Required. **Double** specifying value of the asset at the end of its useful life.
Life
Required. **Double** specifying length of the useful life of the asset.
Period
Required. **Double** specifying period for which asset depreciation is calculated.

Exceptions/Errors

Exception type	Error number	Condition
ArgumentException	5	*Salvage* < 0, *Period* > *Life*, or *Period* <=0 .

Remarks

The *Life* and *Period* arguments must be expressed in the same units. For example, if *Life* is given in months, *Period* must also be given in months. All arguments must be positive numbers.

Example

This example uses the **SYD** function to return the depreciation of an asset for a specified period given the asset's initial cost (`InitCost`), the salvage value at the end of the asset's useful life (`SalvageVal`), and the total life of the asset in years (`LifeTime`). The period in years for which the depreciation is calculated is `PDepr`.

```
Sub TestSYD()
   Dim InitCost, SalvageVal, MonthLife, DepYear, LifeTime, Pdepr As Double
   Dim Fmt As String = "###,##0.00"   ' Define money format.
   Const YEARMONTHS As Integer = 12  ' Number of months in a year.
   InitCost = CDbl(InputBox("What's the initial cost of the asset?"))
   SalvageVal = CDbl(InputBox("What's the asset's value at the end of its life?"))
   MonthLife = CDbl(InputBox("What's the asset's useful life in months?"))
   Do While MonthLife < YEARMONTHS   ' Ensure period is >= 1 year.
      MsgBox("Asset life must be a year or more.")
      MonthLife = CDbl(InputBox("What's the asset's useful life in months?"))
   Loop
   LifeTime = MonthLife / YEARMONTHS   ' Convert months to years.
   If LifeTime <> Int(MonthLife / YEARMONTHS) Then
      LifeTime = Int(LifeTime + 1)   ' Round up to nearest year.
   End If
   DepYear = CInt(InputBox("For which year do you want depreciation?"))
   Do While DepYear < 1 Or DepYear > LifeTime
      MsgBox("You must enter at least 1 but not more than " & LifeTime)
      DepYear = CInt(InputBox("For what year do you want depreciation?"))
   Loop
   Pdepr = SYD(InitCost, SalvageVal, LifeTime, DepYear)
   MsgBox("The depreciation for year " & DepYear & " is " & Format(Pdepr, Fmt) & ".")
End Sub
```

Requirements

Namespace: Microsoft.VisualBasic

Module: Financial

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

DDB Function | FV Function | IPmt Function | IRR Function | MIRR Function | NPer Function | NPV Function | Pmt Function | PPmt Function | PV Function | Rate Function | SLN Function | ArgumentException Class

SystemTypeName Function

Returns a **String** value containing the system data type name of a variable.

```
Public Function SystemTypeName(ByVal VbName As String) As String
```

Parameter

VbName
Required. A **String** variable containing a Visual Basic .NET type name.

Remarks

SystemTypeName returns the fully qualified common language runtime type name corresponding to the Visual Basic type name. For example, if *VbName* contains "Date", **SystemTypeName** returns "System.DateTime". If **SystemTypeName** does not recognize the value of *VbName*, it returns **Nothing** (not the string "Nothing").

Example

This example uses the **SystemTypeName** function to return data type names for several variables.

```
Dim VbLongName As String = "Long"
Dim VbDateName As String = "Date"
Dim VbBadName As String = "Number"
Dim MySysName As String
MySysName = SystemTypeName(VbLongName)   ' Returns "System.Int64".
MySysName = SystemTypeName(VbDateName)   ' Returns "System.DateTime".
MySysName = SystemTypeName(VbBadName)   ' Returns Nothing.
```

Requirements

Namespace: Microsoft.VisualBasic

Module: Information

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Data Type Summary | String Data Type | VbTypeName Function

TAB Function

Used with the **Print** or **PrintLine** functions to position output.

```
Public Overloads Function TAB() As TABInfo
```

-or-

```
Public Overloads Function TAB(ByVal Column As Short) As TABInfo
```

Parameters

Column
Optional. The column number moved to before displaying or printing the next expression in a list. If omitted, **TAB** moves the insertion point to the beginning of the next print zone.

Remarks

If the current print position on the current line is greater than *Column*, **TAB** skips to the column value equal to *Column* on the next output line. If *Column* is less than 1, **TAB** moves the print position to column 1. If *Column* is greater than the output line width, **TAB** calculates the next print position using the formula:

```
Column Mod width
```

For example, if *width* is 80 and you specify **TAB(**90**)**, the next print will start at column 10 (the remainder of 90/80). If *Column* is less than the current print position, printing begins on the next line at the calculated print position. If the calculated print position is greater than the current print position, printing begins at the calculated print position on the same line.

The leftmost print position on an output line is always 1. When you use the **Print** or **PrintLine** functions to print to files, the rightmost print position is the current width of the output file, which you can set using the **FileWidth** function.

The **TAB** function can also be used with the **WriteLine** function. It cannot be used with **System.Debug.WriteLine** or **System.Console.WriteLine**.

> **Note** Make sure your tabular columns are wide enough to accommodate wide letters.

Example

This example uses the **TAB** function to position output in a file and in the **Output** window.

```
FileOpen(1, "TESTFILE", OpenMode.Output) ' Open file for output.
' The second word prints at column 20.
Print(1, "Hello", TAB(20), "World.")
' If the argument is omitted, cursor is moved to the next print zone.
Print(1, "Hello", TAB(), "World")
FileClose(1)
```

Smart Device Developer Notes

This function is not supported.

Requirements

Namespace: Microsoft.VisualBasic

Module: FileSystem

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Math Functions

The math functions in Visual Basic 6 have been replaced by equivalent methods in the **System.Math** class of the .NET Framework.

Remarks

The .NET Framework math methods are functionally identical to their Visual Basic 6 counterparts, although some have slightly different names. For example, the .NET Framework equivalent of the Visual Basic 6 Atn function is **Atan**. The following table lists the Visual Basic 6 math function names and the equivalent .NET Framework methods.

Visual Basic 6 function	Visual Basic .NET method	Description
Abs	Math.Abs Method	Returns the absolute value of a specified number.
Atn	Math.Atan Method	Returns a **Double** value containing the angle whose tangent is the specified number.
Cos	Math.Cos Method	Returns a **Double** value containing the cosine of the specified angle.
Exp	Math.Exp Method	Returns a **Double** value containing e (the base of natural logarithms) raised to the specified power.
Log	Math.Log Method	Returns a **Double** value containing the logarithm of a specified number. This method is overloaded and can return either the natural (base e) logarithm of a specified number or the logarithm of a specified number in a specified base.
Round	Math.Round Method	Returns a **Double** value containing the number nearest the specified value. Additional round functions are available as methods of the intrinsic types such as Decimal.Round Method.
Sgn	Math.Sign Method	Returns an **Integer** value indicating the sign of a number.
Sin	Math.Sin Method	Returns a **Double** value specifying the sine of an angle.
Sqr	Math.Sqrt Method	Returns a **Double** value specifying the square root of a number.
Tan	Math.Tan Method	Returns a **Double** value containing the tangent of an angle.

In addition, the .NET Framework math class provides constants and other static methods for trigonometric, logarithmic, and other common mathematical functions. All of these can be used in a Visual Basic program.

To use these functions without qualification, import the **System.Math** namespace into your project by adding the following code to the top of the source code:

```
Imports System.Math
```

Requirements

Class: Math Class

Abs Example

This example uses the **Abs** method of the **Math** class to compute the absolute value of a number.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim MyNumber As Double
```

```
MyNumber = Abs(50.3)    ' Returns 50.3.
MyNumber = Abs(-50.3)   ' Returns 50.3.
```

Atan Example

This example uses the **Atan** method of the **Math** class to calculate the value of pi.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim pi As Double
pi = 4 * Atan(1)   ' Calculate the value of pi.
```

Cos Example

This example uses the **Cos** method of the **Math** class to return the cosine of an angle.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim MyAngle, MySecant As Double
MyAngle = 1.3   ' Define angle in radians.
MySecant = 1 / Cos(MyAngle)   ' Calculate secant.
```

Exp Example

This example uses the **Exp** method of the **Math** class to return *e* raised to a power.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim MyAngle, MyHSin As Double
' Define angle in radians.
MyAngle = 1.3
' Calculate hyperbolic sine.
MyHSin = (Exp(MyAngle) - Exp(-1 * MyAngle)) / 2
```

Log Example

This example uses the **Log** method of the **Math** class to return the natural logarithm of a number.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim MyAngle, MyLog As Double
' Define angle in radians.
MyAngle = 1.3
' Calculate inverse hyperbolic sine.
MyLog = Log(MyAngle + Sqrt(MyAngle * MyAngle + 1))
```

Round Example

This example uses the **Round** method of the **Math** class to round a number to the nearest integer.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim MyVar1 As Double = 2.8
Dim MyVar2 As Double
MyVar2 =Round(MyVar1)   ' Returns 3.
```

Sign Example

This example uses the **Sign** method of the **Math** class to determine the sign of a number.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim MyVar1, MyVar2, MyVar3 As Double
Dim MySign As Integer
MyVar1 = 12
MyVar2 = -2.4
MyVar3 = 0
MySign = Sign(MyVar1)   ' Returns 1.
MySign = Sign(MyVar2)   ' Returns -1.
MySign = Sign(MyVar3)   ' Returns 0.
```

Sin Example

This example uses the **Sin** method of the **Math** class to return the sine of an angle.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim MyAngle, MyCosecant As Double
MyAngle = 1.3   ' Define angle in radians.
MyCosecant = 1 / Sin(MyAngle)   ' Calculate cosecant.
```

Sqrt Example

This example uses the **Sqrt** method of the **Math** class to calculate the square root of a number.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim MySqr As Double
MySqr = Sqrt(4)    ' Returns 2.
MySqr = Sqrt(23)   ' Returns 4.79583152331272.
MySqr = Sqrt(0)    ' Returns 0.
MySqr = Sqrt(-4)   ' Returns NaN (not a number).
```

Tan Example

This example uses the **Tan** method of the **Math** class to return the tangent of an angle.

```
Imports System.Math
' Code will not compile unless placed within a Sub or Function
Dim MyAngle, MyCotangent As Double
MyAngle = 1.3   ' Define angle in radians.
MyCotangent = 1 / Tan(MyAngle)   ' Calculate cotangent.
```

Requirements

Namespace: System.Math

Assembly: mscorlib (in mscorlib.dll)

See Also

Rnd Function | Randomize Statement | Derived Math Functions | Double.NaN Field

TimeSerial Function

Returns a **Date** value representing a specified hour, minute, and second, with the date information set relative to January 1 of the year 1.

```
Public Function TimeSerial( _
   ByVal Hour As Integer, _
   ByVal Minute As Integer, _
   ByVal Second As Integer _
) As DateTime
```

Parameters

Hour
Required. **Integer** expression from 0 through 23. However, values outside this range are also accepted.

Minute
Required. **Integer** expression from 0 through 59. However, values outside this range are also accepted. The value of *Minute* is added to the calculated hour, so a negative value specifies minutes before that hour.

Second
Required. **Integer** expression from 0 through 59. However, values outside this range are also accepted. The value of *Second* is added to the calculated minute, so a negative value specifies seconds before that minute.

Exceptions/Errors

Exception type	Error number	Condition
ArgumentException	5	Argument is outside the range -2,147,483,648 through 2,147,483,647
ArgumentOutOfRangeException	9	Calculated time is less than negative 24 hours.

Remarks

The following example demonstrates negative, zero, and positive argument values. Here, the **TimeSerial** function returns a time representing 15 minutes before three hours before noon, or 8:45:00 A.M.:

```
Dim AlarmTime As Date = TimeSerial(12 - 3, -15, 0)
```

If either *Minute* or *Second* exceeds its normal range, it is applied to the next larger unit as appropriate. For example, if you specify 75 minutes, it is evaluated as one hour and 15 minutes. If any argument is outside the range -2,147,483,648 through 2,147,483,647, an **ArgumentException** error occurs.

TimeSerial reduces the total seconds modulo 86,400, which is the number of seconds in a day. Therefore, the returned time is always between 00:00:00 and 23:59:59.

The **Date** data type includes date components. **TimeSerial** sets all of these to 1, so the returned value represents the first day of the year 1. However, if the values of the arguments cause the calculated time to exceed 24 hours, the day is incremented as necessary. In the following example, the values of *Hour* and *Minute* result in a combined time of more than 24 hours.

```
MsgBox(TimeSerial(23, 75, 0))  ' Displays "1/2/0001 12:15:00 AM"
```

If the values of the arguments result in a negative calculated time, the date information is set to 1/1/0001 and the time information is adjusted to be between 00:00:00 and 23:59:59. However, if the calculated time is less than negative 24 hours, an **ArgumentOutOfRangeException** error occurs.

Since every **Date** value is supported by a **DateTime** structure, its methods give you additional options in assembling a **Date** value. For example, you can employ one of the overloaded **DateTime** constructors to populate a **Date** variable using the desired combination of components. The following example sets `NewDateTime` to May 6, 1978 at one tenth of a second before 8:30 in the morning:

```
Dim NewDateTime As Date = New Date(1978, 5, 6, 8, 29, 59, 900)
```

Example

This example uses the **TimeSerial** function to return a time for the specified hour, minute, and second.

```
Dim MyTime As Date
MyTime = TimeSerial(16, 35, 17)   ' Return the time in a Date data type.
```

Requirements

Namespace: Microsoft.VisualBasic

Module: DateAndTime

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

DateSerial Function | DateValue Function | Hour Function | Minute Function | Now Property | Second Function | TimeValue Function | System Namespace | DateTime Structure | ArgumentOutOfRangeException Class | ArgumentOutOfRangeException

TimeValue Function

Returns a **Date** value containing the time information represented by a string, with the date information set to January 1 of the year 1.

```
Public Function TimeValue(ByVal StringTime As String) As DateTime
```

Parameter

StringTime
Required. **String** expression representing a date/time value from 00:00:00 on January 1 of the year 1 through 23:59:59 on December 31, 9999.

Exceptions/Errors

Exception type	Error number	Condition
InvalidCastException	13	*StringTime* includes invalid time informatio n.

Remarks

You can enter valid times using a 12-hour or 24-hour clock. For example, `"2:24 P.M."` and `"14:24"` are both valid *StringTime* arguments.

If the *StringTime* argument includes date information, **TimeValue** does not include it in the returned value. However, if *StringTime* includes invalid time information such as "January 32," an **InvalidCastException** error occurs.

Example

This example uses the **TimeValue** function to convert a string to a time. You can also use date literals to directly assign a time to a **Date** variable.

```
Dim MyTime As Date
MyTime = TimeValue("4:35:17 PM")   ' Return the time in a Date data type.
MyTime = TimeValue(#4:35:17 PM#)   ' Return the time in a Date data type.
```

Requirements

Namespace: Microsoft.VisualBasic

Module: DateAndTime

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

DateSerial Function | DateValue Function | Hour Function | Minute Function | Now Property | Second Function | TimeSerial Function | System Namespace | DateTime Structure | InvalidCastException Class

LTrim, RTrim, and Trim Functions

Returns a string containing a copy of a specified string with no leading spaces (**LTrim**), no trailing spaces (**RTrim**), or no leading or trailing spaces (**Trim**).

```
Public Shared Function LTrim(ByVal Str As String) As String
Public Shared Function RTrim(ByVal Str As String) As String
Public Shared Function Trim(ByVal Str As String) As String
```

Parameter

Str
Required. Any valid **String** expression.

Example

This example uses the **LTrim** function to strip leading spaces and the **RTrim** function to strip trailing spaces from a string variable. It uses the **Trim** function to strip both types of spaces.

```
Dim MyString, TrimString As String
MyString = "  <-Trim->  "   ' Initializes string.
TrimString = LTrim(MyString)   ' TrimString = "<-Trim->  ".
TrimString = RTrim(MyString)   ' TrimString = "  <-Trim->".
TrimString = LTrim(RTrim(MyString))   ' TrimString = "<-Trim->".
' Using the Trim function alone achieves the same result.
TrimString = Trim(MyString)   ' TrimString = "<-Trim->".
```

Requirements

Namespace: Microsoft.VisualBasic

Module: Strings

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Left Function | Right Function

TypeName Function

Returns a **String** value containing data type information about a variable.

```
Public Function TypeName(ByVal VarName As Object) As String
```

Parameter

VarName
Required. **Object** variable. If **Option Strict** is **Off**, you can pass a variable of any data type except a structure.

Remarks

The following table shows the **String** values returned by **TypeName** for different contents of *VarName*.

***VarName* contents**	**String returned**
16-bit **True** or **False** value type	"Boolean"
8-bit binary value type	"Byte"
16-bit character value type	"Char"
64-bit date and time value type	"Date"
Reference type indicating missing or nonexistent data	"DBNull"
128-bit fixed point numeric value type	"Decimal"
64-bit floating point numeric value type	"Double"
32-bit integer value type	"Integer"
Reference type pointing to an unspecialized object	"Object"
Reference type pointing to a specialized object created from class *<objectclass>*	"*<objectclass>*"
64-bit integer value type	"Long"
Reference type with no object currently assigned to it	"Nothing"
16-bit integer value type	"Short"
32-bit floating point numeric value type	"Single"
Reference type pointing to a string of 16-bit characters	"String"

If *VarName* is an array, the returned string can be any one of the strings in the preceding table with empty parentheses appended. For example, if *VarName* points to an array of integers, **TypeName** returns "Integer()".

When **TypeName** returns the name of a reference type such as a class, it only returns the simple name, not the qualified name. For example, if *VarName* points to an object of class **System.Drawing.Printing.PaperSource**, **TypeName** returns "PaperSource".

Example

This example uses the **TypeName** function to return data type information about several variables.

```
Dim MyType As String
Dim StrVar As String = "MyString"
Dim DecVar As Decimal
Dim IntVar, ArrayVar(5) As Integer
MyType = TypeName(StrVar)   ' Returns "String".
MyType = TypeName(IntVar)   ' Returns "Integer".
MyType = TypeName(ArrayVar)   ' Returns "Integer()".
```

Requirements

Namespace: Microsoft.VisualBasic

Module: Information

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Data Type Summary | IsArray Function | IsDate Function | IsDBNull Function | IsError Function | IsNothing Function | IsNumeric Function | IsReference Function | Object Data Type

UBound Function

Returns the highest available subscript for the indicated dimension of an array.

```
Public Function UBound( _
   ByVal Array As System.Array, _
   Optional ByVal Rank As Integer = 1 _
) As Integer
```

Parameters

Array
Required. Array of any data type. The array in which you want to find the highest possible subscript of a dimension.

Rank
Optional. **Integer**. The dimension for which the highest possible subscript is to be returned. Use 1 for the first dimension, 2 for the second, and so on. If *Rank* is omitted, 1 is assumed.

Exceptions/Errors

Exception type	Error number	Condition
ArgumentNullException	9	*Array* is **Nothing**
RankException	9	*Rank* is less than 1 or *Rank* is greater than the rank of array.

Remarks

Since array subscripts start at 0, the length of a dimension is greater by one than the highest available subscript for that dimension.

For an array with the following dimensions, **UBound** returns the values in the following table:

```
Dim A(100, 5, 4) As Byte
```

Call to UBound	Return value
`UBound(A, 1)`	100
`UBound(A, 2)`	5
`UBound(A, 3)`	4

You can use **UBound** to determine the total number of elements in an array, but you must adjust the value it returns to account for the fact that the subscripts start at 0. The following example calculates the total size of the array `A` in the preceding example:

```
Dim Total As Integer
Total = (UBound(A, 1) + 1) * (UBound(A, 2) + 1) * (UBound(A, 3) + 1)
```

The value calculated for `Total` is 3030, which is 101 * 6 * 5.

Example

This example uses the **UBound** function to determine the highest available subscript for the indicated dimension of an array.

```
Dim Highest, MyArray(10, 15, 20), AnyArray(6) as Integer
Highest = UBound(MyArray, 1)   ' Returns 10.
Highest = UBound(MyArray, 3)   ' Returns 20.
Highest = UBound(AnyArray)   ' Returns 6.
```

Requirements

Namespace: Microsoft.VisualBasic

Module: Information

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

LBound Function | Dim Statement | ReDim Statement | ArgumentException Class | RankException Class

UCase Function

Returns a string or character containing the specified string converted to uppercase.

```
Public Shared Function UCase(ByVal Value As Char) As Char
```

-or-

```
Public Shared Function UCase(ByVal Value As String) As String
```

Parameter

Value
Required. Any valid **String** or **Char** expression.

Remarks

Only lowercase letters are converted to uppercase; all uppercase letters and nonletter characters remain unchanged.

Example

This example uses the **UCase** function to return an uppercase version of a string.

```
Dim LowerCase, UpperCase As String
LowerCase = "Hello World 1234"   ' String to convert.
UpperCase = UCase(LowerCase)   ' Returns "HELLO WORLD 1234".
```

Requirements

Namespace: Microsoft.VisualBasic

Module: Strings

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Strings | String Manipulation | LCase Function

Lock, Unlock Functions

Controls access by other processes to all or part of a file opened using the **Open** function.

```
Public Overloads Sub Lock(ByVal FileNumber As Integer)
. . .
Public Overloads Sub Unlock(ByVal FileNumber As Integer)
```

-or-

```
Public Overloads Sub Lock(_
   ByVal FileNumber As Integer, _
   ByVal FromRecord As Long _
)
. . .
Public Overloads Sub Unlock( _
   ByVal FileNumber As Integer, _
   ByVal FromRecord As Long _
)
```

-or-

```
Public Overloads Sub Lock( _
   ByVal FileNumber As Integer, _
   ByVal FromRecord As Long, _
   ByVal ToRecord As Long _
)
. . .
Public Overloads Sub Unlock( _
   ByVal FileNumber As Integer, _
   ByVal FromRecord As Long, _
   ByVal ToRecord As Long _
)
```

Parameters

FileNumber
Required. Any valid file number.
FromRecord
Optional. Number of the first record or byte to lock or unlock.
ToRecord
Optional. Number of the last record or byte to lock or unlock.

Exceptions/Errors

Exception type	Error number	Condition
IOException	52	*FileNumber* does not exist.
IOException	54	File mode is invalid.

Remarks

The **Lock** and **Unlock** functions are used in environments where several processes might need access to the same file.

Lock and **Unlock** functions are always used in pairs. The arguments to **Lock** and **Unlock** must match exactly.

If *FromRecord* and *ToRecord* are not supplied, the lock will be for the entire file. If *FromRecord* is specified but not *ToRecord*, the single record will be locked/unlocked.

If the file has been opened for sequential input or output, **Lock** and **Unlock** affect the entire file, regardless of the range specified by *FromRecord* and *ToRecord*.

Example

This example illustrates the use of the **Lock** and **Unlock** functions. This example assumes that `people.txt` is a file containing records of the structure `Person`.

```
Structure Person
   Dim Name As String
   Dim ID As Integer
End Structure

Sub PutInLockedFile(ByVal index As Integer, ByVal onePerson As Person)
   Try
      FileOpen(1, "c:\people.txt", OpenMode.Binary)
      Lock(1)
      FilePut(index, onePerson)
      Unlock(1)
      FileClose(1)
   Catch
      ' Error recovery code here.
   End Try
End Sub
```

Smart Device Developer Notes

This function is not supported.

Requirements

Namespace: Microsoft.VisualBasic

Module: FileSystem

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

FileOpen Function | IOException Class

Val Function

Returns the numbers contained in a string as a numeric value of appropriate type.

```
Public Overloads Function Val(ByVal Expression As String) As Double
```

-or-

```
Public Overloads Function Val(ByVal Expression As Object) As Double
```

-or-

```
Public Overloads Function Val(ByVal Expression As Char) As Integer
```

Parameter

Expression
Required. Any valid **String** expression, **Object** variable, or **Char** value. If *Expression* is of type **Object**, its value must be convertible to **String** or an **ArgumentException** error occurs.

Exceptions/Errors

Exception type	Error number	Condition
OverflowException	6	*InputStr* is too large.
InvalidCastException	13	Number is badly formed.
ArgumentException	438	**Object** type expression not convertible to **String**.

Remarks

The **Val** function stops reading the string at the first character it cannot recognize as part of a number. Symbols and characters that are often considered parts of numeric values, such as dollar signs and commas, are not recognized. However, the function recognizes the radix prefixes `&O` (for octal) and `&H` (for hexadecimal). Blanks, tabs, and linefeed characters are stripped from the argument.

The following returns the value 1615198:

```
Val(" 1615 198th Street N.E.")
```

In the code below, **Val** returns the decimal value -1 for the hexadecimal value shown:

```
Val("&HFFFF")
```

> **Note** The **Val** function recognizes only the period (**.**) as a valid decimal separator. When different decimal separators are used, as in international applications, use **CDbl** or **CInt** instead to convert a string to a number.

Example

This example uses the **Val** function to return the numbers contained in each string. **Val** stops converting at the first character that cannot be interpreted as a numeric digit, numeric modifier, numeric punctuation, or white space.

```
Dim ValResult As Double
ValResult = Val("2457")    ' ValResult is set to 2457.
ValResult = Val(" 2 45 7")    ' ValResult is set to 2457.
ValResult = Val("24 and 57")    ' ValResult is set to 24.
```

Requirements

Namespace: Microsoft.VisualBasic

Module: Conversion

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Str Function | Type Conversion Functions | OverflowException Class | InvalidCastException Class | ArgumentException Class

VarType Function

Returns an **Integer** value containing the data type classification of a variable.

```
Public Function VarType(ByVal VarName As Object) As VariantType
```

Parameter

VarName
Required. **Object** variable. If **Option Strict** is **Off**, you can pass a variable of any data type except a structure.

Remarks

The integer value returned by **VarType** is a member of the **VariantType** enumeration.

The following table shows the values returned by **VarType** for special cases of *VarName*.

Data type represented by *VarName*	**Value returned by *VarType***
Nothing	**VariantType.Object**
DBNull	**VariantType.Null**
Enumeration	Underlying data type (**Byte**, **Short**, **Integer**, or **Long**)
Array	Bitwise OR of array element type and **VariantType.Array**
Array of arrays	Bitwise OR of **VariantType.Object** and **VariantType.Array**
Structure (**System.ValueType**)	**VariantType.UserDefinedType**
System.Exception	**VariantType.Error**
Unknown	**VariantType.Object**

Example

This example uses the **VarType** function to return data type classification information about several variables.

```
Dim MyString As String = "MyString"
Dim MyObject As Object
Dim MyNumber, MyArray(5) As Integer
Dim MyVarType As VariantType   ' Integer enumeration.
MyVarType = VarType(MyVarType)   ' Returns VariantType.Integer.
MyVarType = VarType(MyString)   ' Returns VariantType.String.
MyVarType = VarType(MyObject)   ' Returns VariantType.Object.
MyVarType = VarType(MyArray)   ' Returns the bitwise OR of
                                     ' VariantType.Array and
                                     ' VariantType.Integer.
```

Smart Device Developer Notes

This function is not supported.

Requirements

Namespace: Microsoft.VisualBasic

Module: Information

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Data Type Summary | Object Data Type | VariantType Enumeration

VbTypeName Function

Returns a **String** value containing the Visual Basic data type name of a variable.

```
Public Function VbTypeName(ByVal UrtName As String) As String
```

Parameter

UrtName
Required. **String** variable containing a type name used by the common language runtime.

Remarks

VbTypeName returns the Visual Basic .NET type name corresponding to the common language runtime type name. For example, if *UrtName* contains "Int32" or "System.Int32", **VbTypeName** returns "Integer". If **VbTypeName** does not recognize the value of *UrtName*, it returns **Nothing** (not the string "Nothing").

The type name in *UrtName* can be obtained from various sources, for example from the **Name** property of the **System.Type** class.

Example

This example uses the **VbTypeName** function to return data type names for several variables.

```
Dim SysDateName As String = "System.DateTime"
Dim SysShortName As String = "Int16"
Dim SysBadName As String = "Nonsense"
Dim MyVbName As String
MyVbName = VbTypeName(SysDateName)   ' Returns "Date".
MyVbName = VbTypeName(SysShortName)   ' Returns "Short".
MyVbName = VbTypeName(SysBadName)   ' Returns Nothing.
```

Requirements

Namespace: Microsoft.VisualBasic

Module: Information

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Data Type Summary | String Data Type | Type Class | SystemTypeName Function

Weekday Function

Returns an **Integer** value containing a number representing the day of the week.

```
Public Function Weekday( _
   ByVal DateValue As DateTime, _
   Optional ByVal DayOfWeek As FirstDayOfWeek = FirstDayOfWeek.Sunday _
) As Integer
```

Parameters

DateValue
Required. **Date** value for which you want to determine the day of the week.

DayOfWeek
Optional. A value chosen from the **FirstDayOfWeek** enumeration that specifies the first day of the week. If not specified, **FirstDayOfWeek.Sunday** is used.

Settings

The *DayOfWeek* argument can have one of the following settings.

Enumeration value	Value	Description
FirstDayOfWeek.System	0	First day of week specified in system settings
FirstDayOfWeek.Sunday	1	Sunday (default)
FirstDayOfWeek.Monday	2	Monday (complies with ISO standard 8601, section 3.17)
FirstDayOfWeek.Tuesday	3	Tuesday
FirstDayOfWeek.Wednesday	4	Wednesday
FirstDayOfWeek.Thursday	5	Thursday
FirstDayOfWeek.Friday	6	Friday
FirstDayOfWeek.Saturday	7	Saturday

Exceptions/Errors

Exception type	Error number	Condition
ArgumentException	5	*DayOfWeek* is less than 0 or more than 7 .

Remarks

The value returned by the **Weekday** function corresponds to the values of the **FirstDayOfWeek** enumeration; that is, 1 indicates Sunday and 7 indicates Saturday.

If *DayOfWeek* is less than 0 or more than 7, an **ArgumentException** error occurs.

> **Note** **Weekday** uses the current calendar setting from the **CurrentCulture** property of the **CultureInfo** class in the **System.Globalization** namespace. The default **CurrentCulture** values are determined by **Control Panel** settings.

Example

This example uses the **Weekday** function to obtain the day of the week from a specified date.

```
Dim MyDate As Date
Dim MyWeekDay As Integer
MyDate = #2/12/1969#   ' Assign a date.
MyWeekDay = Weekday(MyDate)   ' MyWeekDay contains 4 because
   ' MyDate represents a Wednesday.
```

Requirements

Namespace: Microsoft.VisualBasic

Module: DateAndTime

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Day Function | Month Function | Now Property | WeekdayName Function | Year Function | DatePart Function | System Namespace | DateTime Structure | ArgumentException Class

WeekdayName Function

Returns a **String** value containing the name of the specified weekday.

```
Public Function WeekdayName( _
   ByVal WeekDay As Integer, _
   Optional ByVal Abbreviate As Boolean = False, _
   Optional ByVal FirstDayOfWeekValue As FirstDayOfWeek = FirstDayOfWeek.System _
) As String
```

Parameters

WeekDay
Required. **Integer**. The numeric designation for the weekday, from 1 through 7; 1 indicates the first day of the week and 7 indicates the last day of the week. The identities of the first and last days depend on the setting of *FirstDayOfWeekValue*.

Abbreviate
Optional. **Boolean** value that indicates if the weekday name is to be abbreviated. If omitted, the default is **False**, which means the weekday name is not abbreviated.

FirstDayOfWeekValue
Optional. A value chosen from the **FirstDayOfWeek** enumeration that specifies the first day of the week. If not specified, **FirstDayOfWeek.System** is used.

Settings

The *FirstDayOfWeekValue* argument can have one of the following settings.

Enumeration value	Value	Description
FirstDayOfWeek.System	0	First day of week specified in system settings (default)
FirstDayOfWeek.Sunday	1	Sunday
FirstDayOfWeek.Monday	2	Monday (complies with ISO standard 8601, section 3.17)
FirstDayOfWeek.Tuesday	3	Tuesday
FirstDayOfWeek.Wednesday	4	Wednesday
FirstDayOfWeek.Thursday	5	Thursday
FirstDayOfWeek.Friday	6	Friday
FirstDayOfWeek.Saturday	7	Saturday

Exceptions/Errors

Exception type	Error number	Condition
ArgumentException	5	*WeekDay* is less than 1 or greater than 7, o r *FirstDayOfWeekValue* is less than 0 or gr eater than 7.

Remarks

The string returned by **WeekdayName** depends not only on the input arguments, but also on the Regional Options settings specified in the Windows Control Panel.

If *WeekDay* is less than 1 or more than 7, or if *FirstDayOfWeekValue* is less than 0 or more than 7, an **ArgumentException** error occurs.

> **Note** **WeekdayName** uses the current calendar setting from the **CurrentCulture** property of the **CultureInfo** class in the **System.Globalization** namespace. The default **CurrentCulture** values are determined by **Control Panel** settings.

Requirements

Namespace: Microsoft.VisualBasic

Module: DateAndTime

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

MonthName Function | System Namespace | System.Globalization Namespace | DateTime Structure | ArgumentException Class | CultureInfo Class

Write, WriteLine Functions

Writes data to a sequential file. Data written with **Write** is usually read from a file with **Input**.

```
Public Sub Write( _
   ByVal FileNumber As Integer, _
   ByVal ParamArray Output As Object _
)
```

-or-

```
Public Sub WriteLine( _
   ByVal FileNumber As Integer, _
   ByVal ParamArray Output() As Object _
)
```

Parameters

FileNumber
Required. An **Integer** expression containing any valid file number.
Output
Optional. One or more comma-delimited expressions to write to a file.

Exceptions/Errors

Exception type	Error number	Condition
IOException	52	*FileNumber* does not exist.
IOException	54	File mode is invalid.

Remarks

If you omit *Output*, a blank line is printed to the file. Multiple expressions can be separated with a comma.

Unlike the **Print** function, the **Write** function inserts commas between items and quotation marks around strings as they are written to the file. You don't have to put explicit delimiters in the list. When **Write** is used to write data to a file, only the following data formats are supported and several universal assumptions are followed so the data can always be read and correctly interpreted using **Input**, regardless of locale:

- Numeric data is always written using the period as the decimal separator.
- For **Boolean** data, either `#TRUE#` or `#FALSE#` is printed. The **True** and **False** keywords are not translated, regardless of locale.
- Date data is written to the file using the universal date format. When either the date or the time component is missing or zero, only the part provided gets written to the file.
- Nothing is written to the file if *Output* data is empty. However, for null data, `#NULL#` is written.
- For **Error** data, the output appears as `#ERROR errorcode#`. The **Error** keyword is not translated, regardless of locale.

WriteLine inserts a newline character (that is, a carriage return–linefeed, or `Chr(13) + Chr(10)`), after it has written the final character in *Output* to the file.

You can embed quotation marks in a string by using double quotation marks, or "". For example,

```
Dim x As String = "Double quotation marks aren't ""difficult"" to handle."
```

returns a string with the value of Double quotation marks aren't "difficult" to handle.

Writing to a file with the **Write** or **WriteLine** functions requires **Append** access from the **FileIOPermissionAccess** enumeration. For more information, see FileIOPermissionAccess Enumeration.

Example

This example uses the **Write** function to write raw data to a sequential file.

```
FileOpen(1, "TESTFILE", OpenMode.Output) ' Open file for output.
Write(1, "This is a test.")  ' Print text to file.
WriteLine(1)  ' Print blank line to file.
WriteLine(1, "Zone 1", TAB(), "Zone 2")   ' Print in two print zones.
WriteLine(1, "Hello", " ", "World")     ' Separate strings with space.
WriteLine(1, SPC(5), "5 leading spaces ")    ' Print five leading spaces.
WriteLine(1, TAB(10), "Hello")   ' Print word at column 10.

' Assign Boolean, Date, and Error values.
Dim aBool As Boolean
Dim aDate As DateTime
aBool = False
aDate = DateTime.Parse("February 12, 1969")

' Dates and Booleans are translated using locale settings of
' your system.
WriteLine(1, aBool, " is a Boolean value")
WriteLine(1, aDate, " is a date")
FileClose(1)   ' Close file.
```

Smart Device Developer Notes

This function is not supported.

Requirements

Namespace: Microsoft.VisualBasic

Module: FileSystem

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Input Function | FileOpen Function | Print, PrintLine Functions | File Access with Visual Basic Run-Time Functions| IOException Class

Year Function

Returns an **Integer** value from 1 through 9999 representing the year.

```
Public Function Year(ByVal DateValue As DateTime) As Integer
```

Parameter

DateValue
Required. **Date** value from which you want to extract the year.

You can also obtain the year by calling **DatePart** and specifying **DateInterval.Year** for the *Interval* argument.

Example

This example uses the **Year** function to obtain the year from a specified date. In the development environment, the date literal is displayed in short date format using the locale settings of your code.

```
Dim MyDate As Date
Dim MyYear As Integer
MyDate = #2/12/1969#   ' Assign a date.
MyYear = Year(MyDate)   ' MyYear contains 1969.
```

Requirements

Namespace: Microsoft.VisualBasic

Module: DateAndTime

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Day Function | Month Function | Now Property | Weekday Function | DatePart Function | System Namespace | DateTime Structure | ArgumentOutOfRangeException Class

Keywords

In This Section

A–E Keywords

F–O Keywords

P–Z Keywords

Related Sections

Visual Basic Language and Run-Time Reference

Visual Basic Language

A-E Keywords

The following table contains a listing of the Visual Basic .NET language keywords.

Alias	Ansi	As	Assembly
Auto	ByRef	ByVal	Case
Default	DirectCast	Each	Else
ElseIf	End	Error	Explicit

See Also

F–O Keywords | P–Z Keywords | Visual Basic Language and Run-Time Reference

Alias

The **Alias** keyword indicates that an external procedure has another name in its DLL.

The **Alias** keyword is used in this context:

Declare Statement

See Also

Visual Basic Language Keywords

Ansi

The **Ansi** keyword indicates that strings are converted to American National Standards Institute (ANSI) values regardless of the name of the method being declared.

The **Ansi** keyword is used in this context:

Declare Statement

Smart Device Developer Notes

This keyword is not supported.

See Also

Visual Basic Language Keywords

As

The **As** keyword introduces an **As** clause, which identifies a data type.

The **As** keyword is used in these contexts:

Const Statement

Declare Statement

Delegate Statement

Dim Statement

Enum Statement

Event Statement

For...Next Statements

For Each...Next Statements

Function Statement

Property Statement

Sub Statement

Try...Catch...Finally Statements

See Also

Visual Basic Language Keywords

Assembly

The **Assembly** keyword indicates that an attribute block at the beginning of a source file applies to the entire assembly and not only to the current module.

See Also

Application of Attributes

Auto

The **Auto** keyword indicates that strings are converted according to common language runtime rules based on the name of the method being declared.

The **Auto** keyword is used in this context:

Declare Statement

Smart Device Developer Notes

This keyword is not supported.

See Also

Visual Basic Language Keywords

ByRef

The **ByRef** keyword indicates that an argument is passed in such a way that the called procedure can change the value of a variable underlying the argument in the calling code.

The **ByRef** keyword is used in these contexts:

Declare Statement

Function Statement

Sub Statement

See Also

Visual Basic Language Keywords | Argument Passing ByVal and ByRef

ByVal

The **ByVal** keyword indicates that an argument is passed in such a way that the called procedure or property cannot change the value of a variable underlying the argument in the calling code.

The **ByVal** keyword is used in these contexts:

Declare Statement

Function Statement

Property Statement

Sub Statement

See Also

Visual Basic Language Keywords | Argument Passing ByVal and ByRef

Case

The **Case** keyword introduces a value or set of values against which the value of an expression is to be tested.

The **Case** keyword is used in this context:

Select...Case Statements

See Also

Visual Basic Language Keywords

Default

The **Default** keyword identifies a property as the default property of its class.

The **Default** keyword is used in this context:

Property Statement

See Also

Visual Basic Language Keywords

DirectCast

The **DirectCast** keyword introduces a type conversion operation. You use it the same way you use the **CType** keyword, as the following example shows:

```
Dim Q As Object = 2.37   ' Requires Option Strict to be Off.
Dim I As Integer = CType(Q, Integer)   ' Succeeds.
Dim J As Integer = DirectCast(Q, Integer)   ' Fails.
```

Both keywords take an expression to be converted as the first argument, and the type to convert it to as the second argument. Both conversions fail if there is no conversion defined between the data type of the expression and the data type specified as the second argument.

The difference between the two keywords is that **CType** succeeds as long as there is a valid conversion defined between the expression and the type, whereas **DirectCast** requires the run-time type of an object variable to be the same as the specified type. If the specified type and the run-time type of the expression are the same, however, the run-time performance of **DirectCast** is better than that of **CType**.

In the preceding example, the run-time type of `Q` is **Double**. **CType** succeeds because **Double** can be converted to **Integer**, but **DirectCast** fails because the run-time type of `Q` is not already **Integer**.

DirectCast throws an **InvalidCastException** error if the argument types do not match.

See Also

Implicit and Explicit Conversions | InvalidCastException Class

Each

The **Each** keyword specifies the loop variable to be used in a **For Each** loop.

The **Each** keyword is used in this context:

For Each...Next Statements

See Also

Visual Basic Language Keywords

Else

The **Else** keyword introduces a group of statements to be executed if no other conditional group of statements has been executed.

The **Else** keyword is used in these contexts:

If...Then...Else Statements

Select...Case Statements

See Also

Visual Basic Language Keywords

ElseIf

The **ElseIf** keyword introduces a condition to be tested if the previous conditional test has failed.

The **ElseIf** keyword is used in this context:

If...Then...Else Statements

See Also

Visual Basic Language Keywords

End

When used with an additional keyword, **End** indicates the end of the definition of a procedure or block.

```
End Class
End Enum
End Function
End Get
End If
End Interface
End Module
End Namespace
End Property
End Select
End Set
End Structure
End Sub
End SyncLock
End Try
End While
End With
```

Parts

End
Required. Terminates the definition of the procedure or block.

Class
Required to end a **Class** definition.

Enum
Required to end an **Enum** definition.

Function
Required to end a **Function** procedure definition. If execution encounters an **End Function** statement, control is returned to the calling code.

Get
Required to end a **Property** procedure definition to retrieve the property's value. If execution encounters an **End Get** statement, control is returned to the statement requesting the value.

If
Required to end the definition of the **If...Then...Else** statements.

Interface
Required to end an **Interface** definition.

Module
Required to end a **Module** definition.

Namespace
Required to end a **Namespace** definition.

Property
Required to end a **Property** definition.

Select
Required to end the definition of the **Select...Case** statements.

Set
Required to end a **Property** procedure definition to set the property's value. If execution encounters an **End Set** statement, control is returned to the statement setting the value.

Structure
Required to end a **Structure** definition.

Sub
Required to end a **Sub** procedure definition. If execution encounters an **End Sub** statement, control is returned to the calling code.

SyncLock
Required to end the definition of the **SyncLock...End SyncLock** statements.

Try
Required to end the definition of the **Try...Catch...Finally** statements.

While

Required to end the definition of the **While** statement.

With

Required to end the definition of the **With...End With** statements.

Remarks

The **End** statement, without an additional keyword, terminates execution immediately.

Smart Device Developer Notes

The **End** statement, without an additional keyword, is not supported.

See Also

End Statement | Class Statement | Enum Statement | Function Statement | If...Then...Else Statements | Interface Statement | Module Statement | Namespace Statement | Property Statement | Select...Case Statements | Structure Statement | Sub Statement | SyncLock Statement | Try...Catch...Finally Statements | While...End While Statements | With...End With Statements

Error

The **Error** keyword either simulates or responds to a run-time error.

The **Error** keyword is used in these contexts:

Error Statement

On Error Statement

See Also

Visual Basic Language Keywords | Exception Handling

Explicit

The **Explicit** keyword is used in this context:

Option Explicit Statement

See Also

Visual Basic Language Keywords

F-O Keywords

The following table contains a listing of the Visual Basic .NET language keywords.

False	For	Friend	Handles
In	Is	Lib	Loop
Me	Module	MustInherit	MustOverride
MyBase	MyClass	New	Next
Nothing	NotInheritable	NotOverridable	Off
On	Option	Optional	Overloads
Overridable	Overrides		

See Also

A–E Keywords | P–Z Keywords | Visual Basic Language and Run-Time Reference

False

The **False** keyword represents a Boolean value that fails a conditional test.

See Also

Boolean Data Type | True

For

The **For** keyword introduces a loop that is iterated with different values of a loop variable.

The **For** keyword is used in these contexts:

For...Next Statements

For Each...Next Statements

See Also

Visual Basic Language Keywords

Friend

The **Friend** keyword confers friend access on one or more declared programming elements. Friend elements are accessible from within the program that contains their declaration and from anywhere else in the same assembly.

The **Friend** keyword can be used in conjunction with the **Protected** keyword in the same declaration. This combination confers both friend and protected access on the declared elements, so they are accessible from the same assembly, from their own class, and from any derived classes.

The **Friend** keyword is used in these contexts:

Class Statement

Const Statement

Declare Statement

Delegate Statement

Dim Statement

Enum Statement

Event Statement

Function Statement

Interface Statement

Module Statement

Property Statement

Structure Statement

Sub Statement

See Also

Protected | Accessibility | Procedures | Structures: Your Own Data Types | Understanding Classes

Handles

The **Handles** keyword is used to declare that a procedure handles a specified event.

```
proceduredeclaration Handles event
```

Parts

proceduredeclaration
The **Sub** procedure declaration for the procedure that will handle the event.

event
The name of the event being handled. This event must be raised by either the base class for the current class, or by an object declared using the **WithEvents** keyword.

Remarks

Use the **Handles** keyword at the end of a procedure declaration to cause it to handle events raised by an object variable declared using the **WithEvents** keyword. The **Handles** keyword can also be used in a derived class to handle events from a base class.

Example

```
Public Class ContainerClass
   WithEvents Obj As New Class1()  ' Module or class level declaration.

   Public Class Class1
      Public Event Ev_Event()      ' Declare an event.
      Sub CauseSomeEvent()
         RaiseEvent Ev_Event()     ' Raise an event.
      End Sub
   End Class

   Sub EventHandler() Handles Obj.Ev_Event
      MsgBox("EventHandler caught event.")   ' Handle the event.
   End Sub

' Call the TestEvents procedure from an instance of the ContainerClass
' class to test the Ev_Event event and the event handler.
   Public Sub TestEvents()
      Obj.CauseSomeEvent()
   End Sub
End Class
```

The following example demonstrates how a derived class can use the **Handles** statement to handle an event from a base class.

```
Public Class BaseClass
   Event Ev1() ' Declare an event.
End Class
Class DerivedClass
   Inherits BaseClass
   Sub TestEvents() Handles MyBase.Ev1
      ' Add code to handle this event.
   End Sub
End Class
```

See Also

WithEvents | AddHandler Statement | RemoveHandler Statement | Events and Event Handlers

Implements

The **Implements** keyword indicates that a class or structure member is providing the implementation for a member defined in an interface. The **Implements** keyword is not the same as the **Implements** statement.

If a class or structure implements an interface, it must include the **Implements** statement immediately after the **Class** or **Structure** statement, and it must implement all the members defined by the interface.

The **Implements** keyword is used in these contexts:

Event Statement

Function Statement

Property Statement

Sub Statement

See Also

Implements Statement | Implements Keyword | Interface Statement | Class Statement | Structure Statement

In

The **In** keyword specifies the group the loop variable is to traverse in a **For Each** loop.

The **In** keyword is used in this context:

For Each...Next Statements

See Also

Visual Basic Language Keywords

Is

The **Is** keyword introduces an **Is** clause that makes a comparison.

The **Is** keyword is used in these contexts:

Is Operator

Select...Case Statements

See Also

Visual Basic Language Keywords

Lib

The **Lib** keyword identifies the library containing an external procedure.

The **Lib** keyword is used in this context:

Declare Statement

See Also

Visual Basic Language Keywords

Loop

The **Loop** keyword terminates a loop introduced with a **Do** statement.

The **Loop** keyword is used in this context:

Do...Loop Statements

See Also

Visual Basic Language Keywords

Me

The **Me** keyword provides a way to refer to the specific instance of a class or structure in which the code is currently executing. **Me** behaves like either an object variable or a structure variable referring to the current instance. Using **Me** is particularly useful for passing information about the currently executing instance of a class or structure to a procedure in another class, structure, or module. For example, suppose you have the following procedure in a module:

```
Sub ChangeFormColor(FormName As Form)
   Randomize()
   FormName.BackColor = Color.FromArgb(Rnd() * 256, Rnd() * 256, Rnd() * 256)
End Sub
```

You can call this procedure and pass the current instance of the **Form** class as an argument using the following statement:

```
ChangeFormColor(Me)
```

The **Me** keyword is used in these contexts:

Class Statement

Structure Statement

See Also

Object Variable Assignment | MyBase | MyClass

Module

The **Module** keyword indicates that an attribute block at the beginning of a source file applies only to the current module and not to the entire assembly. The **Module** keyword is not the same as the **Module** statement.

See Also

Application of Attributes | Module Statement

MustInherit

The **MustInherit** keyword specifies that a class cannot be instantiated and can be used only as a base class.

The **MustInherit** keyword is used in this context:

Class Statement

See Also

Visual Basic Language Keywords

MustOverride

The **MustOverride** keyword specifies that a property or procedure in a base class must be overridden in a derived class before it can be used.

The **MustOverride** keyword is used in these contexts:

Function Statement

Property Statement

Sub Statement

See Also

Visual Basic Language Keywords | Shadowing

MyBase

The **MyBase** keyword behaves like an object variable referring to the base class of the current instance of a class. **MyBase** is commonly used to access base class members that are overridden or shadowed in a derived class. In particular, **MyBase.New** is used to explicitly call a base class constructor from a derived class constructor.

It is invalid to use **MyBase** to call **MustOverride** base methods.

See Also

Inheritance Basics | Me | MyClass | New

MyClass

MyClass behaves like an object variable referring to the current instance of a class as originally implemented. **MyClass** is similar to **Me**, but all method calls on it are treated as if the method were **NotOverridable**. Therefore, the method being called is not affected by overriding in a derived class. The following example compares **Me** and **MyClass**.

```
Class BaseClass
   Public Overridable Sub MyMethod()
      MsgBox("Base class string")
   End Sub
   Public Sub UseMe()
      Me.MyMethod()   ' Use calling class's version, even if an override.
   End Sub
   Public Sub UseMyClass()
      MyClass.MyMethod()   ' Use this version and not any override.
   End Sub
End Class
Class DerivedClass : Inherits BaseClass
   Public Overrides Sub MyMethod()
      MsgBox("Derived class string")
   End Sub
End Class
Class TestClasses
   Sub StartHere()
      Dim TestObj As DerivedClass = New DerivedClass()
      TestObj.UseMe()   ' Displays "Derived class string".
      TestObj.UseMyClass()   ' Displays "Base class string".
   End Sub
End Class
```

Even though `DerivedClass` overrides `MyMethod`, the **MyClass** keyword in `UseMyClass` nullifies the effects of overriding, and the compiler resolves the call to the base class version of `MyMethod`.

MyClass cannot not be used inside a **Shared** method, but you can use it inside an instance method to access a shared member of a class.

See Also

Inheritance Basics | Me | MyBase

New

The **New** keyword introduces a **New** clause, which creates a new object instance. The **New** clause must specify a defined class from which the instance can be created. You can use **New** in a declaration statement or an assignment statement. When the statement is executed, it calls the constructor of the specified class, passing any arguments you have supplied:

```
Dim Obj As Object
Obj = New SomeClass("String required by constructor")
' ...
Dim MyLabel As New Label()
```

Since arrays are classes, **New** can create a new array instance:

```
Dim MyArray As Integer()
MyArray = New Integer() {0, 1, 2, 3}
```

The common language runtime throws an **OutOfMemoryException** error if there is insufficient memory to create the new instance.

The **New** keyword is used in this context:

Dim Statement

See Also

Visual Basic Language Keywords | Object Lifetime: How Objects Are Created and Destroyed | Using Constructors and Destructors | OutOfMemoryException Class

Next

The **Next** keyword either terminates a loop that is iterated with different values of a loop variable, or specifies that execution should continue with the statement following the statement that generated an error.

The **Next** keyword is used in these contexts:

For...Next Statements

For Each...Next Statements

On Error Statement

Resume Statement

See Also

Visual Basic Language Keywords

Nothing

The **Nothing** keyword represents the default value of any data type. Assigning **Nothing** to a variable sets it to the default value for its declared type. If that type contains variable members, they are all set to their default values. The following example illustrates this:

```
Public Structure MyStruct
   Public Name As String
   Public Number As Short
End Structure
Dim S As MyStruct, I As Integer, B As Boolean
S = Nothing   ' Sets S.Name to Nothing, S.Number to 0.
I = Nothing   ' Sets I to 0.
B = Nothing   ' Sets B to False.
```

If the variable is of a reference type — that is, an object variable — **Nothing** means the variable is not associated with any object. For example:

```
Dim MyObject As Object
MyObject = Nothing   ' No object currently referred to.
```

When you assign **Nothing** to an object variable, it no longer refers to any object instance. If the variable had previously referred to an instance, setting it to **Nothing** does not terminate the instance itself. The instance is terminated, and the memory and system resources associated with it are released, only after the garbage collector detects there are no active references remaining.

See Also

Object Lifetime: How Objects are Created and Destroyed | Dim Statement | Lifetime

NotInheritable

The **NotInheritable** keyword specifies that a class cannot be used as a base class.

The **NotInheritable** keyword is used in this context:

Class Statement

See Also

Visual Basic Language Keywords

NotOverridable

The **NotOverridable** keyword specifies that a property or procedure cannot be overridden in a derived class.

The **NotOverridable** keyword is used in these contexts:

Function Statement

Property Statement

Sub Statement

See Also

Visual Basic Language Keywords | Shadowing

Off

The **Off** keyword turns a compiler option off.

The **Off** keyword is used in these contexts:

Option Explicit Statement

Option Strict Statement

See Also

Visual Basic Language Keywords

On

The **On** keyword either introduces a response to a run-time error or turns a compiler option on.

The **On** keyword is used in these contexts:

On Error Statement

Option Explicit Statement

Option Strict Statement

See Also

Visual Basic Language Keywords

Option

The **Option** keyword is used in these contexts:

Option Compare Statement

Option Explicit Statement

Option Strict Statement

See Also

Visual Basic Language Keywords

Optional

The **Optional** keyword indicates that a procedure argument can be omitted when the procedure is called.

The **Optional** keyword is used in these contexts:

Declare Statement

Function Statement

Property Statement

Sub Statement

See Also

Visual Basic Language Keywords

Overloads

The **Overloads** keyword declares a property or method with the same name as an existing member, but with an argument list different from the original member.

Overloads can also be used to shadow an existing member, or set of overloaded members, in a base class. When you use **Overloads** in this way, you declare the property or method with the same name and the same argument list as the base class member, and you do not supply the **Shadows** keyword.

The **Overloads** keyword is used in these contexts:

Function Statement

Property Statement

Sub Statement

See Also

Shadows | Procedure Overloading

Overridable

The **Overridable** keyword specifies that a property or method can be overridden in a derived class.

The **Overridable** keyword is used in these contexts:

Function Statement

Property Statement

Sub Statement

See Also

Visual Basic Language Keywords | Shadowing

Overrides

The **Overrides** keyword specifies that a property or method overrides a member inherited from a base class.

The **Overrides** keyword is used in these contexts:

Function Statement

Property Statement

Sub Statement

See Also

Visual Basic Language Keywords | Shadowing

P-Z Keywords

The following table contains a listing of the Visual Basic .NET language keywords.

ParamArray	Preserve	Private	Protected
Public	ReadOnly	Resume	Shadows
Shared	Static	Step	Then
To	True	TypeOf	Unicode
Until	When	While	WithEvents
WriteOnly			

See Also

A–E Keywords | F–O Keywords | Visual Basic Language and Run-Time Reference

ParamArray

The **ParamArray** keyword indicates that a procedure argument is an optional array of elements of the specified type. **ParamArray** can be used only on the last argument of an argument list. It allows you to pass an arbitrary number of arguments to the procedure. A **ParamArray** argument is always passed using **ByVal**.

The **ParamArray** keyword is used in these contexts:

Declare Statement

Function Statement

Property Statement

Sub Statement

See Also

Visual Basic Language Keywords | Parameter Arrays

Preserve

The **Preserve** keyword prevents the contents of an array from being cleared when the array is redimensioned.

The **Preserve** keyword is used in this context:

ReDim Statement

See Also

Visual Basic Language Keywords

Private

The **Private** keyword confers private access on one or more declared programming elements. Private elements are accessible only from within their declaration context, including from members of any nested types, for example from within a nested procedure or from an assignment expression in a nested enumeration.

The **Private** keyword is used in these contexts:

Class Statement

Const Statement

Declare Statement

Delegate Statement

Dim Statement

Enum Statement

Event Statement

Function Statement

Interface Statement

Property Statement

Structure Statement

Sub Statement

See Also

Accessibility | Procedures | Structures: Your Own Data Types | Understanding Classes

Protected

The **Protected** keyword confers protected access on one or more declared programming elements. Protected elements are accessible only from within their own class or from a derived class. Protected access is not a superset of friend access.

The **Protected** keyword can be used in conjunction with the **Friend** keyword in the same declaration. This combination confers both friend and protected access on the declared elements, so they are accessible from the same assembly, from their own class, and from any derived classes.

The **Protected** keyword is used in these contexts:

Class Statement

Const Statement

Declare Statement

Delegate Statement

Dim Statement

Enum Statement

Event Statement

Function Statement

Interface Statement

Property Statement

Structure Statement

Sub Statement

See Also

Friend | Accessibility | Procedures | Structures: Your Own Data Types | Understanding Classes

Public

The **Public** keyword confers public access on one or more declared programming elements. There are no restrictions on the accessibility of public elements.

The **Public** keyword is used in these contexts:

Class Statement

Const Statement

Declare Statement

Delegate Statement

Dim Statement

Enum Statement

Event Statement

Function Statement

Interface Statement

Module Statement

Property Statement

Structure Statement

Sub Statement

See Also

Accessibility | Procedures | Structures: Your Own Data Types | Understanding Classes

ReadOnly

The **ReadOnly** keyword indicates that a variable or property can be read but not written.

The **ReadOnly** keyword is used in these contexts:

Dim Statement

Property Statement

See Also

Visual Basic Language Keywords

Resume

The **Resume** keyword indicates where execution should continue after an error has been handled.

The **Resume** keyword is used in these contexts:

On Error Statement

Resume Statement

See Also

Visual Basic Language Keywords | Exception Handling

Shadows

The **Shadows** keyword indicates that a declared programming element shadows, or hides, an identically named element, or set of overloaded elements, in a base class. You can shadow any kind of declared element with any other kind.

A shadowed element is normally unavailable from within the derived class that shadows it. However, the following considerations apply:

- If the shadowing element is not accessible from the code referring to it, for example if it is **Private**, the reference is resolved to the shadowed element.
- If you shadow an element, you can still access the shadowed element through an object declared with the type of the base class.

The purpose of shadowing is to protect the definition of your class members. The base class might undergo a change that creates an element with the same name as one you have already defined. If this happens, the **Shadows** keyword forces references through your class to be resolved to your previously defined member, instead of to the new base class element.

The **Shadows** keyword is used in these contexts:

Class Statement

Const Statement

Declare Statement

Delegate Statement

Dim Statement

Enum Statement

Event Statement

Function Statement

Interface Statement

Property Statement

Structure Statement

Sub Statement

See Also

Shadowing | Private | Protected

Shared

The **Shared** keyword indicates that one or more declared programming elements are shared. Shared elements are not associated with a specific instance of a class or structure. You can access them by qualifying them either with the class or structure name, or with the variable name of a specific instance of the class or structure.

The **Shared** keyword is used in these contexts:

Dim Statement

Event Statement

Function Statement

Property Statement

Sub Statement

See Also

Lifetime | Procedures | Structures: Your Own Data Types | Understanding Classes

Static

The **Static** keyword indicates that one or more declared variables are static. Static variables remain in existence and retain their latest values after termination of the procedure in which they are declared.

The **Static** keyword is used in this context:

Dim Statement

See Also

Lifetime | Variable Declaration | Structures: Your Own Data Types | Understanding Classes

Step

The **Step** keyword specifies an increment value for a loop counter.

The **Step** keyword is used in this context:

For...Next Statements

See Also

Visual Basic Language Keywords

Then

The **Then** keyword introduces a statement block to be compiled or executed if a tested condition is true.

The **Then** keyword is used in these contexts:

#If...Then...#Else Directives

If...Then...Else Statements

See Also

Visual Basic Language Keywords

To

The **To** keyword separates the beginning and ending values of a loop counter or of a value match range.

The **To** keyword is used in these contexts:

For...Next Statements

Select...Case Statements

See Also

Visual Basic Language Keywords

True

The **True** keyword represents a Boolean value that passes a conditional test.

See Also

Boolean Data Type | False

TypeOf

The **TypeOf** keyword introduces a comparison clause that tests whether an object is derived from or implements a particular type, such as an interface.

The **TypeOf** keyword is used in this context:

If...Then...Else Statements

See Also

Comparison Operators | Object Variable Values | Determining Object Type

Unicode

The **Unicode** keyword indicates that strings are converted to Unicode values regardless of the name of the method being declared.

The **Unicode** keyword is used in this context:

Declare Statement

Smart Device Developer Notes

This keyword is not supported.

See Also

Visual Basic Language Keywords

Until

The **Until** keyword specifies a condition that terminates the execution of a **Do** loop.

The **Until** keyword is used in this context:

Do...Loop Statements

See Also

Visual Basic Language Keywords

When

The **When** keyword adds a conditional test to a **Catch** statement.

The **When** keyword is used in this context:

Try...Catch...Finally Statements

See Also

Visual Basic Language Keywords

While

The **While** keyword specifies a condition that allows the execution of a **Do** loop to continue. The **While** keyword is not the same as the opening statement of the **While...End While** block.

The **While** keyword is used in this context:

Do...Loop Statements

See Also

While...End While Statements

WithEvents

The **WithEvents** keyword indicates that a declared object variable refers to a class instance that can raise events.

The **WithEvents** keyword is used in this context:

Dim Statement

See Also

Visual Basic Language Keywords | Events and Delegates | WithEvents and the Handles Clause

WriteOnly

The **WriteOnly** keyword indicates that a property can be written but not read.

The **WriteOnly** keyword is used in this context:

Property Statement

See Also

Visual Basic Language Keywords

Methods

In This Section

Add Method

Clear Method

Raise Method

Remove Method

Related Sections

Visual Basic Language and Run-Time Reference

Visual Basic Language

Add Method

Adds a member to a **Collection** object.

```
Public Sub Add( _
   ByVal Item As Object, _
   Optional ByVal Key As String, _
   Optional ByVal { Before | After } As Object = Nothing _
)
```

Parameters

Item
Required. An object of any type that specifies the member to add to the collection.

Key
Optional. A unique **String** expression that specifies a key string that can be used instead of a positional index to access a member of the collection.

Before
Optional. An expression that specifies a relative position in the collection. The member to be added is placed in the collection before the member identified by the *Before* argument. If *Before* is a numeric expression, it must be a number from 1 to the value of the collection's **Count** property. If *Before* is a **String** expression, it must correspond to the key string specified when the member being referred to was added to the collection. You cannot specify both *Before* and *After*.

After
Optional. An expression that specifies a relative position in the collection. The member to be added is placed in the collection after the member identified by the *After* argument. If *After* is a numeric expression, it must be a number from 1 to the value of the collection's **Count** property. If *After* is a **String** expression, it must correspond to the key string specified when the member referred to was added to the collection. You cannot specify both *Before* and *After*.

Exceptions/Error Codes

Exception type	Error number	Condition
ArgumentException	5	Both *Before* and *After* are specified, or argument does not refer to an existing member of the collection.
ArgumentException	5	The specified *Key* already exists.

Remarks

The *Before* or *After* argument must refer to an existing member of the collection; otherwise, an error occurs.

An error also occurs if a specified *Key* value matches the key for an existing member of the collection.

Example

This example uses the **Add** method to add `Child` objects — instances of a class called `Child` containing a **Public** property `Name` — to a collection called `Family`. To see how this works, create a **Form** with two **Buttons** and set their **Text** properties to Add and List. Add the **Child** class definition and the **Family** declaration to the form code. Modify the **Click** events for the **Add** and **List** buttons as shown. The **Add** button allows you to add children. The **List** button will display the names of all the children.

```
Public Class Child
   Public Name As String
   Sub New(ByVal newName As String)
      Name = newName
   End Sub
End Class

Private family As New Collection() ' Create a Collection object.

Private Sub AddChild_Click(ByVal sender As System.Object, _
ByVal e As System.EventArgs) Handles Button1.Click
   Dim newName As String
```

```
    newName = InputBox("Name of new family member: ")
    If newName <> "" Then
       family.Add(New Child(newName), newName)
    End If
End Sub

Private Sub ListChild_Click(ByVal sender As System.Object, _
ByVal e As System.EventArgs) Handles Button2.Click
    Dim aChild As Child
    For Each aChild In family
       MsgBox(aChild.Name)
    Next
End Sub
```

Requirements

Namespace: Microsoft.VisualBasic

Module: Collection

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Item Property | Remove Method | ArgumentException Class

Applies To

Collection Object

Clear Method

Clears all property settings of the **Err** object.

```
object.Clear
```

Parameter

object

Always the **Err** object.

Remarks

Use **Clear** to explicitly clear the **Err** object after an error has been handled, such as when you use deferred error handling with **On Error Resume Next**. The **Clear** method is called automatically whenever any of the following statements executes:

- Any type of **Resume** statement
- **Exit Sub**, **Exit Function**, or **Exit Property**
- Any **On Error** statement
- Any **Try...Catch...Finally** statement

> **Note** The **On Error Resume Next** construct may be preferable to **On Error GoTo** when handling errors generated during access to other objects. Checking **Err** after each interaction with an object removes ambiguity about which object was accessed by the code: you can be sure which object placed the error code in **Err.Number**, as well as which object originally generated the error (the object specified in **Err.Source**).

Example

This example uses the **Err** object's **Clear** method to reset the numeric properties of the **Err** object to zero and its string properties to zero-length strings. Without the call to **Clear**, the second call to **MsgBox** would display the same error message.

```
Sub ClearErr()
   ' Produce overflow error
   On Error Resume Next
   Dim zero As Integer = 0
   Dim result As Integer = 8 / zero
   MsgBox(Err.Description)
   Err.Clear()
   MsgBox(Err.Description)
End Sub
```

Requirements

Namespace: Microsoft.VisualBasic

Class: Err Object

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Description Property | Err Object | HelpContext Property | HelpFile Property | LastDLLError Property | Number Property | On Error Statement | Raise Method | Source Property

Applies To

Err Object

GetEnumerator Method

Returns a reference to an **IEnumerator** interface, whose purpose is to grant access to an enumeration's items.

```
Public Function GetEnumerator() As IEnumerator
```

Remarks

When **GetEnumerator** is called, it constructs an enumerator object containing the current version number in the collection and a reference to the collection's items. Every time the enumerator is accessed, it compares the version of the enumerator with the version of the collection. If the versions do not match, it means that the collection has changed; an exception then occurs.

Example

The following example shows how to use **GetEnumerator** to retrieve a member of an enumeration.

```
Imports System.Collections
Public Class TestClass : Implements IEnumerable

Public Function GetEnumerator() As IEnumerator Implements _
IEnumerable.GetEnumerator
   Return CType(Me, IEnumerator)
End Function
```

Requirements

Namespace: Microsoft.VisualBasic

Module: Collection

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

IEnumerator Interface

Applies To

Collection Object

Raise Method

Generates a run-time error; can be used instead of the **Error** statement.

```
Public Sub Raise( _
   ByVal Number As Integer, _
   Optional ByVal Source As Object = Nothing, _
   Optional ByVal Description As Object = Nothing, _
   Optional ByVal HelpFile As Object = Nothing, _
   Optional ByVal HelpContext As Object = Nothing _
)
```

Parameter

Number
Required. **Long** integer that identifies the nature of the error. Visual Basic errors are in the range 0–65535: the range 0–512 is reserved for system errors; the range 513–65535 is available for user-defined errors. When setting the **Number** property to your own error code in a class module, you add your error code number to the **vbObjectError** constant. For example, to generate the error number 513, assign `vbObjectError + 513` to the **Number** property.

Source
Optional. **String** expression naming the object or application that generated the error. When setting this property for an object, use the form *project.class*. If *Source* is not specified, the process ID of the current Visual Basic project is used.

Description
Optional. **String** expression describing the error. If unspecified, the value in the **Number** property is examined. If it can be mapped to a Visual Basic run-time error code, the string that would be returned by the **Error** function is used as the **Description** property. If there is no Visual Basic error corresponding to the **Number** property, the "Application-defined or object-defined error" message is used.

HelpFile
Optional. The fully qualified path to the Help file in which help on this error can be found. If unspecified, Visual Basic uses the fully qualified drive, path, and file name of the Visual Basic Help file.

HelpContext
Optional. The context ID identifying a topic within *HelpFile* that provides help for the error. If omitted, the Visual Basic Help file context ID for the error corresponding to the **Number** property is used, if it exists.

Exceptions/Errors

Exception type	Error number	Condition
ArgumentException	5	*Number* is greater than 65535 .

Remarks

All of the **Raise** arguments except *Number* are optional. If you omit optional arguments, and the property settings of the **Err** object contain values that have not been cleared, those values serve as the values for your error.

Raise is useful for generating errors when writing class modules, because the **Err** object gives richer information than when you generate errors with the **Error** statement. For example, with the **Raise** method, the source that generated the error can be specified in the **Source** property, online Help for the error can be referenced, and so on.

Example

This example uses the **Err** object's **Raise** method to generate an error within a function written in Visual Basic. The calling function can catch the error and report it to the user with a message box.

```
Const WidthError = 1
Const WidthHelp = 101

Function TestWidth(ByVal width As Integer)
   If width > 1000 Then
      Err.Raise(vbObjectError + 512 + WidthError, "TestWidth", _
      "Width must be less than 1000.", "HelpFile.hlp", WidthHelp)
```

```
    End If
End Function

' Add to calling function.
Try
    TestWidth(2000)
Catch ex As Exception
    MsgBox(ex.Message)
End Try
```

Requirements

Namespace: Microsoft.VisualBasic

Class: Err Object

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Clear Method | Description Property | Err Object | Error Statement | HelpContext Property | HelpFile Property | LastDLLError Property | Number Property | On Error Statement | Source Property | ArgumentException Class

Applies To

Err Object

Remove Method

Removes a member from a **Collection** object.

```
Public Overloads Sub Remove(ByVal { Key As String | Index As Integer })
```

Parameters

Key
A unique **String** expression that specifies a key string that can be used, instead of a positional index, to access a member of the collection.

Index
An expression that specifies the position of a member of the collection. If a numeric expression, *Index* must be a number from 1 to the value of the collection's **Count** property. If a **String** expression, *Index* must correspond to the *Key* argument specified when the member referred to was added to the collection.

Exceptions/Errors

Exception type	Error number	Condition
ArgumentException	5	*Key* is invalid or not specified.
IndexOutOfRangeException	9	*Index* does not match an existing member of the collection.

Remarks

If *Index* doesn't match an existing member of the collection, an error occurs.

Example

This example illustrates the use of the **Remove** method to remove objects from a **Collection** object, `birthdays`.

```
Dim birthdays As New Collection()
birthdays.Add(New DateTime(2001, 1, 12), "Bill")
birthdays.Add(New DateTime(2001, 1, 13), "Joe")
birthdays.Remove(1)
birthdays.Remove("Joe")
```

Requirements

Namespace: Microsoft.VisualBasic

Module: Collection

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Add Method | ArgumentException Class | IndexOutOfRangeException Class

Applies to

Collection Object

Objects

In This Section

Collection Object

Collection Object Properties, Methods, and Events

Err Object

Err Object Properties, Methods, and Events

Related Sections

Visual Basic Language and Run-Time Reference

Visual Basic Language

Collection Object

A **Collection** object is an ordered set of items that can be referred to as a unit.

Remarks

The **Collection** object provides a convenient way to refer to a related group of items as a single object. The items, or members, in a collection need only be related by the fact that they exist in the collection. Members of a collection do not have to share the same data type.

You can create a collection the same way you create other objects. For example:

```
Dim X As New Collection
```

Once you have created a collection, you can add members with the **Add** method and remove them with the **Remove** method. You can return specific members from the collection with the **Item** method, and iterate through the entire collection with the **For Each...Next** statement.

> **Note** Although the **Collection** object in version 6 and in .NET have identical functionality, they cannot interoperate in a COM environment.
>
> **Caution** Enumerating through a collection is not a thread-safe procedure. Even if the collection is synchronized, other threads can still modify the collection, causing the enumerator to throw an exception. To guarantee thread safety during enumeration, either lock the collection or catch the exceptions resulting from changes made by other threads.

Example

This example creates a **Collection** object (`MyClasses`), and then creates a dialog box in which users can add objects to the collection. To see how this works, choose the **Class** command from the **Project** menu and declare a public variable called `InstanceName` at module level of `Class1` (type `Public InstanceName`) to hold the names of each instance. Leave the default name as `Class1`. Copy and paste the following code into the General section of another module, and then start it with the statement `ClassNamer` in another procedure. (This example only works with host applications that support classes.)

```
Public Class Class1
   Public InstanceName As String
End Class

Sub ClassNamer()
   Dim MyClasses As New Collection()   ' Create a Collection object.
   Dim number As Integer   ' Counter for individualizing keys.
   Dim Msg As String   ' Variable to hold prompt string.
   Dim name As String
   Dim oneInst As Class1
   Dim nameList As String = ""
   Do
      Dim inst As New Class1()   ' Create a new instance of Class1.
      number += 1   ' Increment Num, then get a name.
      Msg = "Please enter a name for this object." & ControlChars.CrLf _
         & "Press Cancel to see names in collection."
      name = InputBox(Msg, "Name the Collection Items")
      Inst.InstanceName = name   ' Put name in object instance.
      ' If user entered name, add it to the collection.
      If inst.InstanceName <> "" Then
         ' Add the named object to the collection.
         MyClasses.Add (inst, number.ToString())
      End If
   Loop Until name = ""

   For Each oneInst In MyClasses   ' Create list of names.
      nameList &= oneInst.InstanceName & ControlChars.CrLf
   Next

   ' Display the list of names in a message box.
   MsgBox (nameList, , "Instance Names In MyClasses Collection")
```

```
    Dim count As Integer
    For count = 1 To MyClasses.Count
       ' Remove name from the collection.
       MyClasses.Remove(1)
       ' Since collections are reindexed automatically, remove
       ' the first member on each iteration
    Next
End Sub
```

Requirements

Namespace: Microsoft.VisualBasic

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Collection Object Properties and Methods | For Each...Next Statements | GetEnumerator Method

Collection Object Properties and Methods

Properties

Count Property	Read-only. Returns an integer containing the number of objects in a collection.
Item Property	Returns a specific member of a **Collection** object either by position or by key.

Methods

Add Method	Adds a member to a **Collection** object.
GetEnumerator Method	Returns a reference to an **IEnumerator** interface, whose purpose is to grant access to an enumeration's items.
Remove Method	Removes a member from a **Collection** object.

See Also

Collection Object

Err Object

Contains information about run-time errors.

Remarks

The properties of the **Err** object are set by the generator of an error — Visual Basic, an object, or the programmer.

When a run-time error occurs, the properties of the **Err** object are filled with information that uniquely identifies the error, and information you can use to handle it. To generate a run-time error in your code, use the **Raise** method.

The **Err** object's properties are reset to zero or zero-length strings ("") after an **Exit Sub**, **Exit Function**, **Exit Property**, or **Resume Next** statement within an error-handling routine. Using any form of the **Resume** statement outside of an error-handling routine will not reset the **Err** object's properties. You can use the **Clear** method to explicitly reset **Err**.

Use the **Raise** method rather than the **Error** statement to generate run-time errors for system errors and class modules. Whether you use the **Raise** method in other code depends on the richness of the information you want to return.

The **Err** object is an intrinsic object with global scope. Therefore, you do not need to create an instance of it in your code.

Example

This example uses the properties of the **Err** object in constructing an error-message dialog box. Note that if you use the **Clear** method first, when you generate a Visual Basic error with the **Raise** method, Visual Basic's default values become the properties of the **Err** object.

```
Dim Msg As String
' If an error occurs, construct an error message.
On Error Resume Next   ' Defer error handling.
Err.Clear
Err.Raise(6)   ' Generate an "Overflow" error.
' Check for error, then show message.
If Err.Number <> 0 Then
   Msg = "Error # " & Str(Err.Number) & " was generated by " _
      & Err.Source & ControlChars.CrLf & Err.Description
   MsgBox(Msg, MsgBoxStyle.Information, "Error")
End If
```

Requirements

Namespace: **Microsoft.VisualBasic**

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Err Object Properties and Methods | Error Statement | On Error Statement | Exit Statement | Resume Statement | Visual Basic Run-Time Errors

Err Object Properties and Methods

Properties

Description Property	Read/write. Returns or sets a descriptive string associated with a n error.
Erl Property	Read-only. Returns an integer indicating the line number of the l ast executed statement.
HelpContext Property	Read/write. Returns or sets an **Integer** containing the context ID for a topic in a Help file.
HelpFile Property	Read/write. Returns or sets a **String** expression containing the f ully qualified path to a Help file.
LastDLLError Property	Read-only. Returns a system error code produced by a call to a dynamic-link library (DLL).
Number Property	Read/write. Returns or sets a numeric value specifying an error.
Source Property	Read/write. Returns or sets a **String** expression specifying the n ame of the object or application that originally generated the err or.

Methods

Clear Method	Clears all property settings of the **Err** object.
Raise Method	Generates a run-time error; can be used instead of the **Error** stat ement.

See Also

Err Object

Operators

In This Section

Operator Precedence in Visual Basic

Operators Listed by Functionality

Arithmetic Operators

Assignment Operators

Comparison Operators

Concatenation Operators

Logical/Bitwise Operators

Miscellaneous Operators

Related Sections

Visual Basic Language and Run-Time Reference

Visual Basic Language

Operator Precedence in Visual Basic

When several operations occur in an expression, each part is evaluated and resolved in a predetermined order called *operator precedence*.

When expressions contain operators from more than one category, they are evaluated according to the following rules:

- The arithmetic and concatenation operators have the order of precedence described below, and all have higher precedence than the comparison, logical, and bitwise operators.
- Comparison operators have equal precedence, and all have higher precedence than the logical and bitwise operators, but lower precedence than the arithmetic and concatenation operators.
- Logical and bitwise operators have the order of precedence described below, and all have lower precedence than the arithmetic, concatenation, and comparison operators.
- Operators with equal precedence are evaluated left to right in the order in which they appear in the expression.

Operators are evaluated in the following order of precedence:

Arithmetic and Concatenation Operators

Exponentiation (**^**)

Unary negation (**–**)

Multiplication and division (*****, **/**)

Integer division (****)

Modulus arithmetic (**Mod**)

Addition and subtraction (**+**, **–**), string concatenation (**+**)

String concatenation (**&**)

Arithmetic bit shift (**<<**, **>>**)

Comparison Operators

All comparison operators (**=**, **<>**, **<**, **<=**, **>**, **>=**, **Like**, **Is**, **TypeOf...Is**)

Logical and Bitwise Operators

Negation (**Not**)

Conjunction (**And**, **AndAlso**)

Disjunction (**Or**, **OrElse**, **Xor**)

The string concatenation operator (**&**) is not an arithmetic operator, but in precedence it is grouped with the arithmetic operators.

The **Is** operator is an object reference comparison operator. It does not compare objects or their values; it checks only to determine if two object variables refer to the same object.

When operators of equal precedence occur together in an expression, for example multiplication and division, each operation is evaluated as it occurs from left to right. Parentheses can be used to override the order of precedence and to force some parts of an expression to be evaluated before others. Operations within parentheses are always performed before those outside. Within parentheses, however, operator precedence is maintained. The following example illustrates this:

```
Dim A, B, C, D, E, F, G As Double
A = 3.0
B = 6.0
C = 4.0
D = 2.0
E = 1.0
F = A + B - C / D * E
' The previous line sets F to 7.0. Because of natural operator
' precedence, it is exactly equivalent to the following line:
F = (A + B) - ((C / D) * E)
```

```
' The following line overrides the natural operator precedence:
G = A + (B - C) / (D * E)
' The previous line sets G to 4.0.
```

See Also

= Operator | Is Operator | Like Operator | Operators Listed by Functionality | Operators

Operators Listed by Functionality

Operators	Description
Arithmetic Operators	These operators perform mathematical calculations.
Assignment Operators	These operators perform assignment operations.
Comparison Operators	These operators perform comparisons.
Concatenation Operators	These operators combine strings.
Logical/Bitwise Operators	These operators perform logical operations.
Bit Shift Operators	These operators perform arithmetic shifts on bit patterns.
Miscellaneous Operators	These operators perform miscellaneous operations.

See Also

Operators | Operator Precedence in Visual Basic

& Operator

Generates a string concatenation of two expressions.

```
result = expression1 & expression2
```

Parts

result
Required. Any **String** or **Object** variable.
expression1
Required. Any expression with a data type that widens to **String**.
expression2
Required. Any expression with a data type that widens to **String**.

Remarks

If the data type of *expression1* or *expression2* is not **String** but widens to **String**, it is converted to **String**. If one of the data types does not widen to **String**, the compiler generates an error.

The data type of *result* is **String**. If one or both expressions are stated as Nothing or have a value of **DBNull.value**, they are treated as a string with a value of "".

Example

This example uses the **&** operator to force string concatenation. The result is a string value representing the concatenation of the two string operands.

```
Dim myStr As String
myStr = "Hello" & " World"   ' Returns "Hello World".
```

See Also

&= Operator | Concatenation Operators | Operator Precedence in Visual Basic | Operators Listed by Functionality | Concatenation Operators

* Operator

Multiplies two numbers.

```
number1 * number2
```

Parts

number1
Required. Any numeric expression.
number2
Required. Any numeric expression.

Result

The result is the product of *number1* and *number2*

Supported Types

Byte, **Short**, **Integer**, **Long**, **Single**, **Double**, **Decimal**

Remarks

The data type of the result is the same as that of the data type with the greatest range. The order of range, from least to greatest range, is Byte, Short, Integer, Long, Single, Double, and Decimal.

If an expression is stated as Nothing, or is Empty, it is treated as zero.

Example

This example uses the ***** operator to multiply two numbers. The result is the product of the two operands.

```
Dim myValue As Double
myValue = 2 * 2   ' Returns 4.
myValue = 459.35 * 334.90   ' Returns 153836.315.
```

See Also

*= Operator | Arithmetic Operators | Operator Precedence in Visual Basic | Operators Listed by Functionality | Arithmetic Operators

*= Operator

Multiplies the value of a variable by the value of an expression and assigns the result to the variable.

```
variable *= expression
```

Parts

variable
Required. Any numeric variable.
expression
Required. Any numeric expression.

Example

The following example uses the ***=** operator to multiply one **Integer** variable by a second and assign the result to the first variable.

```
Dim var1 As Integer = 10
Dim var2 As Integer = 3
var1 *= var2   ' The value of var1 is now 30.
```

See Also

&= Operator | * Operator | += Operator | -= Operator | /= Operator | = Operator | \= Operator | ^= Operator | Assignment Operators | Arithmetic Operators | Operator Precedence in Visual Basic | Operators Listed by Functionality | Assignment Statements

+ Operator

Adds two numbers. Also used to concatenate two strings.

```
expression1 + expression2
```

Parts

expression1
Required. Any numeric expression or string.
expression2
Required. Any numeric expression or string.

Result

The value of the result will be the sum of *expression1* and *expression2* if *expression1* and *expression2* are numeric or the result of the concatenation of *expression1* and *expression2* if *expression1* and *expression2* are strings.

Supported Types

Byte, **Short**, **Integer**, **Long**, **Single**, **Double**, **Decimal**, **String**.

Remarks

When you use the **+** operator, you may not be able to determine whether addition or string concatenation will occur. Use the **&** operator for concatenation to eliminate ambiguity and provide self-documenting code.

If neither expression is an **Object**, the following rules apply:

If	**Then**
Both expressions are the same numeric data types (**Byte**, **Short**, **Integer**, **Long**, **Single**, **Double**, or **Decimal**)	Add.
Both expressions are strings	Concatenate.
One expression is a numeric data type and the other is a string	If **Option Strict** is **On**, a compile error is generated; if **Option Strict** is **Off**, implicitly convert **String** to **Double** and add. If **String** cannot be converted to a numeric value, an **InvalidCastException** is thrown.
One expression is a numeric data type, and the other is **Nothing**	If **Option Strict** is **On** a compile error is generated; if **Option Strict** is **Off**, add, with **Nothing** valued at zero.
One expression is a string, and the other is **Nothing**	If **Option Strict** is **On** a compile error is generated; if **Option Strict** is **Off**, concatenation, with **Nothing** = "".

If one expression is an **Object** expression, the following rules apply.

If	**Then**
One expression is a numeric **Object** expression and the other is a numeric value type	If **Option Strict** is **On**, a compiler error is generated; if **Option Strict** is **Off,** add.
One expression is a numeric **Object** expression and the other is a **String** type	If **Option Strict** is **On**, a compiler error is generated; if **Option Strict** is **Off**, implicitly convert **String** to **Double** and add. If **String** cannot be converted to a numeric value, an **InvalidCastException** is thrown.
One expression is a string **Object** expression and the other is a **String** type	If **Option Strict** is **On**, a compiler error is generated; if **Option Strict** is **Off**, implicitly convert **Object** to **String,** and then concatenate.
One expression is a string **Object** expression and the other is a numeric value type	If **Option Strict** is **On**, a compiler error is generated; if **Option Strict** is **Off**, implicitly convert **Object** to **Double** and add. If **Object** cannot be converted to a numeric value, an **InvalidCastException** is thrown.

If both expressions are **Object** expressions, the following rules apply (**Option Strict Off** only):

If	Then
Both **Object** expressions are numeric	Add.
Both **Object** expressions are strings	Concatenate.
One **Object** expression is numeric and the other is a string	Implicitly cast the numeric **Object** to **Double** and add. If **Object** cannot be converted to a numeric value, an **InvalidCastException** is thrown.

If one or both expressions are stated as Nothing or have a value of **DBNull**, they are treated as a string with a value of "".

Example

This example uses the **+** operator to add numbers. You can also use the **+** operator to concatenate strings. However, to eliminate ambiguity, you should use the **&** operator instead. If the components of an expression created with the **+** operator are numeric, the arithmetic result is assigned. If the components are exclusively strings, the strings are concatenated. The expression cannot contain components of mixed types. The arithmetic result returns the sum of the two operands. The concatenation result returns a string representing the concatenation of the two operands.

```
Dim myNumber As Integer
Dim var1 As String
Dim var2 As Integer
myNumber = 2 + 2   ' Returns 4.
myNumber = 4257.04 + 98112   ' Returns 102369.04.

Option Strict On
' Initialize mixed variables.
var1 = "34"
var2 = 6
myNumber = var1 + var2   ' Generates a compile-time error.

Option Strict Off
Var1 = "34"
Var2 = 6
myNumber = var1 + var2
' Returns 40 (addition) after the string in var1 is
' converted to a numeric value. Use of Option Strict Off
' for these operations is not recommended.
```

See Also

& Operator | Concatenation Operators | Arithmetic Operators | Operators Listed by Functionality | Operator Precedence in Visual Basic | Arithmetic Operators (Conceptual)

+= Operator

Adds the value of an expression to the value of a variable and assigns the result to the variable. Also concatenates a **String** expression to a **String** variable and assigns the result to the variable.

```
variable += expression
```

Parts

variable
Required. Any numeric or **String** variable.
expression
Required. Any numeric or **String** expression.

Remarks

This statement will implicitly perform widening but not narrowing conversions if the compilation environment is enforcing strict semantics. For details on these conversions, see Widening and Narrowing Conversions. If permissive semantics are allowed, the operator will implicitly perform a variety of string and numeric conversions. These conversions are identical to the conversions described for the **+** operator. For details on how these conversions are performed, see + Operator. For more information on strict and permissive semantics, see Option Strict Statement.

Example

These examples use the **+=** operator to combine the value of one variable with another. In the first example, **+=** is used with numeric variables to add one value to another. In the second example, **+=** is used with **String** variables to concatenate one value with another. In both cases, the result is assigned to the first variable.

```
Dim var1 As Integer = 10
Dim var2 As Integer = 3
var1 += var2   ' The value of var1 is now 13.

' This example uses string variables.
Dim var1 As String = "10"
Dim var2 As String = "3"
var1 += var2   ' The value of var1 is now "103".
```

See Also

+ Operator | -= Operator | Assignment Operators | Arithmetic Operators | Concatenation Operators | Operator Precedence in Visual Basic | Operators Listed by Functionality | Assignment Statements

= Operator

Used to assign a value to a variable or property.

```
variable = value
```

Parts

variable
Any variable or any writable property.
value
Any literal, constant, or expression.

Remarks

The name on the left side of the equal sign can be a simple scalar variable, a property, or an element of an array. Properties on the left side of the equal sign can only be those properties that are writable at run time. When used, the value on the right side of the equation is assigned to the variable on the left side of the equation.

Example

This example demonstrates use of the assignment operator. The value on the right side of the expression is assigned to the variable on the left side of the expression.

```
Dim myInt as Integer
Dim myString as String
Dim myButton as System.Windows.Forms.Button
Dim myObject as Object
myInt = 42
myString = "This is an example of a string literal"
myButton = New System.Windows.Forms.Button()
myObject = myInt
myObject = myString
myObject = myButton
```

See Also

&= Operator | *= Operator | += Operator | -= Operator | /= Operator | \= Operator | ^= Operator | Assignment Statements

- Operator

Yields the difference between two numbers or indicates the negative value of a numeric expression.

Syntax 1

```
expression1 - expression2
```

Syntax 2

```
-number
```

Parts

number
Required. Any numeric expression.
number1
Required. Any numeric expression.
number2
Required. Any numeric expression.

Result

The result will be the difference between *number1* and *number2*.

Supported Types

Byte, **Short**, **Integer**, **Long**, **Single**, **Double**, **Decimal**

Remarks

In Syntax 1, the **–** operator is the arithmetic subtraction operator for the difference between two numbers. In Syntax 2, the **–** operator is the unary negation operator for the negative value of an expression. The data type of the result is the same as that of the data type with the greatest range. The order of range, from least to greatest range, is Byte, Short, Integer, Long, Single, Double, and Decimal.

If an expression is stated as Nothing, it is treated as zero.

Example

This example uses the **-** operator to calculate and return the difference between two numbers.

```
Dim myResult As Double
myResult = 4 - 2   ' Returns 2.
myResult = 459.35 - 334.90   ' Returns 124.45.
```

See Also

-= Operator | Arithmetic Operators | Operator Precedence in Visual Basic | Operators Listed by Functionality | Arithmetic Operators

-= Operator

Subtracts the value of an expression from the value of a variable and assigns the result to the variable.

```
variable -= expression
```

Parts

variable
Required. Any numeric variable.
expression
Required. Any numeric expression.

Example

The following example uses the **-=** operator to subtract one **Integer** variable from another and assign the result to the latter variable.

```
Dim var1 As Integer = 10
Dim var2 As Integer = 3
var1 -= var2   ' The value of var1 is now 7.
```

See Also

&= Operator | *= Operator | += Operator | - Operator | /= Operator | = Operator | \= Operator | ^= Operator | Assignment Operators | Arithmetic Operators | Operator Precedence in Visual Basic | Operators Listed by Functionality | Assignment Statements

<< Operator

Performs an arithmetic left shift on a bit pattern.

```
result = pattern << amount
```

Parts

result
Required. Integral numeric value. The result of shifting the bit pattern. The data type is the same as that of *pattern*.
pattern
Required. Integral numeric expression. The bit pattern to be shifted. The data type must be **Byte**, **Short**, **Integer**, or **Long**.
amount
Required. Numeric expression. The number of bits to shift the bit pattern. The data type must be **Integer** or widen to **Integer**.

Remarks

Arithmetic shifts are not circular, which means the bits shifted off one end of the result are not reintroduced at the other end. In an arithmetic left shift, the bits shifted beyond the range of the result data type are discarded, and the bit positions vacated on the right are set to zero.

To prevent shifting by more bits than the result can hold, Visual Basic masks the value of *amount* with a size mask corresponding to the data type of *pattern*. The binary AND of these values is used for the shift amount. The size masks are as follows:

Data type of *pattern*	Size mask (decimal)	Size mask (hexadecimal)
Byte	7	&H00000007
Short	15	&H0000000F
Integer	31	&H0000001F
Long	63	&H0000003F

If *amount* is zero, the value of *result* is identical to the value of *pattern*. If *amount* is negative, it is taken as an unsigned value and masked with the appropriate size mask.

Arithmetic shifts never generate overflow exceptions.

Example

This example uses the **<<** operator to perform arithmetic left shifts on integral values. The result always has the same data type as that of the expression being shifted.

```
Dim Pattern As Short = 192   ' Bit pattern is 0000 0000 1100 0000.
Dim Result1, Result2, Result3, Result4, Result5 As Short
Result1 = Pattern << 0      ' Result is 192 (0000 0000 1100 0000).
Result2 = Pattern << 4     ' Result is 3072 (0000 1100 0000 0000).
Result3 = Pattern << 9   ' Result is -32768 (1000 0000 0000 0000).
Result4 = Pattern << 17     ' Result is 384 (0000 0001 1000 0000).
Result5 = Pattern << -1   ' Result is 0 (shifted 15 places to left).
```

The shift amount for `Result4` is calculated as 17 AND 15, which equals 1.

See Also

Bit Shift Operators | Assignment Operators | <<= Operator | Operator Precedence in Visual Basic | Operators Listed by Functionality | Arithmetic Operators

<<= Operator

Performs an arithmetic left shift on the value of a variable and assigns the result back to the variable.

```
variable <<= amount
```

Parts

variable
Required. Variable of data type **Byte**, **Short**, **Integer**, or **Long**.
amount
Required. Numeric expression of a data type that widens to **Integer**.

Remarks

Arithmetic shifts are not circular, which means the bits shifted off one end of the result are not reintroduced at the other end. In an arithmetic left shift, the bits shifted beyond the range of the result data type are discarded, and the bit positions vacated on the right are set to zero.

Example

The following example uses the **<<=** operator to shift the bit pattern of an **Integer** variable left by the specified amount and assign the result to the variable.

```
Dim Var As Integer = 10   ' Variable containing bit pattern to shift.
Dim Shift As Integer = 3   ' Amount to shift bit pattern.
Var <<= Shift   ' The value of Var is now 80.
```

See Also

<< Operator | Assignment Operators | Bit Shift Operators | Operator Precedence in Visual Basic | Operators Listed by Functionality | Assignment Statements

>> Operator

Performs an arithmetic right shift on a bit pattern.

```
result = pattern >> amount
```

Parts

result
Required. Integral numeric value. The result of shifting the bit pattern. The data type is the same as that of *pattern*.
pattern
Required. Integral numeric expression. The bit pattern to be shifted. The data type must be **Byte**, **Short**, **Integer**, or **Long**.
amount
Required. Numeric expression. The number of bits to shift the bit pattern. The data type must be **Integer** or widen to **Integer**.

Remarks

Arithmetic shifts are not circular, which means the bits shifted off one end of the result are not reintroduced at the other end. In an arithmetic right shift, the bits shifted beyond the rightmost bit position are discarded, and the leftmost (sign) bit is propagated into the bit positions vacated at the left. This means that if *pattern* has a negative value, the vacated positions are set to one; otherwise they are set to zero.

Note that data type **Byte** is considered unsigned, so there is no sign bit to propagate. If *pattern* is of type **Byte**, the vacated positions are always set to zero.

To prevent shifting by more bits than the result can hold, Visual Basic masks the value of *amount* with a size mask corresponding to the data type of *pattern*. The binary AND of these values is used for the shift amount. The size masks are as follows:

Data type of *pattern*	Size mask (decimal)	Size mask (hexadecimal)
Byte	7	&H00000007
Short	15	&H0000000F
Integer	31	&H0000001F
Long	63	&H0000003F

If *amount* is zero, the value of *result* is identical to the value of *pattern*. If *amount* is negative, it is taken as an unsigned value and masked with the appropriate size mask.

Arithmetic shifts never generate overflow exceptions.

Example

This example uses the **>>** operator to perform arithmetic right shifts on integral values. The result always has the same data type as that of the expression being shifted.

```
Dim Pattern As Short = 2560   ' Bit pattern is 0000 1010 0000 0000.
Dim Result1, Result2, Result3, Result4, Result5 As Short
Result1 = Pattern >> 0   ' Result is 2560 (0000 1010 0000 0000).
Result2 = Pattern >> 4    ' Result is 160 (0000 0000 1010 0000).
Result3 = Pattern >> 10     ' Result is 2 (0000 0000 0000 0010).
Result4 = Pattern >> 18   ' Result is 640 (0000 0010 1000 0000).
Result5 = Pattern >> -1   ' Result is 0 (shifted 15 places to right).
```

The shift amount for `Result4` is calculated as 18 AND 15, which equals 2.

```
Dim NegPattern As Short = -8192   ' Bit pattern is 1110 0000 0000 0000.
Dim NegResult1, NegResult2 As Short
NegResult1 = NegPattern >> 4   ' Result is -512 (1111 1110 0000 0000).
NegResult2 = NegPattern >> 13   ' Result is -1 (sign bit propagated).
```

See Also

Bit Shift Operators | Assignment Operators | >>= Operator | Operator Precedence in Visual Basic | Operators Listed by Functionality | Arithmetic Operators

>>= Operator

Performs an arithmetic right shift on the value of a variable and assigns the result back to the variable.

```
variable >>= amount
```

Parts

variable
Required. Variable of data type **Byte**, **Short**, **Integer**, or **Long**.
amount
Required. Numeric expression of a data type that widens to **Integer**.

Remarks

Arithmetic shifts are not circular, which means the bits shifted off one end of the result are not reintroduced at the other end. In an arithmetic right shift, the bits shifted beyond the rightmost bit position are discarded, and the leftmost bit is propagated into the bit positions vacated at the left. This means that if *variable* has a negative value, the vacated positions are set to one. If *variable* is positive, or if its data type is **Byte**, the vacated positions are set to zero.

Example

The following example uses the **>>=** operator to shift the bit pattern of an **Integer** variable right by the specified amount and assign the result to the variable.

```
Dim Var As Integer = 10   ' Variable containing bit pattern to shift.
Dim Shift As Integer = 2   ' Amount to shift bit pattern.
Var >>= Shift   ' The value of Var is now 2 (bit lost off right end).
```

See Also

>> Operator | Assignment Operators | Bit Shift Operators | Operator Precedence in Visual Basic | Operators Listed by Functionality | Assignment Statements

/ Operator

Divides two numbers and returns a floating-point result.

```
number1 / number2
```

Parts

result
Required. Any numeric variable.
number1
Required. Any numeric expression.
number2
Required. Any numeric expression.

Result

The result is the quotient of *number1* divided by *number2*.

Supported Types

Byte, **Short**, **Integer**, **Long**, **Single**, **Double**, **Decimal**

Remarks

The data type of *result* is usually **Double**. The following are exceptions to this rule:

If	**Then *result* is**
One expression is **Single** and the other is any data type except **Double**	**Single**.
Both expressions are **Decimal** data types	**Decimal**. If a **Decimal** expression is divided by 0, a **DividebyZero** exception is raised. This exception only occurs with a **Decimal** expression.

If an expression is stated as Nothing, or is Empty, it is treated as zero.

Example

This example uses the **/** operator to perform floating-point division. The result is the quotient of the two operands.

```
Dim MyValue As Double
MyValue = 10 / 4   ' Returns 2.5.
MyValue = 10 / 3   ' Returns 3.333333.
```

See Also

\= Operator | \ Operator | Arithmetic Operators | Operator Precedence in Visual Basic | Operators Listed by Functionality | Arithmetic Operators

/= Operator

Divides the value of a variable by the value of an expression and assigns the result to the variable.

```
variable /= expression
```

Parts

variable
Required. Any numeric variable.
expression
Required. Any numeric expression.

Remarks

This statement assigns a **Double** to the variable on the left hand side. If **Option Strict** is **On**, *variable* must be a **Double**. If **Option Strict** is **Off**, an implicit conversion will be made and the resultant value assigned to *variable*, with a possible error at run time. For more information, see Widening and Narrowing Conversions, and Option Strict Statement.

Example

The following example uses the **/=** operator to divide one **Integer** variable by a second and assign the quotient to the first variable.

```
Dim var1 As Integer = 12
Dim var2 As Integer = 3
var1 /= var2   ' The value of var1 is now 4.
```

See Also

&= Operator | *= Operator | += Operator | -= Operator | / Operator | = Operator | \= Operator | ^= Operator | Assignment Operators | Arithmetic Operators | Operator Precedence in Visual Basic | Operators Listed by Functionality | Assignment Statements

\ Operator

Divides two numbers and returns an integer result.

```
number1 \ number2
```

Parts

number1
Required. Any numeric expression of an integral type.
number2
Required. Any numeric expression of an integral type.

Supported Types

Byte, **Short**, **Integer**, or **Long**.

Result

The result is the integer quotient of *number1* and *number2*, dropping the remainder.

Remarks

Before division is performed, any floating-point numeric expressions are coerced to **Long** if **Option Strict** is **Off**. If **Option Strict** is **On**, a compiler error results.

The data type of the result is **Byte**, **Short**, **Integer**, or **Long**. Any fractional portion is truncated.

If any expression is stated as Nothing, it is treated as zero. Attempting to perform integer division by zero causes a **DivideByZeroExeception** to be thrown.

Example

This example uses the **** operator to perform integer division. The result is an integer representing the integer quotient of the two operands.

```
Dim myValue As Integer
myValue = 11 \ 4   ' Returns 2.
myValue = 9 \ 3   ' Returns 3.
myValue = 100 \ 3   ' Returns 33.
MyValue = 67 \ -3   ' Returns -22.
```

See Also

\= Operator | / Operator | Arithmetic Operators | Operator Precedence in Visual Basic | Operators Listed by Functionality | Arithmetic Operators

\= Operator

Divides the value of a variable by the value of an expression and assigns the integer result to the variable.

```
variable \= expression
```

Parts

variable
Required. Any numeric variable.
expression
Required. Any numeric expression.

Remarks

For further information on integer division, see \ Operator.

Example

The following example uses the **\=** operator to divide one **Integer** variable by a second and assign the integer result to the first variable.

```
Dim var1 As Integer = 10
Dim var2 As Integer = 3
var1 \= var2   ' The value of var1 is now 3.
```

See Also

&= Operator | *= Operator | += Operator | -= Operator | /= Operator | = Operator | \ Operator | ^= Operator | Assignment Operators | Arithmetic Operators | Operator Precedence in Visual Basic | Operators Listed by Functionality | Assignment Statements

^ Operator

Raises a number to the power of another number.

```
Number ^ exponent
```

Parts

number
Required. Any numeric expression.
exponent
Required. Any numeric expression.

Result

The result value will be *number* raised to the *exponent* power.

Supported Types

Double. All operands of a different type will be converted to **Double**.

Remarks

Number can be negative only if *exponent* is an integer value. When more than one exponentiation is performed in a single expression, the **^** operator is evaluated as it is encountered from left to right.

Usually, the data type of *result* is **Double**.

Example

This example uses the **^** operator to raise a number to the power of an exponent. The result is the first operand raised to the power of the second.

```
Dim myValue As Double
myValue = 2 ^ 2   ' Returns 4.
myValue = 3 ^ 3 ^ 3   ' Returns 19683.
myValue = (-5) ^ 3   ' Returns -125.
myValue = (-5) ^ 4   ' Returns 625.
```

See Also

^= Operator | Arithmetic Operators | Operator Precedence in Visual Basic | Operators Listed by Functionality | Arithmetic Operators

^= Operator

Raises the value of a variable to the power of an expression and assigns the result back to the variable.

```
variable ^= expression
```

Parts

variable
Required. Any numeric variable.
expression
Required. Any numeric expression.

Example

The following example uses the **^=** operator to raise the value of one **Integer** variable to the power of a second variable and assign the result to the first variable.

```
Dim var1 As Integer = 10
Dim var2 As Integer = 3
var1 ^= var2   ' The value of var1 is now 1000.
```

See Also

&= Operator | *= Operator | += Operator | -= Operator | /= Operator | = Operator | \= Operator | ^ Operator | Assignment Operators | Arithmetic Operators | Operator Precedence in Visual Basic | Operators Listed by Functionality | Assignment Statements

AddressOf Operator

Creates a procedure delegate instance that references the specific procedure.

```
AddressOf procedurename
```

The required *procedurename* specifies the procedure that will be referenced by the newly created procedure delegate.

Remarks

The **AddressOf** operator creates a function delegate that points to the function specified by *procedurename*. When the specified procedure is an instance method then the function delegate refers to both the instance and the method so that when the function delegate is invoked the specified method of the specified instance is called.

The AddressOf operator can be used as the operand of a delegate constructor or it can be used in a context in which the type of the delegate can be determined by the compiler.

Example

This example uses the **AddressOf** operator to designate a delegate to handle the **Click** event of a button.

```
Public Sub ButtonClickHandler(ByVal sender As Object, e As _
                              System.EventArgs)
' Implementation code omitted.
End Sub

' Add the following line to Sub Form1_Load().
   AddHandler Button1.Click, AddressOf ButtonClickHandler
```

'The following example uses the **AddressOf** operator to designate the startup function for a thread.

```
Imports System.Threading   ' Access threading namespace.

Public Sub CountSheep()
   Dim i As Integer = 1 ' Sheep don't count from 0.
   Do While (True) ' Endless loop.
      Console.WriteLine("Sheep " & i & " Baah")
      i = i + 1
      Thread.Sleep(1000) 'Wait 1 second.
   Loop
End Sub

Sub UseThread()
   Dim t As New System.Threading.Thread(AddressOf CountSheep)
   t.Start()
End Sub
```

See Also

Declare Statement | Function Statement | Sub Statement | Delegates and the AddressOf operator

And Operator

Performs a logical conjunction on two **Boolean** expressions, or bitwise conjunction on two numeric expressions.

```
result = expression1 And expression2
```

Parts

result
Required. Any **Boolean** or numeric expression. The result for **Boolean** comparison is the **Boolean** result of comparison of the two expressions. The result for numeric comparison is a numeric value resulting from the bitwise conjunction of two numeric expressions

expression1
Required. Any **Boolean** or numeric expression.

expression2
Required. Any **Boolean** or numeric expression.

Remarks

For **Boolean** comparison, if both *expression1* and *expression2* evaluate to **True**, *result* is **True**. If *expression1* evaluates to **True** and *expression2* evaluates to **False**, *result* is **False**. If *expression1* evaluates to **False**, and *expression2* evaluates to True, the *result* is **False.** The following table illustrates how *result* is determined:

If *expression1* is	**And *expression2* is**	**Value of *result* is**
True	**True**	**True**
True	**False**	**False**
False	**True**	**False**
False	**False**	**False**

When applied to numeric values, the **And** operator performs a bitwise comparison of identically positioned bits in two numeric expressions and sets the corresponding bit in *result* according to the following table:

If bit in expression1 is	**And bit in expression2 is**	**The result is**
0	0	0
0	1	0
1	0	0
1	1	1

> **Note** Since the logical/bitwise operators have a lower precedence than other arithmetic and relational operators, any bitwise operations should be enclosed in parentheses to ensure accurate execution.

If the operands consist of one **Boolean** expression and one numeric expression, the result **Boolean** expression will be converted to a numeric value (-1 for True, and 0 for False) and the bitwise operation will result.

Example

This example uses the **And** operator to perform a logical conjunction on two expressions. The result is a **Boolean** value that represents whether the entire conjoined expression is true.

```
Dim A As Integer = 10
Dim B As Integer = 8
Dim C As Integer = 6
Dim myCheck As Boolean
myCheck = A > B And B > C   ' Returns True.
myCheck = B > A And B > C   ' Returns False.
```

This example uses the **And** operator to perform logical conjunction of the individual bits of two numeric expressions. The bit in the result pattern is set if the corresponding bits in the operands are both set.

```
Dim A As Integer = 10
```

```
Dim B As Integer = 8
Dim C As Integer = 6
Dim myCheck As Integer
myCheck = (A And B)   ' Returns 8.
myCheck = (A And C)   ' Returns 2.
myCheck = (B And C)   ' Returns 0.
```

See Also

Logical/Bitwise Operators | Operator Precedence in Visual Basic | Operators Listed by Functionality | AndAlso Operator | Logical Operators

AndAlso Operator

Performs short-circuiting logical conjunction on two expressions.

```
result = expression1 AndAlso expression2
```

Parts

result
Required. Any **Boolean** expression. The result is the **Boolean** result of comparison of the two expressions.
expression1
Required. Any **Boolean** expression.
expression2
Required. Any **Boolean** expression.

Remarks

A logical operation is said to be *short-circuiting* if the compiled code can bypass the evaluation of one expression depending on the result of another expression. If the result of the first expression evaluated determines the final result of the operation, there is no need to evaluate the other expression, because it cannot change the final result. Short-circuiting can improve performance if the bypassed expression is complex, or if it involves procedure calls.

If both *expression1* and *expression2* evaluate to **True**, *result* is **True**. If *expression1* evaluates to True and *expression2* evaluates to **False**, *result* is **False**. If *expression1* evaluates to **False**, *expression2* is not evaluated, and *result* is **False** (the operator is said to have short-circuited the expression). The following table illustrates how *result* is determined:

If *expression1* is	**And *expression2* is**	**Value of *result* is**
True	**True**	**True**
True	**False**	**False**
False	(not evaluated)	**False**

Example

This example uses the **AndAlso** operator to perform a logical conjunction on two expressions. The result is a **Boolean** value that represents whether the entire conjoined expression is true. If the first expression is **False**, the second is not evaluated.

```
Dim A As Integer = 10
Dim B As Integer = 8
Dim C As Integer = 6
Dim myCheck As Boolean
myCheck = A > B AndAlso B > C   ' True.
myCheck = B > A AndAlso B > C   ' False. Second expression not evaluated.
myCheck = A > B AndAlso C > B   ' False. Second expression evaluated.

Public Function FindValue(ByVal Arr() As Double, _
                          ByVal SearchValue As Double) As Double
   Dim I As Integer = 0
   While I <= UBound(Arr) AndAlso Arr(I) <> SearchValue
   ' If I is greater than UBound(Arr), SearchValue is not checked.
      I += 1
   End While
   If I >= UBound(Arr) Then I = -1
   Return I
End Function
```

See Also

Logical/Bitwise Operators | Operator Precedence in Visual Basic | Operators Listed by Functionality | And Operator | Logical Operators

GetType Operator

Returns a **Type** object for the specified type. The **Type** object can be used to retrieve information about the type such as its properties, methods, and events.

```
GetType(typename)
```

Parameters

typename
The name of the type for which information is desired.

Remarks

The **GetType** operator returns the type object for the specified *typename*. If you want to get the type object of an object variable, use Type.GetType Method.

Example

The following examples show the **GetType** operator in use.

```
GetType(Integer)   ' Returns the integer Type object.
GetType(String())   ' Returns the Type object for 1D string arrays.
```

See Also

Operator Precedence in Visual Basic | Operators Listed by Functionality | Operators

Is Operator

Compares two object reference variables.

```
result = object1 Is object2
```

Parts

result
Required. Any **Boolean** value.
object1
Required. Any **Object** name.
object2
Required. Any **Object** name.

Remarks

The **Is** operator determines if two object references refer to the same object. However, it does not perform value comparisons. If *object1* and *object2* both refer to the same object, *result* is **True**; if they do not, *result* is **False**.

Example

This example uses the **Is** operator to compare two object references. The object variable names are generic and used for illustration purposes only. The result is a **Boolean** value representing whether the two objects are identical.

```
Dim myObject As New Object
Dim otherObject As New Object
Dim yourObject, thisObject, thatObject As Object
Dim myCheck As Boolean
yourObject = myObject
thisObject = myObject
thatObject = otherObject
myCheck = yourObject Is thisObject   ' Returns True.
myCheck = thatObject Is thisObject   ' Returns False.
myCheck = myObject Is thatObject ' Returns False, if myObject is not thatObject.
```

See Also

Comparison Operators | Operator Precedence in Visual Basic | Operators Listed by Functionality | Operators

Like Operator

Compares two strings.

```
result = string Like pattern
```

Parts

result
Required. Any **Boolean** variable. The result is a **Boolean** value indicating whether or not the *string* matches the *pattern*.
string
Required. Any **String** expression.
pattern
Required. Any **String** expression conforming to the pattern-matching conventions described in Remarks.

Remarks

If *string* matches *pattern*, *result* is **True**; if there is no match, *result* is **False**. If both *string* and *pattern* are an empty string. the result is **True**. Otherwise, if either *string* or *pattern* is an empty string, the result is **False**.

The behavior of the **Like** operator depends on the **Option Compare** statement. The default string-comparison method for each module is **Option Compare Binary**.

Built-in pattern matching provides a versatile tool for string comparisons. The pattern-matching features allow you to match strings using wildcard characters, character lists, or character ranges, in any combination. The following table shows the characters allowed in *pattern* and what they match:

Characters in *pattern*	**Matches in *string***
?	Any single character
*****	Zero or more characters
#	Any single digit (0–9)
[*charlist*]	Any single character in *charlist*
[**!***charlist*]	Any single character not in *charlist*

A group of one or more characters (*charlist*) enclosed in brackets (**[ ]**) can be used to match any single character in *string* and can include almost any character code, including digits.

> **Note** To match the special characters left bracket (**[**), question mark (**?**), number sign (**#**), and asterisk (*****), enclose them in brackets. The right bracket (**]**) can't be used within a group to match itself, but it can be used outside a group as an individual character.

By using a hyphen (**–**) to separate the upper and lower bounds of the range, *charlist* can specify a range of characters. For example, `[A-Z]` results in a match if the corresponding character position in *string* contains any uppercase letters in the range *A*–*Z*. Multiple ranges are included within the brackets without delimiters.

The meaning of a specified range depends on the character ordering valid at run time (as determined by **Option Compare** and the locale setting of the system the code is running on). Using **Option Compare Binary**, the range `[A-E]` matches *A*, *B*, *C*, *D*, and *E*. With **Option Compare Text**, `[A-E]` matches *A*, *a*, À, à, *B*, *b*, *C*, *c*, *D*, *d*, *E*, and *e*. The range does not match Ê or ê because accented characters fall after unaccented characters in the sort order.

Other important rules for pattern matching include the following:

- An exclamation point (**!**) at the beginning of *charlist* means that a match is made if any character except the characters in *charlist* is found in *string*. When used outside brackets, the exclamation point matches itself.
- A hyphen (**–**) can appear either at the beginning (after an exclamation point if one is used) or at the end of *charlist* to match itself. In any other location, the hyphen identifies a range of characters delimited by the characters on either side of the hyphen.
- When a range of characters is specified, they must appear in ascending sort order (that is, from lowest to highest). Thus, `[A-Z]` is a valid pattern, but `[Z-A]` is not.
- The character sequence `[]` is considered a zero-length string ("").

In some languages, there are special characters in the alphabet that represent two separate characters. For example, several languages use the character æ to represent the characters *a* and *e* when they appear together. The **Like** operator recognizes that the single special character and the two individual characters are equivalent.

When a language that uses a special character is specified in the system locale settings, an occurrence of the single special character in either *pattern* or *string* matches the equivalent two-character sequence in the other string. Similarly, a single special character in *pattern* enclosed in brackets (by itself, in a list, or in a range) matches the equivalent two-character sequence in *string*.

Example

This example uses the **Like** operator to compare a string to a pattern. The result is a **Boolean** value representing whether the string fits the pattern.

```
Dim myCheck As Boolean
myCheck = "F" Like "F"   ' Does "F" match "F"? Returns True.
myCheck = "F" Like "f"   ' Does "F" match "f"? Returns False.
myCheck = "F" Like "FFF"   ' Does "F" match "FFF"? Returns False.
myCheck = "aBBBa" Like "a*a"   ' Does "aBBBa" have a "a" at the
   ' beginning, an "a" at the end, and any number of characters in
   ' between? Returns True.
myCheck = "F" Like "[A-Z]"   ' Does "F" occur in the set of
   ' characters from A to Z? Returns True.
myCheck = "F" Like "[!A-Z]"     ' Does "F" NOT occur in the set of
   ' characters from A to Z? Returns False.
myCheck = "a2a" Like "a#a"     ' Does "a2a" begin and end with an
   ' "a" and have any single-digit number inbetween? Returns True.
myCheck = "aM5b" Like "a[L-P]#[!c-e]" ' Does "aM5b" fit the following
   ' pattern: Begins with "a", has and character from the set L through
   ' P, followed byb any single-digit number, and finally contains any
   ' character excluded from the character set c through e. Returns True.
myCheck = "BAT123khg" Like "B?T*"  ' Does "BAT123khg" fit the
   ' following pattern: Begins with "B", followed by any single
   ' character, followed by a "T" and finally zero or more characters
   ' of any type. Returns True.
myCheck = "CAT123khg" Like "B?T*"  ' Does "CAT123khg" fit the
   ' following pattern: Begins with "B", followed by any single
   ' character, followed by a "T" and finally zero or more characters
   ' of any type. Returns False.
```

See Also

Comparison Operators | InStr Function | Operator Precedence in Visual Basic | Operators Listed by Functionality | Option Compare Statement | StrComp Function | Operators

Mod Operator

Divides two numbers and returns only the remainder.

```
number1 Mod number2
```

Parts

number1
Required. Any numeric expression.
number2
Required. Any numeric expression.

Result

The result is the remainder left after division is performed on *number1* and *number2*.

Supported Types

Byte, **Short**, **Integer**, **Long**, **Single**, **Double**, **Decimal**

Remarks

The **Mod** operator divides *number1* by *number2* and returns only the remainder as *result*. For example, in the following expression, `A` (*result*) equals 2.

```
A = 8 Mod 3
```

If *number1* or *number2* are floating-point values, then division is carried out and the floating-point remainder is returned. The data type of the result is the same as that of the data type with the greatest range. The order of range, from least to greatest range, is Byte, Short, Integer, Long, Single, Double, and Decimal.

If any expression is stated as Nothing, or Empty, it is treated as zero. If division by zero occurs, the **Mod** operation returns **NaN (Not a Number)**.

`A Mod B` is the equivalent of `A - Int(A / B) * B + CLng(Math.Sign(A) <> Math.Sign(B)) * B`

Example

This example uses the **Mod** operator to divide two numbers and return only the remainder. If either number is a floating-point number, the result is a floating-point number representing the remainder.

```
Dim myResult As Double
myResult = 10 Mod 5   ' Returns 0.
myResult = 10 Mod 3   ' Returns 1.
myResult = 12 Mod 4.3   ' Returns 3.4.
myResult = 12.6 Mod 5   ' Returns 2.6.
myResult = 47.9 Mod 9.35   ' Returns 1.15.
```

See Also

Arithmetic Operators | Operator Precedence in Visual Basic | Operators Listed by Functionality | Arithmetic Operators

Not Operator

Performs logical negation on a **Boolean** expression, or bitwise negation on a numeric expression.

```
result = Not expression
```

Parts

result
Required. Any **Boolean** or numeric expression.
expression
Required. Any **Boolean** or numeric expression.

Remarks

For **Boolean** expressions, the following table illustrates how *result* is determined:

If *expression* is	Then *result* is
True	**False**
False	**True**

For numeric expressions, the **Not** operator inverts the bit values of any numeric expression and sets the corresponding bit in *result* according to the following table:

If bit in *expression* is	Then bit in *result* is
0	1
1	0

> **Note** Since the logical/bitwise operators have a lower precedence than other arithmetic and relational operators, any bitwise operations should be enclosed in parentheses to ensure accurate execution.

Example

This example uses the **Not** operator to perform logical negation on a **Boolean** expression. The result is a **Boolean** value representing whether the expression is false. That is, if the expression is false, the result of the **Not** operator is true.

```
Dim A As Integer = 10
Dim B As Integer = 8
Dim C As Integer = 6
Dim myCheck As Boolean
myCheck = Not(A > B)   ' Returns False.
myCheck = Not(B > A)   ' Returns True.
```

This example uses the **Not** operator to perform logical negation of the individual bits of two numeric expressions. The bit in the result pattern is set to the reverse of the corresponding bit in the operand pattern, including the sign bit.

```
Dim A As Integer = 10
Dim B As Integer = 8
Dim C As Integer = 6
Dim myCheck As Integer
myCheck = (Not A)   ' Returns -11.
myCheck = (Not B)   ' Returns -9.
myCheck = (Not C)   ' Returns -7.
```

See Also

Logical/Bitwise Operators | Operator Precedence in Visual Basic | Operators Listed by Functionality | Logical Operators

Or Operator

Used to perform a logical disjunction on two **Boolean** expressions, or bitwise disjunction on two numeric values..

```
result = expression1 Or expression2
```

Parts

result
Required. Any **Boolean** or numeric expression. For **Boolean** comparison the result is the logical disjunction of two expressions. For bitwise operations the result is a numeric value resulting from the bitwise disjunction of two numeric expressions.
expression1
Required. Any **Boolean** or numeric expression.
expression2
Required. Any **Boolean** or numeric expression of the same type as *expression1*.

Remarks

For **Boolean** comparison, if either *expression1* or *expression2* evaluates to **True**, *result* is **True.** If *expression1* evaluates to **True**, and *expression2* evaluates to **False** the *result* is **True**. The following table illustrates how *result* is determined:

If *expression1* is	**And *expression2* is**	**Then *result* is**
True	**True**	**True**
True	**False**	**True**
False	**True**	**True**
False	**False**	**False**

For bitwise operations, the **Or** operator performs a bitwise comparison of identically positioned bits in two numeric expressions and sets the corresponding bit in *result* according to the following table:

If bit in *expression1* is	**And bit in *expression2* is**	**Then *result* is**
0	0	0
0	1	1
1	0	1
1	1	1

> **Note** Since the logical/bitwise operators have a lower precedence than other arithmetic and relational operators, any bitwise operations should be enclosed in parentheses to ensure accurate execution.

If the operands consist of one **Boolean** expression and one numeric expression, the result **Boolean** expression will be converted to a numeric value (-1 for True, and 0 for False) and the bitwise operation will result.

Example

This example uses the **Or** operator to perform logical disjunction on two expressions. The result is a **Boolean** value that represents whether either of the two expressions is true.

```
Dim A As Integer = 10
Dim B As Integer = 8
Dim C As Integer = 6
Dim myCheck As Boolean
myCheck = A > B Or B > C   ' Returns True.
myCheck = B > A Or B > C   ' Returns True.
myCheck = B > A Or C > B   ' Returns False.
```

This example uses the **Or** operator to perform logical disjunction of the individual bits of two numeric expressions. The bit in the result pattern is set if either of the corresponding bits in the operands are set.

```
Dim A As Integer = 10
Dim B As Integer = 8
```

```
Dim C As Integer = 6
Dim myCheck As Integer
myCheck = (A Or B)   ' Returns 10.
myCheck = (A Or C)   ' Returns 14.
myCheck = (B Or C)   ' Returns 14.
```

See Also

Logical/Bitwise Operators | Operator Precedence in Visual Basic | Operators Listed by Functionality | OrElse Operator | Logical Operators

OrElse Operator

Used to perform short-circuiting logical disjunction on two expressions.

```
result = expression1 OrElse expression2
```

Parts

result
Required. Any **Boolean** expression.
expression1
Required. Any **Boolean** expression.
expression2
Required. Any **Boolean** expression.

Remarks

A logical operation is said to be *short-circuiting* if the compiled code can bypass the evaluation of one expression depending on the result of another expression. If the result of the first expression evaluated determines the final result of the operation, there is no need to evaluate the other expression, because it cannot change the final result. Short-circuiting can improve performance if the bypassed expression is complex, or if it involves procedure calls.

If either *expression1* or *expression2* evaluates to **True**, *result* is **True.** If *expression1* evaluates to **True**, *expression2* is not evaluated, and *result* is **True** (the operator is said to have short-circuited the expression). The following table illustrates how *result* is determined:

If *expression1* is	**And *expression2* is**	**Then *result* is**
True	(not evaluated)	**True**
False	**True**	**True**
False	**False**	**False**

Example

This example uses the **OrElse** operator to perform logical disjunction on two expressions. The result is a **Boolean** value that represents whether either of the two expressions is true. If the first expression is **True**, the second is not evaluated.

```
Dim A As Integer = 10
Dim B As Integer = 8
Dim C As Integer = 6
Dim myCheck As Boolean
myCheck = A > B OrElse B > C   ' True. Second expression is not evaluated.
myCheck = B > A OrElse B > C   ' True. Second expression is evaluated.
myCheck = B > A OrElse C > B   ' False.

If MyFunction(5) = True OrElse MyOtherFunction(4) = True Then
' If MyFunction(5) is True, MyOtherFunction(4) is not called.
   ' Insert code to be executed.
End If
```

See Also

Logical/Bitwise Operators | Operator Precedence in Visual Basic | Operators Listed by Functionality | Or Operator | Logical Operators

Xor Operator

Performs a logical exclusion operation on two **Boolean** expressions, or a bitwise exclusion on two numeric expressions..

```
result = expression1 Xor expression2
```

Parts

result
Required. Any **Boolean** or numeric variable. The result for **Boolean** comparison is the logical exclusion of two expressions. For bitwise operations the result is a numeric value resulting from the bitwise exclusion of two numeric expressions

expression1
Required. Any **Boolean** or numeric expression.

expression2
Required. Any **Boolean** or numeric expression of the same type as *expression1*.

Remarks

For **Boolean** comparisons, if one and only one of the expressions evaluates to **True**, *result* is **True**. Otherwise, *result* is **False**. If either expression is stated as Nothing, that expression is evaluated as **False**.

If *expression1* is	**And *expression2* is**	**Then *result* is**
True	**True**	**False**
True	**False**	**True**
False	**True**	**True**
False	**False**	**False**

For numeric expressions, the **Xor** operator performs as a bitwise operator. A bitwise comparison of two expressions using exclusive-or logic to form the result, as shown in the following table:

If bit in *expression1* is	**And bit in *expression2* is**	**Then *result* is**
0	0	0
0	1	1
1	0	1
1	1	0

> **Note** Since the logical/bitwise operators have a lower precedence than other arithmetic and relational operators, any bitwise operations should be enclosed in parentheses to ensure accurate execution.

If the operands consist of one **Boolean** expression and one numeric expression, the result **Boolean** expression will be converted to a numeric value (-1 for True, and 0 for False) and the bitwise operation will result.

Example

This example uses the **Xor** operator to perform logical exclusion on two expressions. The result is a **Boolean** value representing whether *only* one of the expressions is true.

```
Dim A As Integer = 10
Dim B As Integer = 8
Dim C As Integer = 6
Dim myCheck As Boolean
myCheck = A > B Xor B > C   ' Returns False.
myCheck = B > A Xor B > C   ' Returns True.
myCheck = B > A Xor C > B   ' Returns False.
```

This example uses the **Xor** operator to perform logical exclusion of the individual bits of two numeric expressions. The bit in the result pattern is set if only one of the corresponding bits in the operands are set.

```
Dim A As Integer = 10
Dim B As Integer = 8
```

```
Dim C As Integer = 6
Dim myCheck As Integer
myCheck = (A Xor B)   ' Returns 2.
myCheck = (A Xor C)   ' Returns 12.
myCheck = (B Xor C)   ' Returns 14.
```

See Also

Logical/Bitwise Operators | Operator Precedence in Visual Basic | Operators Listed by Functionality | Logical Operators

Arithmetic Operators

The following are the arithmetic operators defined in Visual Basic .NET.

^ Operator

* Operator

/ Operator

\ Operator

Mod Operator

+ Operator

- Operator

See Also

Operator Precedence in Visual Basic | Arithmetic Operators

Assignment Operators

= Operator

^= Operator

*= Operator

/= Operator

\= Operator

+= Operator

-= Operator

<<= Operator

>>= Operator

&= Operator

See Also

Operator Precedence in Visual Basic | Operators Listed by Functionality | Assignment Statements

Bit Shift Operators

<< Operator

>> Operator

See Also

Operators Listed by Functionality

Comparison Operators

Compares expressions.

```
result = expression1 comparisonoperator expression2
result = object1 Is object2
result = string Like pattern
```

Parts

result
Required. Any numeric variable. The result is a **Boolean** value representing the result of the comparison.
expression
Required. Any expression.
comparisonoperator
Required. Any comparison operator.
object
Required. Any reference object name.
string
Required. Any **String** expression.
pattern
Required. Any **String** expression or range of characters.

Remarks

The following table contains a list of the comparison operators and the conditions that determine whether *result* is **True** or **False.**

Operator	True if	False if
< (Less than)	*expression1 < expression2*	*expression1 >= expression2*
<= (Less than or equal to)	*expression1 <= expression2*	*expression1 > expression2*
> (Greater than)	*expression1 > expression2*	*expression1 <= expression2*
>= (Greater than or equal to)	*expression1 >= expression2*	*expression1 < expression2*
= (Equal to)	*expression1 = expression2*	*expression1 <> expression2*
<> (Not equal to)	*expression1 <> expression2*	*expression1 = expression2*

Note The **Is** and **Like** operators have specific comparison functionality that differs from the operators in the table.

When an expression of type **Single** is compared to one of type **Double**, the **Single** expression is converted to **Double**. This behavior is opposite to the behavior found in Visual Basic 6.

Similarly, when an expression of type **Decimal** is compared with an expression of type **Single** or **Double**, the **Decimal** expression will be converted to **Single** or **Double**. For **Decimal** expressions, any fractional value less than 1E-28 may be lost. Such fractional value loss may cause two values to compare as equal when they are not. For this reason, care should be taken when using equality (=) to compare two floating-point variables. It is better to test that the absolute value of the difference between the two numbers is less than a small acceptable tolerance.

Comparing Strings

When Strings are compared, the string expressions are evaluated based on their alphabetical sort order. The sort order for strings is evaluated based upon the **Option Compare** setting.

Option Compare Binary results in string comparisons based on a sort order derived from the internal binary representations of the characters. Sort order is determined by the code page. In the following example, a typical binary sort order is shown:

```
A < B < E < Z < a < b < e < z < À < Ê < Ø < à < ê < ø
```

Option Compare Text results in string comparisons based on a case-insensitive, textual sort order determined by your system's locale. When you sort the same characters using **Option Compare Text**, the following text sort order is produced:

(A=a) < (À= à) < (B=b) < (E=e) < (Ê= ê) < (Ø = ø) < (Z=z)

Typeless programming with comparison operators

The use of comparison operators with **Object** expressions is not allowed under **Option Strict On**. When **Option Strict** is **Off**, and *expression1* or *expression2* are **Object** expressions, their runtime type determines how they are compared. The following table shows how the expressions are compared and the result from the comparison, depending on the runtime type of the operands:

If operands are:	**Comparison will be:**
Both strings	Sort comparison based on string sorting characteristics.
Both numeric values	Objects will be converted to **Double**, and numeric comparison performed.
One numeric value, and one **String**	The **String** is converted to a **Double** and numeric comparison i s performed. If the **String** cannot be converted to **Double**, an **In validCastException** will be thrown.
Either or both are reference types other than **String**	An **InvalidCastException** will be thrown.

For numeric comparisons, **Nothing** is treated as 0. For string comparisons, **Nothing** is treated as "".

Example

This example shows various uses of comparison operators, which you use to compare expressions. Comparison operators return a **Boolean** result that represents whether or not the stated expression is true. When the > and < operators are applied to strings, the comparison is made between the normal alphabetical sorting order of the strings. This may be dependent on locale. Whether the sort is case-sensitive or not depends on the Option Compare setting.

```
Dim myResult As Boolean
myResult = 45 < 35   ' Returns False.
myResult = 45 = 45   ' Returns True.
myResult = 4 <> 3   ' Returns True.
myResult = "5" > "4444"   ' Returns True.
```

See Also

= Operator | Is Operator | Like Operator | Option Compare Statement | Operator Precedence in Visual Basic | Operators Listed by Functionality | Option Compare Statement | Comparison Operators

Concatenation Operators

& Operator

+ Operator

See Also

Operator Precedence in Visual Basic | Concatenation Operators

Logical/Bitwise Operators

And Operator

Not Operator

Or Operator

Xor Operator

AndAlso Operator

OrElse Operator

See Also

Operator Precedence in Visual Basic | Logical Operators

Miscellaneous Operators

AddressOf Operator

GetType Operator

See Also

Operators Listed by Functionality

Properties

In This Section

Count Property

DateString Property

Description Property

Erl Property

HelpContext Property

HelpFile Property

Item Property

LastDLLError Property

Now Property

Number Property

ScriptEngine Property

ScriptEngineBuildVersion Property

ScriptEngineMajorVersion Property

ScriptEngineMinorVersion Property

Source Property

TimeOfDay Property

Timer Property

TimeString Property

Today Property

Related Sections

Visual Basic Language and Run-Time Reference

Visual Basic Language

Count Property

Returns an integer containing the number of objects in a collection. Read-only.

```
Overridable Public ReadOnly Property Count() As Integer
```

Example

This example uses the **Collection** object's **Count** property to specify how many iterations are required to remove all the elements of the collection called `Classes1`. When collections are numerically indexed, the base is 1 by default. Since collections are reindexed automatically when a removal is made, the following code removes the first member on each iteration.

```
Dim Num As Long
Dim Classes1 As New Collection()
For Num = 1 To Classes1.Count   ' Remove name from the collection.
   Classes1.Remove(1)   ' Default collection numeric indexes.
Next
```

Requirements

Namespace: Microsoft.VisualBasic

Class: Collection

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Add Method | Item Property | Remove Method

Applies To

Collection Object

DateString Property

Returns or sets a **String** value representing the current date according to your system.

```
Public Property DateString As String
```

Exceptions/Errors

Exception type	Error number	Condition
InvalidCastException	5	Invalid cast.

Remarks

DateString always returns the system date as "MM-dd-yyyy", which uses the abbreviated month name. The accepted formats for setting the date are "M-d-yyyy", "M-d-y", "M/d/yyyy", and "M/d/y".

If you attempt to set **DateString** with an invalid value, an **InvalidCastException** occurs.

To get or set the current system time as a **String**, use the **TimeString** property.

To access the current system date as a **Date**, use the **Today** property.

Example

This example uses the **DateString** property to display the current system date.

```
MsgBox("The current date is " & DateString)
```

Requirements

Namespace: Microsoft.VisualBasic

Module: DateAndTime

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

Because **DateString** is a member of a module, rather than of a class, you do not need to create an object on which to access **DateString**.

See Also

Now Property | TimeString Property | Today Property | System Namespace | DateTime Structure | ArgumentException Class

Description Property

Returns or sets a **String** expression containing a descriptive string associated with an error. Read/write.

For the **Err** object, returns or sets a descriptive string associated with an error.

```
Public Property Description() As String
```

Remarks

The **Description** property setting consists of a short description of the error. Use this property to alert the user to an error that you can't or don't want to handle. When generating a user-defined error, assign a short description of your error to the **Description** property. If the **Description** property isn't filled in, and the value of the Number property corresponds to a Visual Basic run-time error, then the string returned by the ErrorToString function is set in the **Description** property when the error is generated.

Example

This example assigns a user-defined message to the **Description** property of the **Err** object.

```
On Error Resume Next
Err.Raise(60000)
Err.Description = "Your Widget needs a new Frob!"
MsgBox(Err.Description)
```

Requirements

Namespace: Microsoft.VisualBasic

Class: ErrObject

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

ErrorToString Function | HelpContext Property | HelpFile Property | LastDLLError Property | Number Property | Source Property | Version Property

Applies To

Err Object

Erl Property

Returns an integer indicating the line number of the last executed statement. Read-only.

```
Public ReadOnly Property Erl() As Integer
```

Remarks

If Visual Basic encounters no line numbers, it returns 0.

Example

This example uses the **Erl** property to indicate the line number.

```
10:     On Error Resume Next
20:     Err.Raise(60000)
' Returns 20.
30:     MsgBox(Erl())
```

Smart Device Developer Notes

This property is not supported.

Requirements

Namespace: Microsoft.VisualBasic

Class: ErrObject

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

ErrorToString Function

Applies To

Err Object

HelpContext Property

Returns or sets an **Integer** containing the context ID for a topic in a Help file. Read/write.

```
Public Property HelpContext() As Integer
```

Remarks

The **HelpContext** property is used to display context-sensitive help for an application. If a help file is specified in **HelpFile**, the **HelpContext** property is used to automatically display the help file identified. If both the **HelpFile** and **HelpContext** properties are empty, the value of the Number property is checked. If the value of the **Number** property corresponds to a Visual Basic run-time error value, then the Visual Basic Help context ID for the error is used. If the value of the **Number** property doesn't correspond to a Visual Basic error, the contents screen for the Visual Basic Help file is displayed.

> **Note** You should write routines in your application to handle typical errors. When programming with an object, you can use the object's Help file to improve the quality of your error handling, or to display a meaningful message to your user if the error isn't recoverable.

Example

This example uses the **HelpContext** property of the **Err** object to show the Visual Basic Help topic for the `Overflow` error.

```
Dim Msg As String
Err.Clear
On Error Resume Next
Err.Raise(6)   ' Generate "Overflow" error.
If Err.Number <> 0 Then
   Msg = "Press F1 or HELP to see " & Err.HelpFile & " topic for" & _
   " the following HelpContext: " & Err.HelpContext
   MsgBox(Msg, , "Error:" )
End If
```

Requirements

Namespace: Microsoft.VisualBasic

Class: ErrObject

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Description Property | ErrorToString Function | HelpFile Property | LastDLLError Property | Number Property | Source Property

Applies To

Err Object

HelpFile Property

Returns or sets a **String** expression containing the fully qualified path to a Help file. Read/write.

```
Public Property HelpFile() As String
```

Remarks

If a Help file is specified by the **HelpFile** property, it is automatically called when the user presses the Help key (or the F1 key) in the error message dialog box. If the **HelpContext** property contains a valid context ID for the specified file, that topic is automatically displayed. If no **HelpFile** is specified, the Visual Basic Help file is displayed.

> **Note** You should write routines in your application to handle typical errors. When programming with an object, you can use the object's Help file to improve the quality of your error handling, or to display a meaningful message to your user if the error isn't recoverable.

Example

This example uses the **HelpFile** property of the **Err** object to start the Help system. By default, the **HelpFile** property contains the name of the Visual Basic Help file.

```
Dim Msg As String
Err.Clear
On Error Resume Next   ' Suppress errors for demonstration purposes.
Err.Raise(6)   ' Generate "Overflow" error.
Msg = "Press F1 or HELP to see " & Err.HelpFile & _
" topic for this error"
MsgBox (Msg, , "Error: ")
```

Requirements

Namespace: Microsoft.VisualBasic

Class: ErrObject

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Description Property | HelpContext Property | LastDLLError Property | Number Property | Source Property

Applies To

Err Object

Item Property

Returns a specific member of a **Collection** object either by position or by key.

```
ReadOnly Public Default Overloads Property Item( _
   ByVal Index As { Integer | Object } _
) As Object
```

Parameters

Object
An object expression that evaluates to an object in the **Collection** object.

Index
An expression that specifies the position of a member of the collection. If a numeric expression, *Index* must be a number from 1 to the value of the collection's **Count** property. If *Index* is a **String** expression, it must correspond to the key value specified when the member referred to was added to the collection.

Exceptions/Errors

Exception type	Error number	Condition
ArgumentException	5	Index doesn't match an existing member of the collection.
IndexOutOfRangeException	9	*Index* does not match an existing member of the collection.

Remarks

If *Index* doesn't match any existing member of the collection, an error occurs.

The **Item** property is the default property for a collection. Therefore, the following lines of code are equivalent:

```
Print MyCollection(1)
Print MyCollection.Item(1)
```

Example

This example uses the **Item** property to retrieve a reference to an object in a collection. Given that `birthdays` is a **Collection** object, the following code retrieves a reference to the objects representing Bill's birthday from the collection, using the key "Bill" as the *Index* arguments.

```
Dim birthdays As New Collection()
Dim aBirthday As DateTime
birthdays.Add(New DateTime(2001, 1, 12), "Bill")
aBirthday = birthdays("Bill")
MsgBox(aBirthday.ToString())
aBirthday = birthdays.Item("Bill")
MsgBox(aBirthday.ToString())
```

Note that the first call explicitly specifies the **Item** property, but the second does not. Both calls work because the **Item** property is the default for a **Collection** object. The reference, assigned to `aBirthday`, can be used to access the properties and methods of the specified object.

Requirements

Namespace: Microsoft.VisualBasic

Module: Collection

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Add Method | Count Property | Remove Method | IndexOutOfRangeException Class

Applies To

Collection Object

LastDLLError Property

Returns a system error code produced by a call to a dynamic-link library (DLL). Read-only.

```
ReadOnly Property LastDLLError() As Integer
```

Remarks

The **LastDLLError** property applies only to DLL calls made from Visual Basic code. When such a call is made, the called function usually returns a code indicating success or failure, and the **LastDLLError** property is filled. Check the documentation for the DLL's functions to determine the return values that indicate success or failure. Whenever the failure code is returned, the Visual Basic application should immediately check the **LastDLLError** property. No exception is raised when the **LastDLLError** property is set.

> **Note** The **LastDllError** property requires unmanaged code permission, which may affect its execution in partial trust situations. For more information, see SecurityPermission and Code Access Permissions.

Example

When pasted into a **UserForm** module, the following code causes an attempt to call a DLL function. The call fails because the argument that is passed in (a null pointer) generates an error, and in any event, SQL can't be cancelled if it isn't running. The code following the call checks the return of the call, and then prints at the **LastDLLError** property of the **Err** object to reveal the error code. On systems without DLLs, **LastDLLError** always returns zero.

```
Private Declare Function SQLCancel Lib "ODBC32.dll" _
(ByVal hstmt As Integer) As Long

Private Sub Update()
   Dim RetVal As Integer
   Dim myhandle As Integer
   ' Call with invalid argument.
   RetVal = SQLCancel(myhandle)
   ' Check for SQL error code.
   If RetVal = -2 Then
      ' Display the information code.
      MsgBox("Error code is :" & Err.LastDllError)
    End If
End Sub
```

Smart Device Developer Notes

This property always returns zero.

Requirements

Namespace: Microsoft.VisualBasic

Class: ErrObject

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Declare Statement | Description Property | Err Object | ErrorToString Function | HelpContext Property | HelpFile Property | Number Property | Source Property

Applies To

Err Object

Now Property

Returns a **Date** value containing the current date and time according to your system.

```
ReadOnly Public Property Now() As DateTime
```

Remarks

To set the system date, use the **Today** property. To set the system time, use the **TimeOfDay** property.

Example

This example uses the **Now** property to return the current system date and time.

```
Dim ThisMoment As Date
ThisMoment= Now   ' Assign current system date and time.
```

Requirements

Namespace: Microsoft.VisualBasic

Module: DateAndTime

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

Because **Now** is a member of a module, rather than of a class, you do not need to create an object on which to access **Now**.

See Also

Day Function | Hour Function | Minute Function | Month Function | Second Function | TimeOfDay Property | Weekday Function | Year Function | TimeOfDay Property | Today Property | Date Data Type | System Namespace | DateTime Structure | ArgumentOutOfRangeException Class

Number Property

Returns or sets a numeric value specifying an error. Read/write.

```
Public Property Number() As Integer
```

Remarks

When returning a user-defined error from an object, set **Err.Number** by adding the number you selected as an error code to the **VbObjectError** constant. For example, you use the following code to return the number 1051 as an error code:

```
Err.Raise(Number := VbObjectError + 1051, Source:= "SomeClass")
```

Example

This example illustrates a typical use of the **Number** property in an error-handling routine.

```
' Typical use of Number property.
Sub test()
   On Error GoTo out

   Dim x, y As Integer
   x = 1 / y   ' Create division by zero error.
   Exit Sub
   out:
   MsgBox(Err.Number)
   MsgBox(Err.Description)
   ' Check for division by zero error.
   If Err.Number = 11 Then
      y = y + 1
   End If
   Resume Next
End Sub
```

Requirements

Namespace: Microsoft.VisualBasic

Class: ErrObject

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Description Property | ErrorToString Function | HelpContext Property | HelpFile Property | LastDLLError Property | Source Property

Applies To

Err Object

ScriptEngine Property

Returns a string representing the runtime currently in use.

```
ReadOnly Public Property ScriptEngine As String
```

Remarks

The **ScriptEngine** property returns the string "VB".

You can use **ScriptEngine** when your application is running as a standalone program, in a script, or in a host application.

Example

The following example uses the **ScriptEngine** property and related properties to return a string describing the current run-time information:

```
Function GetRuntimeInfo() As String
   Dim Runtime As String = ""
   Runtime = ScriptEngine & " Version "
   Runtime = Runtime & CStr(ScriptEngineMajorVersion) & "."
   Runtime = Runtime & CStr(ScriptEngineMinorVersion) & "."
   Runtime = Runtime & CStr(ScriptEngineBuildVersion)
   Return Runtime   ' Return the current run-time information.
End Function
```

Smart Device Developer Notes

This property is not supported.

Requirements

Namespace: Microsoft.VisualBasic

Module: Globals

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

Because **ScriptEngine** is a member of a module, rather than of a class, you do not need to create an object on which to access **ScriptEngine**.

See Also

ScriptEngineBuildVersion Property | ScriptEngineMajorVersion Property | ScriptEngineMinorVersion Property

ScriptEngineBuildVersion Property

Returns an integer containing the build version number of the runtime currently in use.

```
ReadOnly Public Property ScriptEngineBuildVersion As Integer
```

Remarks

The returned value corresponds directly to the version information contained in the DLL for the current runtime.

You can use **ScriptEngineBuildVersion** when your application is running as a standalone program, in a script, or in a host application.

Example

The following example uses the **ScriptEngineBuildVersion** property and related properties to return a string describing the current run-time information:

```
Function GetRuntimeInfo() As String
   Dim Runtime As String = ""
   Runtime = ScriptEngine & " Version "
   Runtime = Runtime & CStr(ScriptEngineMajorVersion) & "."
   Runtime = Runtime & CStr(ScriptEngineMinorVersion) & "."
   Runtime = Runtime & CStr(ScriptEngineBuildVersion)
   Return Runtime   ' Return the current run-time information.
End Function
```

Smart Device Developer Notes

This property is not supported.

Requirements

Namespace: Microsoft.VisualBasic

Module: Globals

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

Because **ScriptEngineBuildVersion** is a member of a module, rather than of a class, you do not need to create an object on which to access **ScriptEngineBuildVersion**.

See Also

ScriptEngine Property | ScriptEngineMajorVersion Property | ScriptEngineMinorVersion Property

ScriptEngineMajorVersion Property

Returns an integer containing the major version number of the runtime currently in use.

```
ReadOnly Public Property ScriptEngineMajorVersion As Integer
```

Remarks

The returned value corresponds directly to the version information contained in the DLL for the current runtime.

You can use **ScriptEngineMajorVersion** when your application is running as a standalone program, in a script, or in a host application.

Example

The following example uses the **ScriptEngineMajorVersion** property and related properties to return a string describing the current run-time information:

```
Function GetRuntimeInfo() As String
   Dim Runtime As String = ""
   Runtime = ScriptEngine & " Version "
   Runtime = Runtime & CStr(ScriptEngineMajorVersion) & "."
   Runtime = Runtime & CStr(ScriptEngineMinorVersion) & "."
   Runtime = Runtime & CStr(ScriptEngineBuildVersion)
   Return Runtime   ' Return the current run-time information.
End Function
```

Smart Device Developer Notes

This property is not supported.

Requirements

Namespace: Microsoft.VisualBasic

Module: Globals

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

Because **ScriptEngineMajorVersion** is a member of a module, rather than of a class, you do not need to create an object on which to access **ScriptEngineMajorVersion**.

See Also

ScriptEngine Property | ScriptEngineBuildVersion Property | ScriptEngineMinorVersion Property

ScriptEngineMinorVersion Property

Returns an integer containing the minor version number of the runtime currently in use.

```
ReadOnly Public Property ScriptEngineMinorVersion As Integer
```

Remarks

The returned value corresponds directly to the version information contained in the DLL for the current runtime.

You can use **ScriptEngineMinorVersion** when your application is running as a standalone program, in a script, or in a host application.

Example

The following example uses the **ScriptEngineMinorVersion** property and related properties to return a string describing the current run-time information:

```
Function GetRuntimeInfo() As String
   Dim Runtime As String = ""
   Runtime = ScriptEngine & " Version "
   Runtime = Runtime & CStr(ScriptEngineMajorVersion) & "."
   Runtime = Runtime & CStr(ScriptEngineMinorVersion) & "."
   Runtime = Runtime & CStr(ScriptEngineBuildVersion)
   Return Runtime   ' Return the current run-time information.
End Function
```

Smart Device Developer Notes

This property is not supported.

Requirements

Namespace: Microsoft.VisualBasic

Module: Globals

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

Because **ScriptEngineMinorVersion** is a member of a module, rather than of a class, you do not need to create an object on which to access **ScriptEngineMinorVersion**.

See Also

ScriptEngine Property | ScriptEngineBuildVersion Property | ScriptEngineMajorVersion Property

Source Property

Returns or sets a **String** expression specifying the name of the object or application that originally generated the error. Read/write.

```
Public Property Source() As String
```

Remarks

The **Source** property specifies a **String** expression representing the object that generated the error; the expression is usually the object's class name or process ID. Use the **Source** property to provide information when your code is unable to handle an error generated in an accessed object. For example, if you access Microsoft Excel and it generates a `Division by zero` error, Microsoft Excel sets **Err.Number** to its error code for that error and sets **Source** to "Excel.Application".

When the application is generating an error from code, **Source** is your application's programmatic ID. Within a class, **Source** should contain a name having the form *project.class*. When an unexpected error occurs in your code, the **Source** property is automatically filled in. For errors in a module, **Source** contains the project name.

Example

This example demonstrates the use of the **Source** property in a typical error handling routine. When an error is raised from Class1, the string "Class1" is assigned to the **Source** property of the **Err** object. This string is then displayed in an informative message indicating the source and number of the error.

```
Public Class Class1
   Public Sub MySub()
      On Error Resume Next
      Err.Raise(60000, "Class1")
      MsgBox(Err.Source & " caused an error of type " & Err.Number)
   End Sub
End Class
```

Requirements

Namespace: Microsoft.VisualBasic

Class: ErrObject

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Description Property | ErrorToString Function | GetObject Function | HelpContext Property | HelpContext Property | HelpFile Property | LastDLLError Property | Number Property | On Error Statement

Applies To

Err Object

TimeOfDay Property

Returns or sets a **Date** value containing the current time of day according to your system.

```
Public Property TimeOfDay() As DateTime
```

Remarks

The **Date** data type includes date components. When returning the system time, **TimeOfDay** sets these all to 1, so the returned value represents the first day of the year 1. When setting the system time, **TimeOfDay** ignores the date components.

To access the current system time as a **String**, use the **TimeString** property.

To get or set the current system date, use the **Today** property.

Example

This example uses the **TimeOfDay** property to return the current system time.

```
Dim MyTime As Date
MyTime = TimeOfDay   ' Return current system time.
```

Requirements

Namespace: Microsoft.VisualBasic

Module: DateAndTime

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

Because **TimeOfDay** is a member of a module, rather than of a class, you do not need to create an object on which to access **TimeOfDay**.

See Also

Timer Property | TimeString Property | Today Property | System Namespace | DateTime Structure | ArgumentException Class | ArgumentOutOfRangeException Class

Timer Property

Returns a **Double** value representing the number of seconds elapsed since midnight.

```
ReadOnly Public Property Timer() As Double
```

Remarks

The **Timer** property returns both the seconds and the milliseconds since midnight. The seconds are in the integral part of the return value, and the milliseconds are in the fractional part.

Example

This example uses the **Timer** property to pause the application. It can perform other processing during the pause.

```
Public Sub WaitFiveSeconds()
   If TimeOfDay >= #11:59:55 PM# Then
      MsgBox("The current time is within 5 seconds of midnight" & _
         vbCrLf & "The timer returns to 0.0 at midnight")
      Return
   End If
   Dim Start, Finish, TotalTime As Double
   If (MsgBox("Press Yes to pause for 5 seconds", MsgBoxStyle.YesNo)) = _
      MsgBoxResult.Yes Then
      Start = Microsoft.VisualBasic.DateAndTime.Timer
      Finish = Start + 5.0   ' Set end time for 5-second duration.
      Do While Microsoft.VisualBasic.DateAndTime.Timer < Finish
         ' Do other processing while waiting for 5 seconds to elapse.
      Loop
      TotalTime = Microsoft.VisualBasic.DateAndTime.Timer - Start
      MsgBox "Paused for " & TotalTime & " seconds"
   End If
End Sub
```

Note that you must qualify the **Timer** property with the **Microsoft.VisualBasic** namespace, because **Timer** is also a defined class in the **System.Threading**, **System.Timers**, and **System.Windows.Forms** namespaces.

Requirements

Namespace: Microsoft.VisualBasic

Module: DateAndTime

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

Because **Timer** is a member of a module, rather than of a class, you do not need to create an object on which to access **Timer**.

See Also

Randomize Statement | TimeOfDay Property | Today Property | System Namespace | System.Windows.Forms Namespace | DateTime Structure | ArgumentException Class | ArgumentOutOfRangeException Class

TimeString Property

Returns or sets a **String** value representing the current time of day according to your system.

```
Public Property TimeString As String
```

Exceptions/Errors

Exception type	Error number	Condition
InvalidCastException	5	Invalid cast.

Remarks

TimeString always returns the system time as "HH:mm:ss", which is a 24-hour format.

If you attempt to set **TimeString** with an invalid value, an **InvalidCastException** occurs.

To get or set the current system date as a **String**, use the **DateString** property.

To access the current system time as a **Date**, use the **TimeOfDay** property.

Example

This example uses the **TimeString** property to display the current system time.

```
MsgBox("The current time is " & TimeString)
```

Requirements

Namespace: Microsoft.VisualBasic

Module: DateAndTime

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

Because **TimeString** is a member of a module, rather than of a class, you do not need to create an object on which to access **TimeString**.

See Also

Now Property | DateString Property | TimeOfDay Property | System Namespace | DateTime Structure | ArgumentException Class

Today Property

Returns or sets a **Date** value containing the current date according to your system.

```
Public Property Today() As DateTime
```

Remarks

The **Date** data type includes time components. When returning the system date, **Today** sets all of these to 0, so the returned value represents midnight (00:00:00). When setting the system date, **Today** ignores the time components.

To access the current system date as a **String**, use the **DateString** property.

To get or set the current system time, use the **TimeOfDay** property.

Example

This example uses the **Today** property to return the current system date.

```
Dim MyDate As Date
MyDate = Today   ' Return current system date.
```

Requirements

Namespace: Microsoft.VisualBasic

Module: DateAndTime

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

Because **Today** is a member of a module, rather than of a class, you do not need to create an object on which to access **Today**.

See Also

Format Function | Now Property | DateString Property | TimeOfDay Property | Date Data Type | System Namespace | DateTime Structure | ArgumentOutOfRangeException Class

Statements

In This Section

A-E Statements

F-P Statements

Q-Z Statements

Related Sections

Visual Basic Language and Run-Time Reference

Visual Basic Language

A-E Statements

The following table contains a listing of the Visual Basic .NET language statements.

AddHandler	Call	Class	Const
Declare	Delegate	Dim	Do...Loop
End	Enum	Erase	Error
Event	Exit		

See Also

F-P Statements | Q-Z Statements | Visual Basic Language and Run-Time Reference

AddHandler Statement

Associates an event with an event handler.

```
AddHandler event, AddressOf eventhandler
```

Parts

event
The name of the event to handle.
eventhandler
The name of a procedure that will handle the event.

Remarks

The **AddHandler** and **RemoveHandler** statements allow you to start and stop event handling at any time during program execution.

Example

```
Sub TestEvents()
   Dim Obj As New Class1()
   ' Associate an event handler with an event.
   AddHandler Obj.Ev_Event, AddressOf EventHandler
   Obj.CauseSomeEvent()   ' Ask the object to raise an event.
End Sub

Sub EventHandler()
   ' This procedure handles events raised by the object Obj.
   MsgBox("EventHandler caught event.")   ' Handle the event.
End Sub

Public Class Class1
   Public Event Ev_Event()   ' Declare an event.
   Sub CauseSomeEvent()
      RaiseEvent Ev_Event()   ' Raise an event.
   End Sub
End Class
```

See Also

RemoveHandler Statement | Handles | Events and Event Handlers | AddHandler and RemoveHandler

Call Statement

Transfers control to a **Sub** procedure, **Function** procedure, or dynamic-link library (DLL) procedure.

```
[ Call ] ProcedureName[(ArgumentList)]
```

Parts

ProcedureName
Required. Name of the procedure to call.

ArgumentList
Optional. List of variables or expressions representing arguments that are passed to the procedure when it is called. Multiple arguments are separated by commas. If you include *ArgumentList*, you must enclose it in parentheses.

Remarks

You are not required to use the **Call** keyword when calling a procedure; however, if you use the **Call** statement to call any intrinsic, DLL, or user-defined function, the function's return value is discarded.

Example

This example illustrates how the **Call** statement is used to transfer control to a **Sub** procedure, an intrinsic function, and a dynamic-link library (DLL) procedure.

```
' Call a Sub procedure.
Call PrintToDebugWindow("Hello World")
' The above statement passes control to the following Sub procedure.
Sub PrintToDebugWindow(ByVal AnyString As String)
   Debug.WriteLine(AnyString)   ' Print to the Output window.
End Sub

' Call a Visual Basic run-time function (Shell), discard the return value.
Call Shell("C:\WINNT\system32\calc.exe", AppWinStyle.NormalFocus)
' The preceding path is for Windows 2000;
' The Windows XP path C:\Windows\system32\calc.exe.

' Call a Microsoft Windows DLL procedure. The Declare statement must be
' Private in a class, not in a module.
Private Declare Sub MessageBeep Lib "User32" (ByVal N As Integer)
Sub CallMyDll()
   Call MessageBeep(-1)   ' Call Windows DLL procedure.
   MessageBeep(-1)   ' Call again without Call keyword.
End Sub
```

See Also

Declare Statement | Function Statement | Sub Statement

Class Statement

Declares the name of a class, as well as a definition of the variables, properties, events, and methods that comprise the class.

```
[ <attrlist> ] [ Public | Private | Protected | Friend |
Protected Friend ] [ Shadows ] [ MustInherit | NotInheritable ] _
Class name
   [ Inherits classname ]
   [ Implements interfacenames ]
   [ statements ]
End Class
```

Parts

attrlist
Optional. List of attributes that apply to this class. Multiple attributes are separated by commas.

Public
Optional. Entities declared with the **Public** modifier have public access. There are no restrictions on the use of public entities.

Private
Optional. Entities declared with the **Private** modifier have private access. A private entity is accessible only within its declaration context, including any nested entities.

Protected
Optional. Entities declared with the **Protected** keyword have protected access. They are accessible only from within their own class or from a derived class. Protected access can be specified only on members of classes. It is not a superset of friend access.

Friend
Optional. Entities declared with the **Friend** modifier have friend access. An entity with friend access is accessible only within the program that contains the entity declaration. Classes that do not specify an access modifier are declared as **Friend** by default.

Protected Friend
Optional. Entities declared with the **Protected Friend** modifiers have the union of protected and friend accessibility.

Shadows
Optional. Indicates that this class shadows an identically named programming element in a base class. You can shadow any kind of declared element with any other kind. A shadowed element is unavailable from within the derived class that shadows it, unless the shadowing element is inaccessible, for example if it is **Private**.

MustInherit
Optional. Indicates that non-shared members of the **Class** can only be accessed via derived classes. Instances of must-inherit classes cannot be created.

NotInheritable
Optional. Indicates that the **Class** is a class from which no further inheritance is allowed.

name
Required. Name of the **Class**; follows standard variable naming conventions.

Inherits
Optional. Indicates that this class inherits the members of another class. A class can inherit from only one other class.

classname
The name of base class that this class inherits.

Implements
Optional. Indicates that this class implements the members of an interface. If you use the **Implements** statement, it must immediately follow any **Inherits** statements after the **Class** statement, and you must implement every member defined by every interface you specify.

interfacenames
Required if the **Implements** statement is used. The names of the interfaces implemented by this class.

statements
Optional. Statements which comprise the variables, properties, events, methods and nested types of the class.

End Class
Terminates the **Class** block.

Each attribute in the *attrlist* part has the following syntax and parts:

```
attrname [({ attrargs | attrinit })]
```

***attrlist* Parts**

attrname
Required. Name of the attribute. Must be a valid Visual Basic identifier.
attrargs
Optional. List of positional arguments for this attribute. Multiple arguments are separated by commas.
attrinit
Optional. List of field or property initializers for this attribute. Multiple initializers are separated by commas.

Remarks

Classes that do not specify an access modifier are declared as **Friend** by default. Within a **Class** block, members are declared as either **Public**, **Private**, **Protected**, **Friend**, or **Protected Friend** using the appropriate declaration statements. Anything declared as **Private** is visible only within the **Class** block. Anything declared as **Public** is visible within the **Class** block, as well as by code outside the **Class** block. Anything not explicitly declared is **Public** by default, except for fields and constants, which default to **Private**. **Public** variables, sometimes called Fields, serve as properties of the class, as do properties explicitly declared using **Property** declarations. Default properties and methods for the class are specified in their declarations using the **Default** keyword. See the individual declaration statement topics for information on how this keyword is used.

Binding of unqualified names in nested classes will search the members of the class itself, then the members of its containing class, and so on out to the outermost containing class. Private members of outer classes can be referenced but an error will be given for references to instance members of containing classes.

Nested classes cannot inherit from their containing class.

Example

This example uses the **Class** Statement to define a class, in which variables, properties, methods and events can be created.

```
Public Class ThisClass
  ' [variable, property, method and event declarations]
End Class
```

See Also

Inherits Statement | Implements Statement | Interface Statement | Property Statement |
Object Lifetime: How Objects are Created and Destroyed | Understanding Classes

Const Statement

Used at module, class, structure, procedure, or block level to declare constants for use in place of literal values.

```
[ <attrlist> ] [{ Public | Protected | Friend | Protected Friend |
Private }] [ Shadows ] Const name [ As type ] = initexpr
```

Parts

attrlist
Optional. List of attributes that apply to the constants declared in this statement. Multiple attributes are separated by commas.

Public
Optional. Constants declared with the **Public** keyword have public access. There are no restrictions on the accessibility of public constants.

Protected
Optional. Constants declared with the **Protected** keyword have protected access. They are accessible only from within their own class or from a derived class. Protected access can be specified only on members of classes. It is not a superset of friend access.

Friend
Optional. Constants declared with the **Friend** keyword have friend access. They are accessible from within their declaration context and from anywhere else in the same program.

Protected Friend
Optional. Constants declared with the **Protected Friend** keywords have the union of protected and friend access. They can be used by code in the same program, as well as by code in derived classes. Protected friend access can be specified only on members of classes.

Private
Optional. Constants declared with the **Private** keyword have private access. They are accessible only from within their declaration context, including from members of any nested types such as procedures.

Shadows
Optional. Indicates that this constant shadows an identically named programming element, or set of overloaded elements, in a base class. You can shadow any kind of declared element with any other kind. A shadowed element is unavailable from within the derived class that shadows it, unless the shadowing element is inaccessible, for example if it is **Private**.

name
Required. Name of the constant. Must be a valid Visual Basic identifier. You can declare as many constants as you like in the same declaration statement, specifying the *name* and *initexpr* parts for each one. Multiple constants are separated by commas.

type
Optional unless **Option Strict** is **On**. Data type of the constant. Can be **Boolean**, **Byte**, **Char**, **Date**, **Decimal**, **Double**, **Integer**, **Long**, **Object**, **Short**, **Single**, **String**, or the name of an enumeration. You must use a separate **As** clause for each constant being defined.

You cannot use variables or functions in *initexpr*. However, you can use the conversion keywords such as **CInt** and **CDate**, and you can use the **AscW** and **ChrW** methods of the **Strings** class in the **Microsoft.VisualBasic** namespace.

If you do not specify *type*, the constant takes the data type of *initexpr*. If you do specify *type*, the data type of *initexpr* must be convertible to *type*.

If *type* is **Object**, *initexpr* must be **Nothing**.

initexpr
Required. Expression that is evaluated and assigned to the constant when it is created. Can consist of a literal; another constant that is already defined; a member of an enumeration; or any combination of literals, constants, and enumeration members. You can use arithmetic and logical operators to combine such elements.

Each attribute in the *attrlist* part has the following syntax and parts:

```
attrname [({ attrargs | attrinit })]
```

***attrlist* Parts**

attrname
Required. Name of the attribute. Must be a valid Visual Basic identifier.

attrargs
Optional. List of positional arguments for this attribute. Multiple arguments are separated by commas.
attrinit
Optional. List of field or property initializers for this attribute. Multiple initializers are separated by commas.

Remarks

Constants declared in a **Sub**, **Function**, or **Property** procedure are local to that procedure. A constant declared outside a procedure is defined throughout the class, module, or structure in which it is declared.

You can use a constant anywhere you can use an expression.

Example

This example uses the **Const** statement to declare constants for use in place of literal values.

```
' Constants are shared by default.
Const MyVar As Integer = 459
' Declare Public String constant -- valid only at class or module level.
Public Const MyString As String = "HELP"
' Declare Private Integer constant -- valid only at class or module level.
Private Const MyInt As Integer = 5
```

See Also

Enum Statement | #Const Directive | Dim Statement | ReDim Statement

Declare Statement

Used at module level to declare references to external procedures in a dynamic-link library (DLL).

```
[ <attrlist> ] [ Public | Private | Protected | Friend | Protected Friend ] [ Shadows ] _
Declare [ Ansi | Unicode | Auto ] [ Sub ] name Lib "libname" _
[ Alias "aliasname" ] [([ arglist ])]
```

-or-

```
[ <attrlist> ] [ Public | Private | Protected | Friend | Protected Friend ] [ Shadows ] _
Declare [ Ansi | Unicode | Auto ] [ Function ] name Lib "libname" _
[ Alias "aliasname" ] [([ arglist ])] [ As type ]
```

Parts

attrlist
Optional. List of attributes that apply to this procedure. Multiple attributes are separated by commas.

Public
Optional. Entities declared with the **Public** modifier have public access. There are no restrictions on the use of public entities.

Private
Optional. Entities declared with the **Private** modifier have private access. A private entity is accessible only within its declaration context, including any nested entities.

Protected
Optional. Entities declared with the **Protected** keyword have protected access. They are accessible only from within their own class or from a derived class. Protected access can be specified only on members of classes. It is not a superset of friend access.

Friend
Optional. Entities declared with the **Friend** modifier have friend access. An entity with friend access is accessible only within the program that contains the entity declaration.

Protected Friend
Optional. Entities declared with the **Protected Friend** modifiers have the union of protected and friend accessibility.

Shadows
Optional. Indicates that this procedure shadows an identically named programming element in a base class. You can shadow any kind of declared element with any other kind. A shadowed element is unavailable from within the derived class that shadows it, unless the shadowing element is inaccessible, for example if it is **Private**.

Ansi
Optional. Converts all strings to ANSI values. If no modifier is specified, **Ansi** is the default.

Unicode
Optional. Converts all strings to Unicode values.

Auto
Optional. Converts the strings according to common language runtime rules based on the name of the method (or the alias name, if specified).

Sub
Optional (either **Sub** or **Function** must appear). Indicates that the procedure does not return a value.

Function
Optional (either **Sub** or **Function** must appear). Indicates that the procedure returns a value that can be used in an expression.

name
Required. Any valid procedure name. Note that DLL entry points are case sensitive.

Lib
Required. Indicates that a DLL or code resource contains the procedure being declared. The **Lib** clause is required for all declarations.

libname
Required. Name of the DLL or code resource that contains the declared procedure.

Alias
Optional. Indicates that the procedure being called has another name in the DLL. This is useful when the external procedure name is the same as a keyword. You can also use **Alias** when a DLL procedure has the same name as a public variable, constant, or any other procedure in the same scope. **Alias** is also useful if any characters in the DLL procedure name are not allowed by the DLL naming convention.

aliasname

Optional. Name of the procedure in the DLL or code resource. If the first character is not a number sign (**#**), *aliasname* is the name of the procedure's entry point in the DLL. If (**#**) is the first character, all characters that follow must indicate the ordinal number of the procedure's entry point.

arglist
Optional. List of variables representing arguments that are passed to the procedure when it is called.

type
Optional. Data type of the value returned by a Function procedure; may be **Byte**, **Boolean**, **Char**, **Short**, **Integer**, **Long**, **Single**, **Double**, **Decimal**, **String** (variable length only), **Object**, or structure. Arrays of any type cannot be returned, but an object containing an array can.

> **Note** Functions declared with the **Declare** statement cannot return dates.

Each attribute in the *attrlist* part has the following syntax and parts:

```
attrname [({ attrargs | attrinit })]
```

attrlist Parts

attrname
Required. Name of the attribute. Must be a valid Visual Basic identifier.

attrargs
Optional. List of positional arguments for this attribute. Multiple arguments are separated by commas.

attrinit
Optional. List of field or property initializers for this attribute. Multiple initializers are separated by commas.

The *arglist* argument has the following syntax and parts:

```
[ <attrlist> ] [ Optional ] [ ByVal | ByRef ] [ ParamArray ] varname[( )] [ As type ]
```

arglist Parts

attrlist
Optional. List of attributes that apply to this argument. Multiple attributes are separated by commas.

Optional
Optional. Indicates that an argument is not required. If used, all subsequent arguments in *arglist* must also be optional and declared using the **Optional** keyword. **Optional** can't be used for any argument if **ParamArray** is used.

ByVal
Optional. Indicates that the argument is passed by value. **ByVal** is the default in Visual Basic.

ByRef
Optional. Indicates that the argument is passed by reference.

ParamArray
Optional. Used only as the last argument in *arglist* to indicate that the final argument is an **Optional** array of **Object** elements. The **ParamArray** keyword allows you to provide an arbitrary number of arguments. The **ParamArray** keyword can't be used with **ByVal**, **ByRef**, or **Optional**.

varname
Required. Name of the variable representing the argument being passed to the procedure; follows standard variable naming conventions.

( )
Required for array variables. Indicates that *varname* is an array.

type
Optional. Data type of the argument passed to the procedure; may be **Byte**, **Boolean**, **Char**, **Short**, **Integer**, **Long**, **Single**, **Double**, **Decimal**, **String** (variable length only), **Object**, a user-defined type, or an object type.

Remarks

For **Function** procedures, the data type of the procedure determines the data type it returns. You can use an **As** clause following *arglist* to specify the return type of the function. Within *arglist*, you can use an **As** clause to specify the data type of any of the arguments passed to the procedure.

> **Note** If the external procedure was created on a platform other than Visual Basic .NET, you must take care that the data types correspond. For example, if you declare a reference to a Visual Basic 6.0 procedure with an **Integer**

argument (two bytes in Visual Basic 6.0), you must identify that argument as **Short** in the **Declare** statement, because that is the two-byte integer type in Visual Basic .NET.

Example

The following example declares a function that returns the current user name.

```
Declare Function GetUserName Lib "advapi32.dll" Alias _
"GetUserNameA" (ByVal lpBuffer As String, ByRef nSize As Integer) As Integer
Function GetUser()
   Dim RetVal As Integer
   Dim UserName As String
   Dim Buffer As String
   Buffer = New String(CChar(" "), 25)
   RetVal = GetUserName(Buffer, 25)
   UserName = Strings.Left(Buffer, InStr(Buffer, Chr(0)) - 1)
   MsgBox(UserName)
End Function
```

The **DllImport** attribute provides an alternative way of using functions in unmanaged code. The following example declares an imported function without using a **Declare** statement.

```
' Add an Imports statement at the top of the class or module
' where the DllImport attribute will be used.
Imports System.Runtime.InteropServices
'...
<DllImport("KERNEL32.DLL", EntryPoint:="MoveFileW", SetLastError:=True, _
CharSet:=CharSet.Unicode, ExactSpelling:=True, _
CallingConvention:=CallingConvention.StdCall)> _
Public Shared Function _
MoveFile(ByVal src As String, ByVal dst As String) As Boolean
   ' This function copies a file from the path src to the path dst.
   ' Leave function empty - DLLImport attribute forces calls
   ' to MoveFile to be forwarded to MoveFileW in KERNEL32.DLL.
End Function
```

See Also

DllImportAttribute Class | AddressOf Operator | Call Statement | Function Statement | LastDLLError Property | Sub Statement | Integer Data Type Changes in Visual Basic | Walkthrough: Calling Windows APIs

Delegate Statement

Used to declare a delegate. Delegates are a reference type that refers to a **Shared** method of a type or to an instance method of an object. Any procedure with matching parameters types and return type may be used to create an instance of this delegate class. The procedure can then later be invoked via the delegate instance.

```
[ <attrlist> ] [ Public | Private | Protected | Friend | Protected Friend ] _
[ Shadows ] Delegate [ Sub ] name [([ arglist ])]
```

-or-

```
[ <attrlist> ] [ Public | Private | Protected | Friend | Protected Friend ] _
[ Shadows ] Delegate [ Function ] name [([ arglist ])] As type
```

Parts

attrlist
Optional. List of attributes that apply to this delegate. Multiple attributes are separated by commas.

Public
Optional. Entities declared with the **Public** modifier have public access. There are no restrictions on the use of public entities.

Private
Optional. Entities declared with the **Private** modifier have private access. A private entity is accessible only within its declaration context, including any nested entities.

Protected
Optional. Entities declared with the **Protected** keyword have protected access. They are accessible only from within their own class or from a derived class. Protected access can be specified only on members of classes. It is not a superset of friend access.

Friend
Optional. Entities declared with the **Friend** modifier have friend access. An entity with friend access is accessible only within the program that contains the entity declaration.

Protected Friend
Optional. Entities declared with the **Protected Friend** modifiers have the union of protected and friend accessibility.

Shadows
Optional. Indicates that this delegate shadows an identically named programming element in a base class. You can shadow any kind of declared element with any other kind. A shadowed element is unavailable from within the derived class that shadows it, unless the shadowing element is inaccessible, for example if it is **Private**.

Sub
Optional, but either **Sub** or **Function** must appear. Declares this procedure as a delegate subroutine that does not return a value.

Function
Optional, but either **Sub** or **Function** must appear. Declares this procedure as a delegate function that returns a value.

name
Required. Name of the delegate type; follows standard variable naming conventions.

arglist
Optional. List of variables representing arguments that are passed to the procedure when it is called.

type
Optional. Data type of the return value.

Each attribute in the *attrlist* part has the following syntax and parts:

```
attrname [({ attrargs | attrinit })]
```

***attrlist* Parts**

attrname
Required. Name of the attribute. Must be a valid Visual Basic identifier.

attrargs
Optional. List of positional arguments for this attribute. Multiple arguments are separated by commas.

attrinit
Optional. List of field or property initializers for this attribute. Multiple initializers are separated by commas.

Each argument in the *arglist* part has the following syntax and parts:

```
[ <attrlist> ] [ ByVal | ByRef ] varname[( )] [ As type ]
```

arglist Parts

attrlist
Optional. List of attributes that apply to this argument. Multiple attributes are separated by commas.

ByVal
Optional. Indicates that the argument is passed by value. **ByVal** is the default in Visual Basic.

ByRef
Optional. Indicates that the argument is passed by reference.

varname
Required. Name of the variable representing the argument being passed to the procedure; follows standard variable naming conventions.

( )
Required for array variables. Indicates that *varname* is an array.

type
Optional. Data type of the argument passed to the procedure; may be **Byte**, **Boolean**, **Char**, **Short**, **Integer**, **Long**, **Single**, **Double**, **Decimal**, **Date**, **String** (variable length only), **Object**, a user-defined type, or an object type.

Remarks

The **Delegate** statement defines the parameter types and return type of a delegate class. Any procedure with matching parameters types and return type may be used to create an instance of this delegate class. The procedure can then later be invoked via the delegate instance, by calling the delegate's **Invoke** method.

Delegates, which can be declared at the namespace, module, or class level, cannot be declared within procedures.

Each delegate class defines a constructor that is passed the specification of an object method. An argument to a delegate constructor must be an expression of the form:

```
AddressOf [<expression>.]<methodname>
```

The compile-time type of the *expression* must be the name of a class or an interface that contains a method of the specified name whose signature matches the signature of the delegate class. The *methodname* can be either a shared method or an instance method. The *methodname* is not optional even if you create a delegate for the default method of the class.

Example

This example uses the **Delegate** statement to declare a delegate for comparing two integers and returning a Boolean. The `SelectionSort` method takes an instance of a delegate of this type and uses it to sort an integer array.

```
Public Class SortClass
   Delegate Function Compare(ByVal x As Integer, _
                             ByVal y As Integer) As Boolean

   Function CompareValues(ByVal X As Integer, _
                          ByVal Y As Integer) As Boolean
      If X > Y Then
         CompareValues = True
      Else
         CompareValues = False
      End If
   End Function

   Sub SelectionSort(ByVal IsGreaterThan As Compare, _
                     ByVal IntArray() As Integer)
      Dim MaxVal As Integer
      Dim MaxIndex As Integer
      Dim i, j As Integer

      ' Step through the elements in the array starting with the
```

```
        ' last element in the array.
        For i = UBound(IntArray) To 1 Step -1
           MaxVal = IntArray(i)
           MaxIndex = i
           For j = 1 To i
              ' Use the delegate to compare values.
              If IsGreaterThan.Invoke(IntArray(j), MaxVal) Then
                 MaxVal = IntArray(j)
                 MaxIndex = j
              End If
           Next j
           ' Use the delegate to compare values.
           If IsGreaterThan.Invoke(i, MaxIndex) Then
              IntArray(MaxIndex) = IntArray(i)
              IntArray(i) = MaxVal
           End If
        Next i
     End Sub
  End Class

  Class Class1
     Sub SortArray()
        Dim Sort As New SortClass()
        Dim arr1() As Integer = {1, 5, 3, 2, 7, 22, 5, 54, 12}
        Sort.SelectionSort(AddressOf Sort.CompareValues, arr1)
        MsgBox("Array sorted.")
     End Sub
  End Class

  ' Add a button to your main form and insert the following code
  ' into the Click event handlr.
     Private Sub Button1_Click(ByVal sender As System.Object, _
                               ByVal e As System.EventArgs) _
                               Handles Button1.Click
        Dim c As New Class1()
        c.SortArray()
     End Sub
```

See Also

AddressOf Operator | Events and Delegates

Dim Statement

Used at module, class, structure, procedure, or block level to declare and allocate storage space for variables.

```
[ <attrlist> ] [{ Public | Protected | Friend | Protected Friend |
Private | Static }] [ Shared ] [ Shadows ] [ ReadOnly ]
Dim [ WithEvents ] name[ (boundlist) ] [ As [ New ] type ] [ = initexpr ]
```

Parts

attrlist
Optional. List of attributes that apply to the variables declared in this statement. Multiple attributes are separated by commas.

Public
Optional. Variables declared with the **Public** keyword have public access. There are no restrictions on the accessibility of public variables.

You can use **Public** only at module, namespace, or file level. This means you can declare public variables in a source file or inside a module, class, or structure, but not inside a procedure. If you specify **Public**, you can optionally omit the **Dim** keyword.

Protected
Optional. Variables declared with the **Protected** keyword have protected access. They are accessible only from within their own class or from a derived class. Protected access is not a superset of friend access.

You can use **Protected** only at class level. This means you can declare protected variables inside a class but not inside a procedure, and not at module, namespace, or file level. You can use **Protected** only to declare members of the class. If you specify **Protected**, you can optionally omit the **Dim** keyword.

Friend
Optional. Variables declared with the **Friend** keyword have friend access. They are accessible from within their declaration context and from anywhere else in the same program.

You can use **Friend** only at module, namespace, or file level. This means you can declare friend variables in a source file or inside a module, class, or structure, but not inside a procedure. If you specify **Friend**, you can optionally omit the **Dim** keyword.

Protected Friend
Optional. Variables declared with the **Protected Friend** keywords have the union of protected and friend access. They can be used by code anywhere in the same program, by code in their own class, and by code in any derived classes.

You can use **Protected Friend** only at class level. This means you can declare protected friend variables inside a class but not inside a procedure, and not at module, namespace, or file level. You can use **Protected Friend** only to declare members of the class. If you specify **Protected Friend**, you can optionally omit the **Dim** keyword.

Private
Optional. Variables declared with the **Private** keyword have private access. They are accessible only from within their declaration context, including from members of any nested types such as procedures.

You can use **Private** only at module level. This means you can declare private variables inside a module, class, or structure, but not at namespace or file level and not inside a procedure. If you specify **Private**, you can optionally omit the **Dim** keyword.

Static
Optional. Variables declared with the **Static** keyword remain in existence and retain their latest values after termination of the procedure in which they are declared.

You can use **Static** only at procedure level. This means you can declare static variables inside a procedure or a block within a procedure, but not at class or module level. If you specify **Static**, you can optionally omit the **Dim** keyword.

You cannot specify **Static** together with either **Shared** or **Shadows** in the same variable declaration.

Shared
Optional. Indicates that this variable is shared. This means it is not associated with a specific instance of a class or structure. You can access a shared variable by qualifying it either with the class or structure name, or with the variable name of a specific instance of the class or structure.

You can use **Shared** only at module, namespace, or file level. This means you can declare shared variables in a source file or inside a module, class, or structure, but not inside a procedure. If you specify **Shared**, you can optionally omit the **Dim** keyword.

You cannot specify both **Static** and **Shared** in the same variable declaration.

Shadows

Optional. Indicates that this variable shadows an identically named programming element, or set of overloaded elements, in a base class. You can shadow any kind of declared element with any other kind. A shadowed element is unavailable from within the derived class that shadows it, unless the shadowing element is inaccessible, for example if it is **Private**.

You can use **Shadows** only at module, namespace, or file level. This means you can declare shadowing variables in a source file or inside a module, class, or structure, but not inside a procedure. If you specify **Shadows**, you can optionally omit the **Dim** keyword.

You cannot specify both **Static** and **Shadows** in the same variable declaration.

ReadOnly

Optional. Variables declared with the **ReadOnly** keyword can only be read and not written. This can be useful for creating constant members of reference types, such as an object variable with preset data members.

You can use **ReadOnly** only at module, namespace, or file level. This means you can declare read-only variables in a source file or inside a module, class, or structure, but not inside a procedure. If you specify **ReadOnly**, you can optionally omit the **Dim** keyword.

WithEvents

Optional. Keyword that specifies that *name* is an object variable that refers to an instance of a class that can raise events. You can declare as many individual variables as you like using **WithEvents**, but you cannot declare arrays this way.

If you use the **WithEvents** keyword, you cannot declare *name* as **Object**. You must declare it as the specific class that can raise the events.

name

Required. Name of the variable. Must be a valid Visual Basic identifier. You can declare as many variables as you like in the same declaration statement, specifying the *name* part for each one and supplying the *boundlist* part for arrays. Multiple variables are separated by commas.

You can declare several variables to be of the same data type. You can also specify different types for different variables or groups of variables. Each variable takes the data type specified in the first **As** clause encountered after its *name* part.

boundlist

Optional. List of non-negative integers representing the upper bounds of the dimensions of an array variable. Multiple upper bounds are separated by commas. An array can have up to 60 dimensions.

Each value in *boundlist* specifies the upper bound of a dimension, not the length. The lower bound is always zero, so the subscript for each dimension can vary from zero through the upper-bound value.

It is possible to use -1 to declare the upper bound of an array dimension. This signifies that the array is empty but not **Nothing**, a distinction required by certain common language runtime functions. However, Visual Basic code cannot successfully access such an array. If you attempt to do so, an **IndexOutOfRangeException** error occurs during execution.

New

Optional. Keyword that enables immediate creation of an object. If you use **New** when declaring the object variable, a new instance of the object is created when the **Dim** statement is executed.

type

Optional unless **Option Strict** is **On**. Data type of the variable. Can be **Boolean**, **Byte**, **Char**, **Date**, **Decimal**, **Double**, **Integer**, **Long**, **Object**, **Short**, **Single**, or **String**; or the name of an enumeration, structure, class, or interface. You can use a separate **As** clause for each variable being defined, or you can define several variables to be of the same type by using a common **As** clause.

If you do not specify *type*, the variable takes the data type of *initexpr*. If you do specify *type*, the data type of *initexpr* must be convertible to *type*. If neither *type* nor *initexpr* is present, the data type defaults to **Object**.

initexpr

Optional. Expression that is evaluated and assigned to the variable when it is created. If you declare more than one variable with the same **As** clause, you cannot supply *initexpr* for that group of variables.

Each attribute in the *attrlist* part has the following syntax and parts:

```
attrname [({ attrargs | attrinit })]
```

***attrlist* Parts**

attrname
Required. Name of the attribute. Must be a valid Visual Basic identifier.
attrargs
Optional. List of positional arguments for this attribute. Multiple arguments are separated by commas.
attrinit
Optional. List of field or property initializers for this attribute. Multiple initializers are separated by commas.

Remarks

Variables declared with **Dim** are available to all code within the region containing the **Dim** statement. If they are declared in a module, class, or structure, but outside any procedure, they can be accessed from anywhere within that module, class, or structure. To specify their accessibility in more detail, include the **Public**, **Protected**, **Friend**, **Protected Friend**, or **Private** keywords.

```
Private TryCount As Integer
Protected Salary As Decimal
Friend ValidDate As Date
Public MaximumAllowed As Double
Protected Friend CurrentUserName As String
```

Variables declared inside a procedure or a block are accessible only from within that procedure or block. You cannot use any of the accessibility keywords for such variables. Their declaration statements must use either **Dim** or **Static**.

```
Dim Counter As Integer
Static TotalSales As Double
```

The **Dim** statement can declare the data type of a variable and initialize its contents. The declaration statements in the following example declare an **Integer** variable, a **Boolean** variable, and an object variable. These are initialized, respectively, to 10, **True**, and a newly created instance of the **Label** class in the **System.Windows.Forms** namespace.

```
Dim Quantity As Integer = 10
Dim FirstTry As Boolean = True
Dim MyLabel As New System.Windows.Forms.Label
```

If you do not specify an initialization value for a variable, Visual Basic initializes it to the default value for its data type. The default initialization values are as follows:

- 0 for all numeric types (including **Byte**).
- Binary 0 for **Char.**
- **Nothing** for all reference types (including **Object**, **String**, and all arrays).
- **False** for **Boolean.**
- 12:00 AM of January 1 of the year 1 for **Date.**

Each element of a structure or array is initialized as if it were a separate variable.

You can initialize the values of an array by surrounding the initialization values with braces (**{}**). The following example creates an array with four elements:

```
Dim A1() As Integer = {0, 1, 2, 3}
```

For multidimensional arrays, each separate dimension is surrounded with braces. The elements are specified in row-major order. The following example illustrates this:

```
' Initialize each array element to its row index + its column index.
Dim A2( , ) As Integer = {{0+0, 0+1, 0+2}, {1+0 ,1+1, 1+2}}
```

> **Note** You should put all your declaration statements at the beginning of the code region in which they appear, for example a module or a procedure. If a declaration statement initializes the value of a variable, it should be executed before any other statement makes reference to the variable.

A public variable can be either early bound or late bound. Protected, friend, and private variables cannot be late bound. This means you cannot access members on a late-bound object variable unless it is declared **Public**. If you attempt to do this, the compiler does not generate an error, but a **MissingMemberException** error occurs at run time.

A static variable has a longer lifetime than that of the procedure in which it is declared. The boundaries of the variable's lifetime depend on where the procedure is declared and whether or not it is **Shared**.

- If the procedure is declared in a module, its static variables are initialized the first time the procedure is called, and they continue to exist until your program terminates execution.
- If the procedure is declared as **Shared** within a class or structure, its static variables are initialized the first time the procedure is called either on a specific instance or on the class or structure itself. They remain in existence until the program terminates.
- If the procedure is declared within a class or structure and is not **Shared**, its static variables are associated with each individual instance of the class or structure. They are initialized the first time the procedure is called in that instance, and they continue to exist until that instance is released.

Example

This example uses the **Dim** statement to declare variables and arrays. The lower bound for array subscripts is always 0, and the upper bound is the value that appears in the **Dim** statement.

```
' Declarations do not have to initialize their variables.
' AnyValue and MyValue are initialized by default to Nothing.
Dim AnyValue, MyValue As Object
' Number is initialized by default to 0.
Dim Number As Integer
' Multiple declarations can be of the same data type or different types.
Dim FirstNumber, SecondNumber, ThirdNumber As Integer
Dim MyDate As Date, MyValue As Single, MySwitch As Boolean
' DayArray is an array of 51 Objects indexed from 0 through 50.
Dim DayArray(50) As Object
' Matrix2 is a two-dimensional array of type Integer.
Dim Matrix2(3, 4) As Integer
' Matrix3 is a three-dimensional array of type Double.
Dim Matrix3(5, 9, 5) As Double
' BirthDay is an array of 11 dates indexed from 0 through 10.
Dim BirthDay(10) As Date
' MyArray is an array of objects. The length of the array is assigned
' when another statement assigns an array object to MyArray.
Dim MyArray() As Object
```

See Also

Const Statement | ReDim Statement | Variable Declaration | Object Variables | Events and Delegates | Declaring Array Variables

Do...Loop Statements

Repeats a block of statements while a **Boolean** condition is **True** or until the condition becomes **True**.

```
Do { While | Until } condition
   [ statements ]
[ Exit Do ]
   [ statements ]
Loop
```

-or-

```
Do
   [ statements ]
[ Exit Do ]
   [ statements ]
Loop { While | Until } condition
```

Parts

While
Required unless **Until** is used. Keyword. Repeat the loop until *condition* is **False**.
Until
Required unless **While** is used. Keyword. Repeat the loop until *condition* is **True**.
condition
Optional. **Boolean**. Expression that evaluates to a value of **True** or **False**.
statements
Optional. One or more statements that are repeated while, or until, *condition* is **True**.

Remarks

The **Exit Do** statement transfers control immediately to the statement following the **Loop** statement. Any number of **Exit Do** statements can be placed anywhere in the **Do** loop. **Exit Do** is often used after evaluating some condition, for example with **If...Then...Else**.

Example

This example shows how **Do...Loop** statements can be used. The inner **Do...Loop** statement loops 10 times, sets the value of the flag to **False**, and exits prematurely using the **Exit Do** statement. The outer loop exits immediately upon checking the value of the flag.

```
Dim Check As Boolean = True
Dim Counter As Integer = 0
Do   ' Outer loop.
   Do While Counter < 20   ' Inner loop.
      Counter += 1   ' Increment Counter.
      If Counter = 10 Then   ' If condition is True,
         Check = False   ' Set value of flag to False.
         Exit Do   ' Exit inner loop.
      End If
   Loop
Loop Until Check = False   ' Exit outer loop immediately.
```

See Also

Exit Statement | For...Next Statements | Faster For...Next Loops | Do...Loop Statements (Conceptual) | While...End While Statements

End Statement

Terminates execution immediately.

```
End
```

Remarks

The **End** statement can be placed anywhere in a procedure to end code execution, close files opened with an **Open** statement, and clear variables. The **End** statement calls the **Exit** method of the **Environment** class in the **System** namespace. **System.Environment.Exit** requires that you have **SecurityPermissionFlag.UnmanagedCode** permissions. If you do not, a **SecurityException** error occurs.

When executed, the **End** statement clears all variables at module and class level and all static local variables in all modules.

> **Note** The **End** statement stops code execution abruptly, without invoking the **Finalize** method or any other Visual Basic code. Object references held by other programs are invalidated.

The **End** statement provides a way to force your program to halt. Your program closes as soon as there are no other programs holding references to its objects and none of its code is executing.

If your application has any forms open, you should close them before executing **End**. A console application can either use **End** or simply return from the **Main** procedure.

With an additional keyword, **End** delineates the end of the definition of the appropriate procedure or block.

Example

This example uses the **End** Statement to terminate code execution if the user requests it.

```
Sub Form_Load()
   ' ...
   Dim Answer As MsgBoxResult
   Answer = MsgBox("Do you want to quit now?", MsgBoxStyle.YesNo)
   If Answer = MsgBoxResult.Yes Then
      MsgBox("Terminating program")
      End
   End If
   ' ...
End Sub
```

Smart Device Developer Notes

This statement is not supported.

See Also

End | Requesting Minimum Permissions | SecurityPermissionFlag Enumeration

Enum Statement

Used at module, class, or structure level to declare an enumeration and define the values of its members.

```
[ <attrlist> ] [{ Public | Protected | Friend | Protected Friend |
Private }] [ Shadows ] Enum name [ As type ]
   [<attrlist1>] membname1 [ = initexpr1 ]
   [<attrlist2>] membname2 [ = initexpr2 ]
...
   [<attrlistn>] membnamen [ = initexprn ]
End Enum
```

Parts

attrlist
Optional. List of attributes that apply to the enumeration or to this member. Multiple attributes are separated by commas.

Public
Optional. Enumerations declared with the **Public** keyword have public access. There are no restrictions on the accessibility of public enumerations.

Protected
Optional. Enumerations declared with the **Protected** keyword have protected access. They are accessible only from within their own class or from a derived class. Protected access can be specified only on members of classes. It is not a superset of friend access.

Friend
Optional. Enumerations declared with the **Friend** keyword have friend access. They are accessible from within their declaration context and from anywhere else in the same program.

Protected Friend
Optional. Enumerations declared with the **Protected Friend** keywords have the union of protected and friend access. They can be used by code in the same program, as well as by code in derived classes. Protected friend access can be specified only on members of classes.

Private
Optional. Enumerations declared with the **Private** keyword have private access. They are accessible only from within their declaration context, including from members of any nested types such as procedures.

Shadows
Optional. Indicates that this enumeration shadows an identically named programming element, or set of overloaded elements, in a base class. You can shadow any kind of declared element with any other kind. A shadowed element is unavailable from within the derived class that shadows it, unless the shadowing element is inaccessible, for example if it is **Private**.

name
Required. Name of the enumeration. Must be a valid Visual Basic identifier. When you subsequently declare variables or arguments to be of this **Enum** type, you specify *name* for their data type.

type
Optional, even if **Option Strict** is **On**. Data type of the enumeration and all of its members. Can be **Byte**, **Integer**, **Long**, or **Short**. If you do not specify *type*, it defaults to **Integer**.

membname
Required. Name of this member of the enumeration. Must be a valid Visual Basic identifier.

The accessibility of every member is **Public**, and you cannot declare it otherwise. However, if the enumeration itself is declared with **Protected**, **Friend**, **Protected Friend**, or **Private** access, this limits the accessibility of the members.

initexpr
Optional. Expression that is evaluated and assigned to this member when the enumeration is created. Can consist of a literal; a constant that is already defined; another enumeration member; or any combination of literals, constants, and enumeration members. You can use arithmetic and logical operators to combine such elements.

You cannot use variables or functions in *initexpr*. However, you can use conversion keywords such as **CByte** and **CShort**. You can also use **AscW** if you call it with a constant **String** or **Char** argument, since that can be evaluated at compile time.

If you do not specify *initexpr*, the value assigned is either zero (if it is the first *membname*), or greater by one than the value of the immediately preceding *membname*.

End Enum
Terminates an **Enum** block.

Each attribute in the *attrlist* part has the following syntax and parts:

```
attrname [({ attrargs | attrinit })]
```

attrlist Parts

attrname
Required. Name of the attribute. Must be a valid Visual Basic identifier.
attrargs
Optional. List of positional arguments for this attribute. Multiple arguments are separated by commas.
attrinit
Optional. List of field or property initializers for this attribute. Multiple initializers are separated by commas.

Remarks

The **Enum** statement can appear only at module, namespace, or file level. This means you can declare enumerations in a source file or inside a module, class, or structure, but not inside a procedure. Once the **Enum** type is defined, it can be used to declare the type of variables, procedure arguments, and **Function** returns.

Enumerations can be accessed from anywhere within the module, class, or structure in which they are declared. An enumeration and all its members are **Public** by default. To specify the accessibility in more detail, include **Public**, **Protected**, **Friend**, **Protected Friend**, or **Private** in the **Enum** statement. The accessibility you specify applies to all the members as well as to the enumeration itself.

An enumeration is a related set of constants. The enumeration members between the **Enum** and **End Enum** statements are initialized to constant values. The defined values cannot be modified at run time. Values can include both positive and negative numbers, as the following example shows:

```
Enum SecurityLevel
   IllegalEntry = -1
   MinimumSecurity = 0
   MaximumSecurity = 1
End Enum
```

If the value of a member exceeds the allowable range for the underlying data type, or if you initialize any member to the maximum value allowed by the underlying data type, the compiler reports an error.

Enumeration variables are variables declared to be of an **Enum** type. Declaring a variable in this way helps you to control the values you assign to it. If **Option Strict** is **On**, you can assign only enumeration members to the enumeration variable. In this case, you can use the **CType** keyword to explicitly convert a numeric data type to an **Enum** type.

You must qualify every reference to an enumeration member, either with the name of an enumeration variable or with the enumeration name itself. For example, in the preceding example, you can refer to the first member as `SecurityLevel.IllegalEntry`, but not as `IllegalEntry`.

Example

This example uses the **Enum** statement to define a set of named constants. In this case, the constants are colors you might choose to design data entry forms for a database.

```
Public Enum InterfaceColors
   MistyRose   = &HE1E4FF&
   SlateGray   = &H908070&
   DodgerBlue  = &HFF901E&
   DeepSkyBlue = &HFFBF00&
   SpringGreen = &H7FFF00&
   ForestGreen = &H228B22&
   Goldenrod   = &H20A5DA&
   Firebrick   = &H2222B2&
End Enum
```

See Also

Erase Statement

Used to release array variables and deallocate the memory used for their elements.

```
Erase arraylist
```

Parts

arraylist
Required. List of array variables to be erased. Multiple variables are separated by commas.

Remarks

The **Erase** statement can appear only at procedure level. This means you can release arrays inside a procedure but not at class or module level.

The **Erase** statement is equivalent to assigning **Nothing** to each array variable.

Example

This example uses the **Erase** statement to clear two arrays and free their memory (1000 and 100 storage elements respectively). The **ReDim** statement then assigns a new array instance to the three-dimensional array.

```
Dim Int3DArray(9, 9, 9), Int2DArray(9, 9) As Integer
' ...
Erase Int3DArray, Int2DArray
' ...
ReDim Int3DArray(4, 4, 9)
```

See Also

Nothing | ReDim Statement

Error Statement

Simulates the occurrence of an error.

```
Error errornumber
```

Parts

errornumber
Required. Can be any valid error number.

Remarks

The **Error** statement is supported for backward compatibility. In new code, especially when creating objects, use the **Err** object's **Raise** method to generate run-time errors.

If *errornumber* is defined, the **Error** statement calls the error handler after the properties of the **Err** object are assigned the following default values:

Property	Value
Number	Value specified as argument to **Error** statement. Can be any valid error number.
Source	Name of the current Visual Basic project.
Description	String expression corresponding to the return value of the **Error** function for the specified **Number**, if this string exists. If the string doesn't exist, **Description** contains a zero-length string (").
HelpFile	The fully qualified drive, path, and file name of the appropriate Visual Basic Help file.
HelpContext	The appropriate Visual Basic Help file context ID for the error corresponding to the **Number** propert y.
LastDLLError	Zero.

If no error handler exists or if none is enabled, an error message is created and displayed from the **Err** object properties.

> **Note** Some Visual Basic host applications cannot create objects. See your host application's documentation to determine whether it can create classes and objects.

Example

This example uses the **Error** statement to generate error number 11.

```
On Error Resume Next   ' Defer error handling.
Error 11   ' Simulate the "Division by zero" error.
```

Requirements

Namespace: Microsoft.VisualBasic

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Clear Method | Err Object | On Error Statement | Raise Method | Resume Statement | Visual Basic Run-Time Errors

Event Statement

Declares a user-defined event.

```
[ <attrlist> ] [ Public | Private | Protected | Friend | Protected Friend ] _
[ Shadows ] Event eventname[(arglist)] _
[ Implements interfacename.interfaceeventname ]
```

Parts

attrlist
Optional. List of attributes that apply to this event. Multiple attributes are separated by commas.

Public
Optional. Entities declared with the **Public** modifier have public access. There are no restrictions on the use of public entities. Events that do not specify an access modifier are declared as **Public** by default.

Private
Optional. Entities declared with the **Private** modifier have private access. A private entity is accessible only within its declaration context, including any nested entities.

Protected
Optional. Entities declared with the **Protected** keyword have protected access. They are accessible only from within their own class or from a derived class. Protected access can be specified only on members of classes. It is not a superset of friend access.

Friend
Optional. Entities declared with the **Friend** modifier have friend access. An entity with friend access is accessible only within the program that contains the entity declaration.

Protected Friend
Optional. Entities declared with the **Protected Friend** modifiers have the union of protected and friend accessibility.

Shadows
Optional. Indicates that this event shadows an identically named programming element in a base class. You can shadow any kind of declared element with any other kind. A shadowed element is unavailable from within the derived class that shadows it, unless the shadowing element is inaccessible, for example if it is **Private**.

eventname
Required. Name of the event; follows standard variable naming conventions.

Implements
Optional. Indicates that this event implements an event of an interface.

interfacename
The name of an interface.

interfaceeventname
The name of the event being implemented.

Each attribute in the *attrlist* part has the following syntax and parts:

```
attrname [({ attrargs | attrinit })]
```

***attrlist* Parts**

attrname
Required. Name of the attribute. Must be a valid Visual Basic identifier.

attrargs
Optional. List of positional arguments for this attribute. Multiple arguments are separated by commas.

attrinit
Optional. List of field or property initializers for this attribute. Multiple initializers are separated by commas.

The *arglist* argument has the following syntax and parts:

```
[ <attrlist> ] [ ByVal | ByRef ] varname[ ( ) ] [ As type ]
```

***arglist* Parts**

attrlist

Optional. List of attributes that apply to this argument. Multiple attributes are separated by commas.

ByVal

Optional. Indicates that the argument is passed by value. **ByVal** is the default in Visual Basic.

ByRef

Optional. Indicates that the argument is passed by reference.

varname

Required. Name of the variable representing the argument being passed to the procedure; follows standard variable naming conventions.

type

Optional. Data type of the argument passed to the procedure; may be **Byte**, **Boolean**, **Char**, **Short**, **Integer**, **Long**, **Single**, **Double**, **Decimal**, **Date**, **String** (variable length only), **Object**, a user-defined type, or an object type.

Remarks

Once the event has been declared, use the **RaiseEvent** statement to raise the event. A typical event might be declared and raised as shown in the following fragments:

```
Public Class EventSource
   ' Declare an event.
   Public Event LogonCompleted(ByVal UserName As String)
   Sub CauseEvent()
      ' Raise an event on successful logon.
      RaiseEvent LogonCompleted("AustinSteele")
   End Sub
End Class
```

Note You can declare event arguments just as you do arguments of procedures, with the following exceptions: events cannot have named arguments, or **Optional** arguments. Events do not have return values.

Example

The following example uses events to count off seconds during a demonstration of the fastest 100-meter race. The code illustrates all of the event-related methods, properties, and statements, including the **RaiseEvent** statement.

The class that raises an event is the event source, and the methods that process the event are the event handlers. An event source can have multiple handlers for the events it generates. When the class raises the event, that event is raised on every class that has elected to handle events for that instance of the object.

The example also uses a form (`Form1`) with a button (`Command1`), a label (`Label1`), and two text boxes (`Text1` and `Text2`). When you click the button, the first text box displays "From Now" and the second starts to count seconds. When the full time (9.84 seconds) has elapsed, the first text box displays "Until Now" and the second displays "9.84"

The code for `Form1` specifies the initial and terminal states of the form. It also contains the code executed when events are raised.

To use this example, open a new Windows Forms project, add a button named `Button1`, a label named `Label1` and two text boxes, named `TextBox1` and `TextBox2`, to the main form, named `form1`. Then right click the form and click **View Code** to open the code editor.

To simplify access to the **Timer** property, add an **Imports** statement as the first line of code above the `Class Form1` statement.

```
Imports Microsoft.VisualBasic.DateAndTime
```

Add a **WithEvents** variable to the declarations section of the `Form1` class.

```
Private WithEvents mText As TimerState
```

Add the following code to the code for `Form1`. Replace any duplicate procedures that may exist, such as `Form_Load`, or `Button_Click`.

```
Private Sub Form1_Load(ByVal sender As Object, _
                       ByVal e As System.EventArgs) _
                       Handles MyBase.Load
   Button1.Text = "Click to Start Timer"
```

```
    TextBox1.Text = ""
    TextBox2.Text = ""
    Label1.Text = "The fastest 100 meters ever run took this long:"
    mText = New TimerState()
End Sub
Private Sub Button1_Click(ByVal sender As System.Object, _
                          ByVal e As System.EventArgs) _
                          Handles Button1.Click
    TextBox1.Text = "From Now"
    TextBox1.Refresh()
    TextBox2.Text = "0"
    TextBox2.Refresh()
    mText.TimerTask(9.84)
End Sub

Private Sub mText_ChangeText() Handles mText.ChangeText
    TextBox1.Text = "Until Now"
    TextBox2.Text = "9.84"
End Sub

Private Sub mText_UpdateTime(ByVal Jump As Double) _
       Handles mText.UpdateTime
    TextBox2.Text = Format(Jump, "##0.00")
    Application.DoEvents()
End Sub

Class TimerState
    Public Event UpdateTime(ByVal Jump As Double)
    Public Event ChangeText()
    Public Sub TimerTask(ByVal Duration As Double)
       Dim Start As Double
       Dim Second As Double
       Dim SoFar As Double
       Start = Timer
       SoFar = Start
       Do While Timer < Start + Duration
          If Timer - SoFar >= 0.1 Then
             SoFar = SoFar + 0.1
             RaiseEvent UpdateTime(Timer - Start)
          End If
       Loop
       RaiseEvent ChangeText()
    End Sub
End Class
```

Press F5 to run this example, and click the button labeled **Click to start timer**. The first text box displays "From Now" and the second starts to count seconds. When the full time (9.84 seconds) has elapsed, the first text box displays "Until Now" and the second displays "9.84".

See Also

RaiseEvent Statement | Implements Statement | Adding Events to a Class | Events and Delegates

Exit Statement

Exits a procedure or block and transfers control immediately to the statement following the procedure call or the block definition.

```
Exit { Do | For | Function | Property | Select | Sub | Try | While }
```

Parts

Do
Immediately exits the **Do** loop in which it appears. Execution continues with the statement following the **Loop** statement. **Exit Do** can be used only inside a **Do** loop. When used within nested **Do** loops, **Exit Do** transfers control to the loop that is one nested level above the loop where **Exit Do** occurs.

For
Immediately exits the **For** loop in which it appears. Execution continues with the statement following the **Next** statement. **Exit For** can be used only inside a **For...Next** or **For Each...Next** loop. When used within nested **For** loops, **Exit For** transfers control to the loop that is one nested level above the loop where **Exit For** occurs.

Function
Immediately exits the **Function** procedure in which it appears. Execution continues with the statement following the statement that called the **Function** procedure. **Exit Function** can be used only inside a **Function** procedure.

Property
Immediately exits the **Property** procedure in which it appears. Execution continues with the statement that called the **Property** procedure, that is, with the statement requesting or setting the property's value. **Exit Property** can be used only inside a **Property** procedure.

Select
Immediately exits the **Select Case** in which it appears. Execution continues with the statement following the **End Select** statement. **Exit Select** can be used only inside a **Select Case** statement.

Sub
Immediately exits the **Sub** procedure in which it appears. Execution continues with the statement following the statement that called the **Sub** procedure. **Exit Sub** can be used only inside a **Sub** procedure.

Try
Immediately exits the **Try** or **Catch** block in which it appears. Execution continues with the **Finally** block if there is one, or with the statement following the **End Try** statement otherwise. **Exit Try** can be used only inside a **Try...Catch...Finally** statement.

While
Immediately exits the **While** loop in which it appears. Execution continues with the statement following the **End While** statement. **Exit While** can be used only inside a **While** loop. When used within nested **While** loops, **Exit While** transfers control to the loop that is one nested level above the loop where **Exit While** occurs.

Remarks

Do not confuse **Exit** statements with **End** statements. **Exit** does not define the end of a statement.

Example

This example uses the **Exit** statement to exit a **For...Next** loop, a **Do...Loop**, and a **Sub** procedure.

```
Sub ExitStatementDemo()
Dim I, MyNum As Integer
   Do   ' Set up infinite loop.
      For I = 1 To 1000   ' Loop 1000 times.
         MyNum = Int(Rnd * 1000)   ' Generate random numbers.
         Select Case MyNum   ' Evaluate random number.
            Case 7: Exit For   ' If 7, exit For...Next.
            Case 29: Exit Do   ' If 29, exit Do...Loop.
            Case 54: Exit Sub   ' If 54, exit Sub procedure.
         End Select
      Next I
   Loop
End Sub
```

See Also

Do...Loop Statements | End Statement | For Each...Next Statements | For...Next Statements | Function Statement | Stop Statement | Sub Statement | Try...Catch...Finally Statements | For...Next Statements (Conceptual) | For Each...Next Statements (Conceptual)

F-P Statements

The following table contains a listing of the Visual Basic .NET language statements.

For Each...Next	For...Next	Function	Get
GoTo	If...Then...Else	Implements	Imports
Inherits	Interface	Mid	Module
Namespace	On Error	Option Compare	Option Explicit
Option Strict	Property		

See Also

A-E Statements | Q-Z Statements | Visual Basic Language and Run-Time Reference

For Each...Next Statements

Repeats a group of statements for each element in a collection.

```
For Each element [ As datatype ] In group
   [ statements ]
[ Exit For ]
   [ statements ]
Next [ element ]
```

Parts

element
Required. Variable. Used to iterate through the elements of the collection. The data type of *element* must be such that the data type of the elements of *group* can be converted to it.

datatype
Required if *element* is not already declared. Data type of *element*. If *element* is declared outside this loop, you cannot use the **As** clause to redeclare it.

group
Required. Object variable. Must refer to an object collection or array.

statements
Optional. One or more statements between **For Each** and **Next** that are executed on each item in *group*.

Remarks

If *element* has not been declared outside this loop, you can declare it within the **For Each** statement. In this case, the scope of *element* is the body of the loop. However, you cannot declare *element* both outside and inside the loop.

The **For Each...Next** loop is entered if there is at least one element in *group*. Once the loop has been entered, the statements are executed for the first element in *group*; if there are more elements in *group*, the statements in the loop continue to execute for each element. When there are no more elements, the loop is terminated and execution continues with the statement following the **Next** statement.

Any number of **Exit For** statements may be placed anywhere in the loop as an alternative way to exit. **Exit For** is often used after evaluating some condition, for example with an **If...Then...Else** statement, and transfers control to the statement immediately following **Next**.

You can nest **For Each...Next** loops by placing one loop within another. Each loop must have a unique *element* variable.

Example

This example uses the **For Each...Next** statement to search the **Text** property of all elements in a collection for the existence of the string "Hello". In the example, `MyObject` is a text-related object and is an element of the collection `MyCollection`. Both are generic names used for illustration purposes only.

```
Dim Found As Boolean = False
Dim MyCollection As New Collection
For Each MyObject As Object In MyCollection   ' Iterate through elements.
   If CStr(MyObject.Text) = "Hello" Then   ' If Text equals "Hello"
      Found = True   ' Set Found to True.
      Exit For   ' Exit loop.
   End If
Next
```

See Also

For Each...Next Statements (Conceptual) | Collections in Visual Basic .NET | Do...Loop Statements | Exit Statement | For...Next Statements | Faster For...Next Loops | While...End While Statements | Do...Loop Statements (Conceptual) | For...Next Statements (Conceptual)

For...Next Statements

Repeats a group of statements a specified number of times.

```
For counter [ As datatype ] = start To end [ Step step ]
   [ statements ]
[ Exit For ]
   [ statements ]
Next [ counter ]
```

Parts

counter
Required. Variable. The data type of *counter* is usually **Integer** but can be any elementary numeric type that supports the greater than or equal to (**>=**), less than or equal to (**<=**), and addition (**+**) operators.

datatype
Required if *counter* is not already declared. Data type of *counter*. If *counter* is declared outside this loop, you cannot use the **As** clause to redeclare it.

start
Required. Expression. The initial value of *counter*. The *start* expression usually evaluates to type **Integer** but can evaluate to any data type that widens to the type of *counter*.

end
Required. Expression. The final value of *counter*. The *end* expression usually evaluates to type **Integer** but can evaluate to any data type that widens to the type of *counter*.

step
Optional. Expression. The amount by which *counter* is incremented each time through the loop. The *step* expression usually evaluates to type **Integer** but can evaluate to any data type that widens to the type of *counter*. If not specified, *step* defaults to 1.

statements
Optional. One or more statements between **For** and **Next** that are executed the specified number of times.

Remarks

If *counter* has not been declared outside this loop, you can declare it within the **For** statement. In this case, the scope of *counter* is the body of the loop. However, you cannot declare *counter* both outside and inside the loop.

The *step* argument can be either positive or negative. The value of the *step* argument determines loop processing as follows:

Step value	Loop executes if
Positive or zero	*counter* <= *end*
Negative	*counter* >= *end*

The expressions *start*, *end*, and *step* are all evaluated only once, when the **For** statement is first encountered. They are not evaluated again, even if the loop statements change their constituent parts.

The *counter* variable is compared to *end* every time before the loop is entered. This includes the first time the **For** statement is executed. Therefore, if the value of *start* is past the value of *end* when the loop is entered, the loop is not executed, and execution passes immediately to the statement following the **Next** statement.

After the loop statements have executed, *step* is added to *counter*. At this point, the **For** statement again compares *counter* to *end*. As a result of this comparison, either the statements in the loop execute again, or the loop is terminated and execution continues with the statement following the **Next** statement.

Note Changing the value of *counter* while inside a loop can make it more difficult to read and debug your code.

The **Exit For** statement transfers control immediately to the statement following the **Next** statement. Any number of **Exit For** statements can be placed anywhere in the **For...Next** loop. **Exit For** is often used after evaluating some condition, for example with an **If...Then...Else** statement.

You can *nest* **For...Next** loops by placing one loop within another. Each loop must have a unique *counter* variable. The following construction is correct:

```
For I As Integer = 1 To 10
   For J As Integer = 1 To 10
```

```
      For K As Integer = 1 To 10
         ' Statements to operate with current values of I, J, and K.
      Next K
   Next J
Next I
```

Note If a **Next** statement is encountered before its corresponding **For** statement, or if a **Next** statement of an outer nesting level is encountered before the **Next** of an inner level, an error occurs.

Example

This example uses the **For...Next** statement to create a string that contains 10 instances of the numbers 0 through 9, each string separated from the other by a single space. The outer loop uses a loop counter variable that is decremented each time through the loop.

```
Dim Words, Digit As Integer
Dim MyString As String
For Words = 10 To 1 Step -1    ' Set up 10 repetitions.
   For Digit = 0 To 9   ' Set up 10 repetitions.
      MyString = MyString & CStr(Digit)   ' Append number to string.
   Next Digit   ' Increment counter.
   MyString = MyString & " "   ' Append a space.
Next Words
```

See Also

For...Next Statements (Conceptual) | Do...Loop Statements | Exit Statement | For Each...Next Statements | While...End While Statements | Faster For...Next Loops | Do...Loop Statements (Conceptual) | For Each...Next Statements (Conceptual)

Function Statement

Declares the name, arguments, and code that define a **Function** procedure.

```
[ <attrlist> ] [{ Overloads | Overrides | Overridable |
NotOverridable | MustOverride | Shadows | Shared }]
[{ Public | Protected | Friend | Protected Friend | Private }]
Function name[(arglist)] [ As type ] [ Implements interface.definedname ]
   [ statements ]
   [ Exit Function ]
   [ statements ]
End Function
```

Parts

attrlist
Optional. List of attributes that apply to this procedure. Multiple attributes are separated by commas.

Overloads
Optional. Indicates that this **Function** procedure overloads one or more procedures defined with the same name in a base class. The argument list in this declaration must be different from the argument list of every overloaded procedure. The lists must differ in the number of arguments, their data types, or both. This allows the compiler to distinguish which version to use.

You do not have to use the **Overloads** keyword when you are defining multiple overloaded procedures in the same class. However, if you use **Overloads** in one of the declarations, you must use it in all of them.

You cannot specify both **Overloads** and **Shadows** in the same procedure declaration.

Overrides
Optional. Indicates that this **Function** procedure overrides an identically named procedure in a base class. The number and data types of the arguments, and the data type of the return value, must exactly match those of the base class procedure.

Overridable
Optional. Indicates that this **Function** procedure can be overridden by an identically named procedure in a derived class. **Overridable** is the default setting for a procedure that itself overrides a base class procedure.

NotOverridable
Optional. Indicates that this **Function** procedure cannot be overridden in a derived class. **NotOverridable** is the default setting for a procedure that does not itself override a base class procedure.

MustOverride
Optional. Indicates that this **Function** procedure is not implemented in this class, and must be implemented in a derived class for that class to be creatable.

Shadows
Optional. Indicates that this **Function** procedure shadows an identically named programming element, or set of overloaded elements, in a base class. You can shadow any kind of declared element with any other kind. If you shadow a procedure with another procedure, the arguments and the return type do not have to match those in the base class procedure. A shadowed element is unavailable from within the derived class that shadows it, unless the shadowing element is inaccessible, for example if it is **Private**.

You cannot specify both **Overloads** and **Shadows** in the same procedure declaration.

Shared
Optional. Indicates that this **Function** procedure is a shared procedure. This means it is not associated with a specific instance of a class or structure. You can call a shared procedure by qualifying it either with the class or structure name, or with the variable name of a specific instance of the class or structure.

Public
Optional. Procedures declared with the **Public** keyword have public access. There are no restrictions on the accessibility of public procedures.

Protected
Optional. Procedures declared with the **Protected** keyword have protected access. They are accessible only from within their own class or from a derived class. Protected access can be specified only on members of classes. It is not a superset of friend access.

Friend
Optional. Procedures declared with the **Friend** keyword have friend access. They are accessible from within their declaration context and from anywhere else in the same program.

Protected Friend
Optional. Procedures declared with the **Protected Friend** keywords have the union of protected and friend access. They can be used by code in the same program, as well as by code in derived classes. Protected friend access can be specified only on members of classes.

Private
Optional. Procedures declared with the **Private** keyword have private access. They are accessible only from within their declaration context, including from members of any nested types such as procedures.

name
Required. Name of the **Function** procedure. Must be a valid Visual Basic identifier.

arglist
Optional. List of variables or expressions representing arguments that are passed to the **Function** procedure when it is called. Multiple arguments are separated by commas. If you supply an argument list, you must enclose it in parentheses.

type
Optional unless **Option Strict** is **On**. Data type of the value returned by the **Function** procedure. Can be **Boolean**, **Byte**, **Char**, **Date**, **Decimal**, **Double**, **Integer**, **Long**, **Object**, **Short**, **Single**, or **String**; or the name of an enumeration, structure, class, or interface.

Implements
Optional. Indicates that this **Function** procedure implements a **Function** procedure defined by an interface.

interface
Required if **Implements** is supplied. The interface implemented by the class or structure containing this **Function** procedure. The class or structure must specify *interface* in an **Implements** statement immediately following the **Class** or **Structure** statement.

definedname
Required if **Implements** is supplied. The name by which the **Function** procedure is defined in *interface*. The name of this **Function** procedure (in *name*) does not have to be the same as *definedname*.

statements
Optional. A block of statements to be executed within the **Function** procedure.

End Function
Terminates the definition of this **Function** procedure.

Each argument in the *arglist* part has the following syntax and parts:

```
[ <attrlist> ] [ Optional ] [{ ByVal | ByRef }] [ ParamArray ]
argname[( )] [ As argtype ] [ = defaultvalue ]
```

***arglist* Parts**

attrlist
Optional. List of attributes that apply to this argument. Multiple attributes are separated by commas.

Optional
Optional. Indicates that this argument is not required when the procedure is called. If this keyword is used, all subsequent arguments in *arglist* must also be optional and be declared using the **Optional** keyword. Every optional argument declaration must supply the *defaultvalue* clause. **Optional** cannot be used for any argument if **ParamArray** is used.

ByVal
Optional. Indicates that the procedure cannot replace or reassign the underlying variable element in the calling code. However, if the argument is a reference type, the procedure can modify the contents or members of the underlying object. **ByVal** is the default in Visual Basic.

ByRef
Optional. Indicates that the procedure can modify the underlying variable in the calling code the same way the calling code itself can.

ParamArray
Optional. Used only as the last argument in *arglist* to indicate that the final argument is an optional array of elements of the specified type. The **ParamArray** keyword allows you to pass an arbitrary number of arguments to the procedure. A **ParamArray** argument is always passed using **ByVal**.

argname
Required. Name of the variable representing the argument. Must be a valid Visual Basic identifier.

argtype
Optional unless **Option Strict** is **On**. Data type of the argument passed to the procedure. Can be **Boolean**, **Byte**, **Char**, **Date**, **Decimal**, **Double**, **Integer**, **Long**, **Object**, **Short**, **Single**, or **String**; or the name of an enumeration, structure, class, or interface.

defaultvalue

Required for **Optional** arguments. Any constant or constant expression that evaluates to the data type of the argument. If the type is **Object**; or a class, interface, array, or structure, the default value can only be **Nothing**.

Each attribute in the *attrlist* part has the following syntax and parts:

```
attrname [({ attrargs | attrinit })]
```

attrlist Parts

attrname
Required. Name of the attribute. Must be a valid Visual Basic identifier.
attrargs
Optional. List of positional arguments for this attribute. Multiple arguments are separated by commas.
attrinit
Optional. List of field or property initializers for this attribute. Multiple initializers are separated by commas.

Remarks

All executable code must be in procedures. You can define a **Function** procedure inside a module, class, interface, or structure, but not inside another procedure.

Function procedures are **Public** by default. To specify a different accessibility, include **Protected**, **Friend**, **Protected Friend**, or **Private** in the declaration.

When the **Function** procedure returns to the calling program, execution continues with the statement following the statement that called it.

The **Exit Function** statement causes an immediate exit from a **Function** procedure. Any number of **Exit Function** statements can appear anywhere in the procedure.

To return a value from a function, you can either assign the value to the function name or include it in a **Return** statement. The following example assigns the return value to the function name `MyFunction` and then uses the **Exit Function** statement to return:

```
Function MyFunction(ByVal J As Integer) As Double
   ' ...
   MyFunction = 3.87
   ' ...
   Exit Function
   ' ...
End Function
```

If you use **Exit Function** without assigning a value to *name*, the function returns the default value appropriate to *argtype*. This is 0 for **Byte**, **Char**, **Decimal**, **Double**, **Integer**, **Long**, **Short**, and **Single**; **Nothing** for **Object**, **String**, and all arrays; **False** for **Boolean**; and #1/1/0001 12:00 AM# for **Date**.

The **Return** statement simultaneously assigns the return value and exits the function, as in the following example:

```
Function MyFunction(ByVal J As Integer) As Double
   ' ...
   Return 3.87
   ' ...
End Function
```

Any number of **Return** statements can appear anywhere in the procedure. You can also mix **Exit Function** and **Return** statements.

You can use a **Function** procedure on the right side of an expression when you want to use the value returned by the function. You use the **Function** procedure the same way you use any library function such as **Sqrt**, **Cos**, or **ChrW**.

You call a **Function** procedure by using the procedure name, followed by the argument list in parentheses, in an expression. You can omit the parentheses only if you are not supplying any arguments. A function can also be called using the **Call** statement, in which case the return value is ignored.

Note Visual Basic sometimes rearranges arithmetic expressions to increase internal efficiency. For that reason, avoid using a **Function** procedure in an arithmetic expression when the function changes the value of variables in the same expression.

Example

This example uses the **Function** statement to declare the name, arguments, and code that form the body of a **Function** procedure. The **ParamArray** keyword enables the function to accept a variable number of arguments.

```
Public Function CalcSum(ByVal ParamArray Args() As Double) As Double
   Dim I As Integer
   CalcSum = 0
   If Args.Length <= 0 Then Exit Function   ' No arguments passed.
   For I = 0 To UBound(Args, 1)
      CalcSum += Args(I)
   Next I
End Function   ' Returns latest value of CalcSum.
```

The function can be invoked as follows:

```
Dim ReturnedValue As Double
ReturnedValue = CalcSum(4, 3, 2, 1)
' The function's local variables are assigned the following values:
' Args(0) = 4, Args(1) = 3, and so on.
```

See Also

Call Statement | Sub Statement | Parameter Arrays | Dim Statement | Implements Statement

Get Statement

Declares a **Get** property procedure used to retrieve the value of a property.

```
[ <attrlist> ] Get()
   [ block ]
End Get
```

Parts

attrlist
Optional. List of attributes that apply to this **Get** block. Multiple attributes are separated by commas.
block
Optional. The body of the **Get** property procedure contains code to return the property value.
End Get
Terminates a **Get** property procedure.

Each attribute in the *attrlist* part has the following syntax and parts:

```
attrname [({ attrargs | attrinit })]
```

attrlist Parts

attrname
Required. Name of the attribute. Must be a valid Visual Basic identifier.
attrargs
Optional. List of positional arguments for this attribute. Multiple arguments are separated by commas.
attrinit
Optional. List of field or property initializers for this attribute. Multiple initializers are separated by commas.

Remarks

Get property procedures can return a value using either the **Return** keyword or by assigning the return value to the property where the get procedure is declared. The value returned by a **Get** property procedure is usually a private class level variable that stored the property value after a property set operation.

Example

This example uses the **Get** statement to return the value of a property.

```
Class PropClass
   ' Define a local variable to store the property value.
   Private CurrentTime As String
   ' Define the property.
   Public ReadOnly Property DateAndTime() As String
      Get
         ' Get procedure is called when the value
         ' of a property is retrieved.
         CurrentTime = CStr(Now)
         Return CurrentTime    ' Returns the date and time As a string.
      End Get
   End Property
End Class
```

See Also

Set Statement | Property Statement

GoTo Statement

Branches unconditionally to a specified line within a procedure.

```
GoTo line
```

Part

line
Required. Any line label.

Remarks

GoTo can branch only to lines within the procedure where it appears.

You cannot use a **GoTo** to branch from outside a **For...Next**, **For Each...Next**, **SyncLock...End SyncLock**, **Try...Catch...Finally**, or **With...End With** block to a label inside.

Within a **Try...Catch...Finally** construction, you cannot use **GoTo** to branch out of a **Try** block, into a **Catch** block, or into or out of a **Finally** block. You can branch from a **Catch** block into the **Try** block associated with that **Catch**. For example, if one **Try...Catch...Finally** construction is nested within another, a **Catch** block can branch into the **Try** block at its own nesting level, but not into any other **Try** block.

> **Note** **GoTo** statements can make code difficult to read and maintain. Whenever possible, use **Do...Loop**, **For...Next**, **If...Then...Else**, **Select Case**, **Try...Catch...Finally**, **While**, and **With...End With** structures instead.

Example

This example uses the **GoTo** statement to branch to line labels within a procedure.

```
Sub GotoStatementDemo()
Dim Number As Integer
dim MyString As String
   Number = 1   ' Initialize variable.
   ' Evaluate Number and branch to appropriate label.
   If Number = 1 Then GoTo Line1 Else GoTo Line2

Line1:
   MyString = "Number equals 1"
   GoTo LastLine   ' Go to LastLine.
Line2:
   ' The following statement never gets executed because Number = 1.
   MyString = "Number equals 2"
LastLine:
   ' Print "Number equals 1" in the Output window.
   Debug.WriteLine (MyString)
End Sub
```

See Also

Do...Loop Statements | For...Next Statements | If...Then...Else Statements | Select...Case Statements | Try...Catch...Finally Statements

If...Then...Else Statements

Conditionally executes a group of statements, depending on the value of an expression.

```
If condition [ Then ]
   [ statements  ]
[ ElseIf elseifcondition [ Then ]
   [ elseifstatements ] ]
[ Else
   [ elsestatements ] ]
End If
```

-or-

```
If condition Then [ statements ] [ Else elsestatements ]
```

Parts

condition
Required. Expression. The expression you supply for *condition* must evaluate to **True** or **False**, or to a data type that is implicitly convertible to **Boolean**.

statements
Optional in multiple-line form; required in single-line form that has no **Else** clause. One or more statements following **If...Then** that are executed if *condition* is **True**.

elseifcondition
Required if **ElseIf** is present. Same as *condition*.

elseifstatements
Optional. One or more statements following **ElseIf...Then** that are executed if the associated *elseifcondition* is **True**.

elsestatements
Optional in multiple-line form; required in single-line form that has an **Else** clause. One or more statements that are executed if no previous *condition* or *elseifcondition* expression is **True**.

End If
Terminates **If...Then** block.

Remarks

You can use the single-line form for short, simple tests. However, the multiple-line form provides more structure and flexibility than the single-line form and is usually easier to read, maintain, and debug.

With the single-line form, it is possible to have multiple statements executed as the result of an **If...Then** decision. All statements must be on the same line and be separated by colons, as in the following example:

```
If A > 10 Then A = A + 1 : B = B + A : C = C + B
```

In the multiple-line form, the **If** statement must be the only statement on the first line. The **Else**, **ElseIf**, and **End If** statements can be preceded only by a line label. The multiple-line **If...Then...Else** must end with an **End If** statement.

To determine whether or not an **If** statement introduces a multiple-line form, examine what follows the **Then** keyword. If anything other than a comment appears after **Then** in the same statement, the statement is treated as a single-line **If** statement. If **Then** is absent, it must be the beginning of a multiple-line **If...Then...Else**.

The **ElseIf** and **Else** clauses are both optional. You can have as many **ElseIf** clauses as you want in a multiple-line **If...Then...Else**, but none can appear after an **Else** clause. Multiple-line forms can be nested within one another.

When a multiple-line **If...Then...Else** is encountered, *condition* is tested. If *condition* is **True**, the statements following **Then** are executed. If *condition* is **False**, each **ElseIf** statement is evaluated in order. When a **True** *elseifcondition* is found, the statements immediately following the associated **Then** are executed. If no *elseifcondition* evaluates to **True**, or if there are no **ElseIf** statements, the statements following **Else** are executed. After executing the statements following **Then**, **ElseIf**, or **Else**, execution continues with the statement following **End If**.

Tip **Select Case** might be more useful when evaluating a single expression that has several possible values.

Example

This example shows both the multiple- and single-line forms of the **If...Then...Else** statement.

```
Dim Number, Digits As Integer
Dim MyString As String
Number = 53   ' Initialize variable.
If Number < 10 Then
   Digits = 1
ElseIf Number < 100 Then
' Condition evaluates to True so the next statement is executed.
   Digits = 2
Else
   Digits = 3
End If

' Assign a value using the single-line form of syntax.
If Digits = 1 Then MyString = "One" Else MyString = "More than one"
```

Use the **TypeOf** keyword to determine whether the **Control** object passed into a procedure is a text box.

```
Sub ControlProcessor(ByVal MyControl As Control)
   If TypeOf MyControl Is ComboBox Then
      Debug.WriteLine ("You passed in a " & TypeName(MyControl))
   ElseIf TypeOf MyControl Is CheckBox Then
      Debug.WriteLine ("You passed in a " & TypeName(MyControl))
   ElseIf TypeOf MyControl Is TextBox Then
      Debug.WriteLine ("You passed in a " & TypeName(MyControl))
   End If
End Sub
```

See Also

#If...Then...#Else Directives | Choose Function | Select...Case Statements | Switch Function | If...Then...Else Statements (Conceptual)

Implements Statement

Specifies one or more interfaces, or interface members, that will be implemented in the class or structure definition in which it appears.

```
Implements interfacename [, ...]
```

-or-

```
Implements interfacename.interfacemember [, ...]
```

Parts

interfacename
Required. An interface whose properties, methods, and events will be implemented by corresponding members in the class or structure.

interfacemember
Required. The member of an interface that is being implemented.

Remarks

An interface is a collection of prototypes representing the members (properties, methods and events) the interface encapsulates. Interfaces contain only the declarations for members; classes and structures implement these members.

When you implement an interface, you must implement all the members declared in the interface. Omitting any member is considered to be a syntax error. Classes can use **Private** implementations of properties and methods, but these members are only accessible by casting an instance of the implementing class into a variable declared to be of the type of the interface.

Example

The following example shows how to use the **Implements** statement to implement the properties, methods and events of an interface.

This example defines an interface named **ICustomerInfo** with a property, method, and an event. The Class **CustomerInfo** implements all the members defined in the interface.

```
Interface IcustomerInfo
' Declare an interface.
   Property CustomerName() As String
   Sub UpdateCustomerStatus()
   Event UpdateComplete()
End Interface

Public Class CustomerInfo
' The CustomerInfo class implements the IcustomerInfo interface.
   Implements ICustomerInfo
   Private CustomerNameValue As String ' Used to store the property value.
   Public Event UpdateComplete() Implements ICustomerInfo.UpdateComplete
   Public Property CustomerName() As String Implements _
      ICustomerInfo.CustomerName
      Get
         Return CustomerNameValue
      End Get
      Set(ByVal Value As String)
         ' The Value parameter is passed to the Set procedure
         ' when the contents of this property is modified.
         CustomerNameValue = Value ' Save the new value.
      End Set
   End Property
   Public Sub UpdateCustomerStatus() Implements _
      ICustomerInfo.UpdateCustomerStatus
      ' Add code here to update the status of this account.
```

```
        ' Raise an event to indicate that this method is done.
        RaiseEvent UpdateComplete()
    End Sub
End Class
```

The following two procedures show how you could use the implemented interface. To test the interface implementation, add these procedures to your project and call the `TestImplements` procedure.

```
Public Sub TestImplements()
'  This procedure tests the interface implementation by
'  creating an instance of the class that implements ICustomerInfo.
   Dim Cust As New CustomerInfo()
   ' Associate an event handler with the event
   ' that is raised by the Cust object.
   AddHandler Cust.UpdateComplete, AddressOf HandleUpdateComplete
   ' Set the CustomerName Property
   Cust.CustomerName = "Fred"
   ' Retrieve and display the CustomerName property.
   MsgBox("Customer name is: " & Cust.CustomerName)
   ' Call the update CustomerStatus method that
   ' raises the UpdateComplete event.
   Cust.UpdateCustomerStatus()
End Sub

Sub HandleUpdateComplete()
   ' This is the event handler for the UpdateComplete event.
   MsgBox("Update is complete.")
End Sub
```

See Also

Interface Statement | Interfaces in Visual Basic .NET | Polymorphism | How Visual Basic Provides Polymorphism

Imports Statement

Imports namespace names from referenced projects and assemblies. Also imports namespace names defined within the same project as the file in which the statement appears.

```
Imports [ aliasname = ] namespace
```

-or-

```
Imports [ aliasname = ] namespace.element
```

Parts

aliasname
Optional. The name by which *namespace* may also be known or referred to. When the **Imports** statement does not include *aliasname*, the elements defined within the specified *namespace* can be accessed within the file without qualification. When *aliasname* is specified, it must be used as a qualifier for the names contained in the namespace. Aliases are useful when you need to use items with the same name that are declared in one or more namespaces.

namespace
Required. The name of the namespace being imported. The namespace can be any number of nesting levels deep.

element
Optional. The name of an element declared in the namespace. The element can be an enumeration, structure, class, or module.

Remarks

Each file can contain any number of **Imports** statements. **Imports** statements must be placed before any declarations, including **Module** or **Class** statements, and before any references to identifiers.

The scope of the elements made available by an **Imports** statement depends on how specific you are when using the **Imports** statement. For example, if only a namespace is specified, all uniquely named members of that namespace, and members of modules within that namespace, are available without qualification. If both a namespace and the name of an element of that namespace are specified, only the members of that element are available without qualification.

It is not permitted to define a member at module level with the same name as an import alias.

Example

The following example imports the **Microsoft.VisualBasic.Strings** class and assigns an alias, `Str`, that can be used to access the **Left** method.

```
Imports Str = Microsoft.VisualBasic.Strings
' Place Imports statements at the top of your program
Class MyClass1
   Sub ShowHello()
      MsgBox(Str.Left("Hello World", 5)) ' Displays the word "Hello"
   End Sub
End Class
```

See Also

Introduction to the .NET Framework Class Library in Visual Studio | Namespace Statement | Namespaces | Visual Basic .NET and the .NET Framework

Inherits Statement

Causes the current class or interface to inherit the attributes, fields, properties, methods, and events from another class or interface.

```
Inherits typename
```

Parts

typename
Required. The name of a class or interface being inherited by the class or interface in which the **Inherits** statement is used.

Remarks

If used, the **Inherits** statement must be the first non-blank, non-comment line in a class definition.

An interface may inherit from more than one interface, but a class may only inherit from one other class.

Example

This example uses the **Inherits** statement to show how a class named `ThisClass` can inherit the fields, properties, methods and events from another class named `AnotherClass`.

```
Public Class ThisClass
   Inherits AnotherClass
   ' Add code to override, overload, or extend members
   ' inherited from the base class.
   ...' Variable, property, method, and event declarations here.
End Class
```

See Also

MustInherit | NotInheritable | MyBase | MyClass | Inheritance

Interface Statement

Declares the name of an interface, as well as the properties, methods and events that comprise the interface.

```
[ <attrlist> ] [ Public | Private | Protected | Friend | Protected Friend ] _
[ Shadows ] Interface name
   [ Inherits interfacename[, interfacename ]]
   [ [ Default ] Property proname ]
   [ Function memberame ]
   [ Sub memberame ]
   [ Event memberame ]
End Interface
```

Parts

attrlist
Optional. List of attributes that apply to this interface. Multiple attributes are separated by commas.

Public
Optional. Entities declared with the **Public** modifier have public access. There are no restrictions on the use of public entities.

Private
Optional. Entities declared with the **Private** modifier have private access. A private entity is accessible only within its declaration context, including any nested entities.

Protected
Optional. Entities declared with the **Protected** keyword have protected access. They are accessible only from within their own class or from a derived class. Protected access can be specified only on members of classes. It is not a superset of friend access.

Friend
Optional. Entities declared with the **Friend** modifier have friend access. An entity with friend access is accessible only within the program that contains the entity declaration. Interfaces that do not specify an access modifier are declared as **Friend** by default.

Protected Friend
Optional. Entities declared with the **Protected Friend** modifiers have the union of protected and friend accessibility.

Shadows
Optional. Indicates that this interface shadows an identically named programming element in a base class. You can shadow any kind of declared element with any other kind. A shadowed element is unavailable from within the derived class that shadows it, unless the shadowing element is inaccessible, for example if it is **Private**.

name
Required. Name of the interface; follows standard variable naming conventions.

Inherits
Optional. Specifies that the interface being declared inherits the properties and methods from the interface specified by *interfacename*.

interfacename
Required if optional **Inherits** statement is present. Indicates the name of one or more interfaces being inherited.

Default
Optional. Indicates that the property with which the **Default** keyword is associated is the default property for the interface.

Property
Optional. Indicates a **Property** procedure that is a member of the interface.

Function
Optional. Indicates a **Function** procedure that is a member of the interface.

Sub
Optional. Indicates a **Sub** procedure that is a member of the interface.

Event
Optional. Indicates an **Event** that is a member of the interface.

memberame
Required if optional **Property**, **Event**, **Function**, or **Sub** statements exist. Indicates the name of the property, procedure, or event.

End Interface
Terminates an **Interface** block.

Each attribute in the *attrlist* part has the following syntax and parts:

```
attrname [({ attrargs | attrinit })]
```

attrlist Parts

attrname
Required. Name of the attribute. Must be a valid Visual Basic identifier.
attrargs
Optional. List of positional arguments for this attribute. Multiple arguments are separated by commas.
attrinit
Optional. List of field or property initializers for this attribute. Multiple initializers are separated by commas.

Remarks

The methods and properties of an interface are implemented in a class. When methods are implemented, the return data type and the argument data types must exactly match those of the method described in the **Interface** block. The method name, however, need not be the same as that described in the **Interface** block. Methods being implemented in a class cannot be designated as **Shared,** nor can they be **Private**, except in a non-inheritable class.

The **Interface** statement can appear in the declaration section of any module. Interfaces are implicitly **Friend** by default but can be explicitly declared **Public**, **Friend**, **Protected**, **Protected Friend** or **Private**.

An interface can have, at most, one default property. This implies that an interface can either inherit only one interface containing a default property, or its definition block can contain one property that is marked as default.

An interface cannot inherit from another interface whose access level is more restrictive than its own. For example, a public interface cannot inherit a friend interface.

Example

This example uses the **Interface** statement to define a `MyInterface` interface, which must be implemented with a **Property** statement and a **Function** statement.

```
Public Interface MyInterface
   Property MyProp(ByVal MyStr As String)
   Function MyFunc(ByVal MyInt As Integer) As Integer
End Interface
```

See Also

Class Statement | Function Statement | Inherits Statement | Implements Statement | Interfaces in Visual Basic .NET | Sub Statement

Mid Statement

Replaces a specified number of characters in a **String** variable with characters from another string.

```
Mid( _
   ByRef Target As String, _
   ByVal Start As Integer, _
   Optional ByVal Length As Integer _
) = StringExpression
```

Parts

Target
Required. Name of the **String** variable to modify.
Start
Required. **Integer** expression. Character position in *Target* where the replacement of text begins. *Start* uses a one-based index.
Length
Optional. **Integer** expression. Number of characters to replace. If omitted, all of *String* is used.
StringExpression
Required. **String** expression that replaces part of *Target*.

Remarks

The number of characters replaced is always less than or equal to the number of characters in *Target*.

> **Note The MidB statement of earlier versions of Visual Basic replaces a substring in bytes, rather than characters. It is used primarily for converting strings in double-byte character set (DBCS) applications. All Visual Basic .NET strings are in Unicode, and MidB is no longer supported**.

Example

This example uses the **Mid** statement to replace a specified number of characters in a string variable with characters from another string.

```
Dim MyString As String
MyString = "The dog jumps"   ' Initializes string.
Mid(MyString, 5, 3) = "fox"   ' MyString = "The fox jumps".
Mid(MyString, 5) = "cow"   ' MyString = "The cow jumps".
Mid(MyString, 5) = "cow jumped over"   ' MyString = "The cow jumpe".
Mid(MyString, 5, 3) = "duck"   ' MyString = "The duc jumpe".
```

Requirements

Namespace: Microsoft.VisualBasic

Module: Strings

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Strings | String Manipulation | Mid Function

Module Statement

Declares a module block.

```
[ <attrilist> ] [ Public | Friend ] Module name
   [ statements ]
End Module
```

Parts

attrlist
Optional. List of attributes that apply to this module. Multiple attributes are separated by commas.

Public
Optional. Modules declared with the **Public** keyword have public access. There are no restrictions on the accessibility of public modules.

Friend
Optional. Modules declared with the **Friend** keyword have friend access. They are accessible from within their declaration context and from anywhere else in the same program. Modules that do not specify an access modifier are declared as **Friend** by default.

name
Required. Name of the module. Must be a valid Visual Basic identifier.

statements
Optional. Statements that comprise the variables, properties, events, and methods of the module.

End Module
Terminates a **Module** block.

Each attribute in the *attrlist* part has the following syntax and parts:

```
attrname [({ attrargs | attrinit })]
```

attrlist Parts

attrname
Required. Name of the attribute. Must be a valid Visual Basic identifier.

attrargs
Optional. List of positional arguments for this attribute. Multiple arguments are separated by commas.

attrinit
Optional. List of field or property initializers for this attribute. Multiple initializers are separated by commas.

Remarks

Modules are a reference type similar to classes, but with some important distinctions. The members of a module are implicitly **Shared** and scoped to the declaration space of the standard module's containing namespace, rather than just to the module itself. Unlike classes, modules can never be instantiated, do not support inheritance, and cannot implement interfaces. A module can only be declared in a namespace and cannot be nested in another type.

You can have multiple modules in a project, but members with the same name defined in two or more modules must be qualified with their module name when accessed outside of their module.

Example

```
Public Module Module1
   ' Add classes, properties, methods, fields, and events for this module.
End Module
```

See Also

Class Statement | Namespace Statement | Property Statement

Namespace Statement

Declares the name of a namespace.

```
Namespace { name | name.name }
   [ componenttypes ]
End Namespace
```

Parts

name
Required. A unique name that identifies the namespace.

componenttypes
Optional. Elements that make up the namespace. These include (but are not limited to) enumerations, structures, interfaces, classes, modules, delegates, and other namespaces.

End Namespace
Terminates a **Namespace** block.

Remarks

Namespaces are used as an organizational system - a way of presenting program components that are exposed to other programs and applications.

Namespaces are always **Public;** therefore the declaration of a namespace cannot include access modifiers. However, the components within the namespace may have **Public** or **Friend** access. If it is not stated, the default access type is **Friend**.

Example

The following example declares two namespaces.

```
Namespace N1   ' Declares a namespace named N1.
   Namespace N2   ' Declares a namespace named N2 within N1.
      Class A   ' Declares a class within N1.N2.
      ' Add class statements here.
      End Class
   End Namespace
End Namespace
```

The next example declares multiple nested namespaces on a single line, and is equivalent to the previous example.

```
Namespace N1.N2 ' Declares two namespaces; N1 and N2.
   Class A   ' Declares a class within N1.N2.
      ' Add class statements here.
   End Class
End Namespace
```

See Also

Imports Statement | Namespaces

On Error Statement

Enables an error-handling routine and specifies the location of the routine within a procedure; can also be used to disable an error-handling routine. Without an **On Error** statement, any run-time error that occurs is fatal: an error message is displayed, and execution stops.

```
On Error { GoTo [ line | 0 | -1 ] | Resume Next }
```

Parts

GoTo *line*
Enables the error-handling routine that starts at the line specified in the required *line* argument. The *line* argument is any line label or line number. If a run-time error occurs, control branches to the specified line, making the error handler active. The specified line must be in the same procedure as the **On Error** statement, or a compile-time error will occur.

GoTo 0
Disables enabled error handler in the current procedure and resets it to **Nothing**.

GoTo -1
Disables enabled exception in the current procedure and resets it to **Nothing**.

Resume Next
Specifies that when a run-time error occurs, control goes to the statement immediately following the statement where the error occurred and execution continues from that point. Use this form rather than **On Error GoTo** when accessing objects.

Remarks

An "enabled" error handler is one that is turned on by an **On Error** statement; an "active" error handler is an enabled handler that is in the process of handling an error. If an error occurs while an error handler is active (between the occurrence of the error and a **Resume**, **Exit Sub**, **Exit Function**, or **Exit Property** statement), the current procedure's error handler can't handle the error. Control returns to the calling procedure. If the calling procedure has an enabled error handler, it is activated to handle the error. If the calling procedure's error handler is also active, control passes back through previous calling procedures until an enabled, but inactive, error handler is found. If no such error handler is found, the error is fatal at the point at which it actually occurred.

Each time the error handler passes control back to a calling procedure, that procedure becomes the current procedure. Once an error is handled by an error handler in any procedure, execution resumes in the current procedure at the point designated by the **Resume** statement.

> **Note** An error-handling routine is not a **Sub** procedure or a **Function** procedure. It is a section of code marked by a line label or a line number.

Error-handling routines rely on the value in the **Number** property of the **Err** object to determine the cause of the error. The routine should test or save relevant property values in the **Err** object before any other error can occur or before a procedure that might cause an error is called. The property values in the **Err** object reflect only the most recent error. The error message associated with **Err.Number** is contained in **Err.Description**.

An error that is raised with the **Err.Raise** method will set the **Exception** property to a newly created instance of the **Exception** class. In order to support the raising of exceptions of derived exception types, a **Throw** statement is supported in the language. This takes a single parameter that is the exception instance to be thrown. The following example shows how these features can be used with the existing exception handling support:

```
    On Error GoTo Handler
    Throw New DivideByZeroException()
Handler:
    If (TypeOf Err.GetException() Is DivideByZeroException) Then
    ' Code for handling the error is entered here.
    End If
```

Note that the **On Error GoTo** statement traps all errors regardless of the exception class.

On Error Resume Next causes execution to continue with the statement immediately following the statement that caused the run-time error, or with the statement immediately following the most recent call out of the procedure containing the **On Error Resume Next** statement. This statement allows execution to continue despite a run-time error. You can place the error-handling routine where the error would occur rather than transferring control to another location within the procedure. An **On Error**

Resume Next statement becomes inactive when another procedure is called, so you should execute an **On Error Resume Next** statement in each called routine if you want inline error handling within that routine.

> **Note** The **On Error Resume Next** construct may be preferable to **On Error GoTo** when handling errors generated during access to other objects. Checking **Err** after each interaction with an object removes ambiguity about which object was accessed by the code. You can be sure which object placed the error code in **Err.Number**, as well as which object originally generated the error (the object specified in **Err.Source**).

On Error GoTo 0 disables error handling in the current procedure. It doesn't specify line 0 as the start of the error-handling code, even if the procedure contains a line numbered 0. Without an **On Error GoTo 0** statement, an error handler is automatically disabled when a procedure is exited.

On Error GoTo -1 disables the exception in the current procedure. It doesn't specify line -1 as the start of the error-handling code, even if the procedure contains a line numbered -1. Without an **On Error GoTo -1** statement, an exception is automatically disabled when a procedure is exited.

To prevent error-handling code from running when no error has occurred, place an **Exit Sub**, **Exit Function**, or **Exit Property** statement immediately before the error-handling routine, as in the following fragment:

```
Public Sub InitializeMatrix(Var1, Var2, Var3, Var4)
   On Error GoTo ErrorHandler
   ' Insert code that might generate an error here
   Exit Sub
ErrorHandler:
   ' Insert code to handle the error here
   Resume Next
End Sub
```

Here, the error-handling code follows the **Exit Sub** statement and precedes the **End Sub** statement to separate it from the procedure flow. You can place error-handling code anywhere in a procedure.

Untrapped errors in objects are returned to the controlling application when the object is running as an executable file. Within the development environment, untrapped errors are returned to the controlling application only if the proper options are set. See your host application's documentation for a description of which options should be set during debugging, how to set them, and whether the host can create classes.

If you create an object that accesses other objects, you should try to handle any unhandled errors they pass back. If you cannot, map the error codes in **Err.Number** to one of your own errors and then pass them back to the caller of your object. You should specify your error by adding your error code to the **VbObjectError** constant. For example, if your error code is 1052, assign it as follows:

```
Err.Number = VbObjectError + 1052
```

> **Caution** System errors during calls to Windows dynamic-link libraries (DLL) do not raise exceptions and cannot be trapped with Visual Basic error trapping. When calling DLL functions, you should check each return value for success or failure (according to the API specifications), and in the event of a failure, check the value in the **Err** object's **LastDLLError** property.

Example

This example first uses the **On Error GoTo** statement to specify the location of an error-handling routine within a procedure. In the example, an attempt to divide by zero generates error number 6. The error is handled in the error-handling routine, and control is then returned to the statement that caused the error. The **On Error GoTo 0** statement turns off error trapping. Then the **On Error Resume Next** statement is used to defer error trapping so that the context for the error generated by the next statement can be known for certain. Note that **Err.Clear** is used to clear the **Err** object's properties after the error is handled.

```
Public Sub OnErrorDemo()
   On Error GoTo ErrorHandler   ' Enable error-handling routine.
   Dim x As Integer = 32
   Dim y As Integer = 0
   Dim z As Integer
   z = x / y   ' Creates a divide by zero error
   On Error GoTo 0   ' Turn off error trapping.
   On Error Resume Next   ' Defer error trapping.
```

```
    z = x / y   ' Creates a divide by zero error again
    If Err.Number = 6 Then
       ' Tell user what happened. Then clear the Err object.
       Dim Msg As String
       Msg = "There was an error attempting to divide by zero!"
       MsgBox(Msg, , "Divide by zero error")
       Err.Clear() ' Clear Err object fields.
    End If
 Exit Sub      ' Exit to avoid handler.
 ErrorHandler:  ' Error-handling routine.
    Select Case Err.Number   ' Evaluate error number.
       Case 6   ' Divide by zero error
          MsgBox("You attempted to divide by zero!")
          ' Insert code to handle this error
       Case Else
          ' Insert code to handle other situations here...
    End Select
    Resume Next  ' Resume execution at same line
                 ' that caused the error.
 End Sub
```

Requirements

Namespace: Microsoft.VisualBasic

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

End Statement | Err Object | Exit Statement | LastDLLError Property | Resume Statement | Visual Basic Run-Time Errors

Option Compare Statement

Used at file level to declare the default comparison method to use when string data is compared.

```
Option Compare { Binary | Text }
```

Parts

Binary
Optional. Results in string comparisons based on a sort order derived from the internal binary representations of the characters.

Text
Optional. Results in string comparisons based on a case-insensitive text sort order determined by your system's locale.

Remarks

If used, the **Option Compare** statement must appear in a file before any other source statements.

The **Option Compare** statement specifies the string comparison method (**Binary** or **Text**) for a class, module or structure. If an **Option Compare** statement is not included, the default text comparison method is **Binary**.

In Microsoft Windows, sort order is determined by the code page. In the following example, characters are sorted using **Option Compare Binary**, which produces a typical binary sort order:

```
A < B < E < Z < a < b < e < z < À < Ê < Ø < à < ê < ø
```

When the same characters are sorted using **Option Compare Text**, the following text sort order is produced:

```
(A=a) < (À = à) < (B=b) < (E=e) < (Ê = ê) < (Z=z) < (Ø = ø)
```

Example

This example uses the **Option Compare** statement to set the default string comparison method. The **Option Compare** statement is used at the module level only.

```
' Set the string comparison method to Binary.
Option Compare Binary   ' That is, "AAA" is less than "aaa".
' Set the string comparison method to Text.
Option Compare Text   ' That is, "AAA" is equal to "aaa".
```

See Also

Comparison Operators | InStr Function | InStrRev Function | Option Strict Statement | Option Explicit Statement | Replace Function | Split Function | StrComp Function | /optionexplicit | /optionstrict | /optioncompare:binary | /optioncompare:text

Option Explicit Statement

Used at file level to force explicit declaration of all variables in that file.

```
Option Explicit { On | Off }
```

Parts

On
Optional. Enables **Option Explicit** checking. If **On** or **Off** is not specified after the **Option Explicit** statement, the default is **On**.

Off
Optional. Disables **Option Explicit** checking.

Remarks

If used, the **Option Explicit** statement must appear in a file before any other source statements.

When **Option Explicit** appears in a file, you must explicitly declare all variables using the **Dim**, **Private**, **Public**, or **ReDim** statements. If you attempt to use an undeclared variable name, an error occurs at compile time.

If you don't use the **Option Explicit** statement, all undeclared variables are of **Object** type.

> **Note** Use **Option Explicit** to avoid incorrectly typing the name of an existing variable or to avoid confusion in code where the scope of the variable is not clear. If you do not specify **Option Explicit** in your code, the compiler default is **Option Explicit On**.

Example

This example uses the **Option Explicit** statement to force explicit declaration of all variables. Attempting to use an undeclared variable causes an error at compile time. The **Option Explicit** statement is used at the module level only.

```
Option Explicit On   ' Force explicit variable declaration.
Dim MyVar As Integer   ' Declare variable.
MyInt = 10   ' Undeclared variable generates error.
MyVar = 10   ' Declared variable does not generate error.
```

See Also

Const Statement | Dim Statement | Function Statement | Option Compare Statement | Private | Public | ReDim Statement | Sub Statement | /optionexplicit | /optionstrict | /optioncompare:binary | /optioncompare:text

Option Strict Statement

Restricts implicit data type conversions to only *widening* conversions. This explicitly disallows any data type conversions in which data loss would occur and any conversion between numeric types and strings.

```
Option Strict { On | Off }
```

Parts

On
Optional. Enables **Option Strict** checking. If **On** or **Off** is not specified after the **Option Strict** statement, the default is **On**.

Off
Optional. Disables **Option Strict** checking.

Remarks

When used, the **Option Strict** statement must appear before any other code.

Visual Basic .NET generally allows implicit conversions of any data type to any other data type. Data loss can occur when the value of one data type is converted to a data type with less precision or smaller capacity, however, a run-time error message will occur if data will be lost in such a conversion. **Option Strict** ensures compile-time notification of these types of conversions so they may be avoided.

In addition to disallowing narrowing conversions, **Option Strict** generates an error for *late binding*. An object is late bound when it is assigned to a variable that is declared to be of type Object.

> **Note** The compiler default is **Option Strict Off** if do not specify **Option Strict** in your code.

Example

This example demonstrates how the **Option Strict** statement disallows late binding and conversions where data would be lost.

```
Option Strict On   ' Force explicit variable declaration.

Dim MyVar As Integer   ' Declare variables.
Dim Obj As Object

MyVar = 1000   ' Declared variable does not generate error.

'Attempting to convert to an Integer generates an error.
MyVar = 1234567890.987654321
'
'Call Obj.Method1()   ' Late-bound call generates an error
```

See Also

Widening and Narrowing Conversions | Option Explicit Statement | Option Compare Statement | /optionexplicit | /optionstrict | /optioncompare:binary | /optioncompare:text

Property Statement

Declares the name of a property, and the property procedures used to store and retrieve the value of the property.

```
[ <attrlist> ] [ Default ] [ Public | Private | Protected | Friend | Protected Friend ] _
[[ ReadOnly | WriteOnly ] [ Overloads | Overrides ] _
[ Overridable | NotOverridable ] | MustOverride | Shadows | Shared ] _
Property varname([ ByVal parameter list ]) [ As typename ] [ Implements interfacemember ]
   [ <attrlist> ] Get
      [ block ]
   End Get
   [ <attrlist> ] Set(ByVal value As typename )
      [ block ]
   End Set
End Property
```

Parts

attrlist
Optional. List of attributes that apply to this property. Multiple attributes are separated by commas.

Default
Optional. Declares a default property. Default properties must accept parameters and can be set and retrieved without specifying the property name.

Overloads
Optional. Indicates that this property overloads one or more properties defined with the same name in a base class. The argument list in this declaration must be different from the argument list of every overloaded property. The lists must differ in the number of arguments, their data types, or both. This allows the compiler to distinguish which version to use.

You do not have to use the **Overloads** keyword when you are defining multiple overloaded properties in the same class. However, if you use **Overloads** in one of the declarations, you must use it in all of them.

You cannot specify both **Overloads** and **Shadows** in the same property declaration.

Overrides
Optional. Indicates that this property overrides an identically named property in a base class. The number and data types of the arguments, and the data type of the return value, must exactly match those of the base class property.

Overridable
Optional. Indicates that this property can be overridden by an identically named property in a derived class.

NotOverridable
Optional. Indicates that this property cannot be overridden in a derived class. Properties are **NotOverridable** by default but you cannot specify the **NotOverridable** modifier with properties or methods that do not override another member.

MustOverride
Optional. Indicates that this property is not implemented in this class, and must be implemented in a derived class for that class to be creatable.

Shadows
Optional. Indicates that this property shadows an identically named programming element, or set of overloaded elements, in a base class. You can shadow any kind of declared element with any other kind. If you shadow a property with another property, the arguments and the return type do not have to match those in the base class property. A shadowed element is unavailable from within the derived class that shadows it, unless the shadowing element is inaccessible, for example if it is **Private**.

> **Note** You cannot specify both **Overloads** and **Shadows** in the same property declaration.

Shared
Optional. Indicates that this is a shared property. This means it is not associated with a specific instance of a class or structure. You can call a shared property by qualifying it either with the class or structure name, or with the variable name of a specific instance of the class or structure.

Public
Optional. Entities declared with the **Public** modifier have public access. There are no restrictions on the use of public entities. Properties that do not specify an access modifier are **Public** by default.

Private
Optional. Entities declared with the **Private** modifier have private access. A private entity is accessible only within its declaration context, including any nested entities.

Protected
Optional. Entities declared with the **Protected** keyword have protected access. They are accessible only from within their own class or from a derived class. Protected access can be specified only on members of classes. It is not a superset of friend access.
Friend
Optional. Entities declared with the **Friend** modifier have friend access. An entity with friend access is accessible only within the program that contains the entity declaration.
Protected Friend
Optional. Entities declared with the **Protected Friend** modifiers have the union of protected and friend accessibility.
ReadOnly
Optional. Indicates that a property's value can be retrieved, but it cannot be modified. **ReadOnly** properties contain **Get** blocks but lack **Set** blocks.
WriteOnly
Optional. Indicates that a property can be the target of assignment but its value cannot be retrieved. **WriteOnly** properties contain **Set** blocks but lack **Get** blocks.
varname
Required. A unique name that identifies the property.
parameter list
Optional. This identifies the signature of the property. Parameters for properties must be passed **ByVal**.
typename
Optional. If no data type is specified, the default type is **Object**. If the property is implementing a property in an interface, *typename* must match the type declared in the interface.
Implements
Optional. Indicates that this property implements a property of an interface.
interfacemember
Optional. When a property is part of a class that implements an interface, this is the name of the property being implemented.
Get
Starts a **Get** property procedure used to return the value of a property. **Get** blocks are required unless the property is marked **WriteOnly**.
End Get
Terminates a **Get** property procedure.
Set
Starts a **Set** property procedure used to set the value of a property. **Set** blocks are required unless the property is marked **ReadOnly**. The new value of a property is passed to the **Set** property procedure in a parameter named *value* when the value of the property changes
End Set
Terminates a **Set** property procedure.
End Property
Terminates a **Property** definition.

Each attribute in the *attrlist* part has the following syntax and parts:

```
attrname [({ attrargs | attrinit })]
```

***attrlist* Parts**

attrname
Required. Name of the attribute. Must be a valid Visual Basic identifier.
attrargs
Optional. List of positional arguments for this attribute. Multiple arguments are separated by commas.
attrinit
Optional. List of field or property initializers for this attribute. Multiple initializers are separated by commas.

Remarks

Visual Basic .NET passes a parameter to the **Set** block during property assignments. If you do not supply a parameter for **Set**, the integrated development environment (IDE) supplies a parameter named **Value**. The parameter contains the contents of the item that was assigned to the property when the Set block was called. The contents of the parameter are usually stored in a private local variable and returned whenever the **Get** block is called.

The declaration of the property determines what the user can do with that property:

- If the **ReadOnly** modifier is used, the property is known as a "Read-only property" and must only have a **Get...End Get**

block. Therefore, the user is only able to retrieve the value of the property. An error will be raised if the user attempts to assign a value to that property.

- If the **WriteOnly** modifier is used, the property is known as a "Write-only property" and must only have a **Set...End Set** block. This allows the user to store a value to the property. An error will be raised if the user attempts to refer to the property, except in the assignment of a value to that property.
- If neither the **ReadOnly** nor the **WriteOnly** modifier is used, the property must have both a **Set...End Set** and a **Get...End Get** block. The property is said to be a read-write property.

Example

The following example declares a property in a class.

```
Class Class1
   ' Define a local variable to store the property value.
   Private PropertyValue As String
   ' Define the property.
   Public Property Prop1() As String
      Get
         ' The Get property procedure is called when the value
         ' of a property is retrieved.
         Return PropertyValue
      End Get
      Set(ByVal Value As String)
         ' The Set property procedure is called when the value
         ' of a property is modified.
         ' The value to be assigned is passed in the argument to Set.
         PropertyValue = Value
      End Set
   End Property
End Class
```

See Also

Adding Fields and Properties to a Class | Get Statement | Set Statement | Default Properties

Q-Z Statements

The following table contains a listing of the Visual Basic .NET language statements.

RaiseEvent	Randomize	ReDim	REM
RemoveHandler	Resume	Return	Select Case
Set	Stop	Structure	Sub
SyncLock	Throw	Try...Catch...Finally	While...End While
With...End With			

See Also

A-E Statements | F-P Statements | Visual Basic Language and Run-Time Reference

RaiseEvent Statement

Triggers an event declared at module level within a class, form, or document.

```
RaiseEvent eventname[( argumentlist )]
```

The required *eventname* is the name of an event declared within the module. It follows Visual Basic variable naming conventions.

Parts

eventname
Required. Name of the event to trigger.
argumentlist
Optional. Comma-delimited list of variables, arrays, or expressions. The *argumentlist* argument must be enclosed by parentheses. If there are no arguments, the parentheses must be omitted.

Remarks

If the event has not been declared within the module in which it is raised, an error occurs. The following fragment illustrates an event declaration and a procedure in which the event is raised.

```
' Declare an event at module level.
Event LogonCompleted(UserName As String)

Sub Logon(ByVal UserName As String)
   ' Raise the event.
   RaiseEvent LogonCompleted(UserName)
End Sub
```

You cannot use **RaiseEvent** to raise events that are not explicitly declared in the module. For example, if a form has a Click event, you cannot raise its Click event using **RaiseEvent**. If you declare a Click event in the form module, it shadows the form's own Click event. You can still invoke the form's Click event using normal syntax for calling the event, but not using the **RaiseEvent** statement.

Events are raised in the order that the connections are established. Since events can have **ByRef** parameters, a process that connects late may receive parameters that have been changed by an earlier event handler.

> **Note** Non-shared events should not be raised within the constructor of the class in which they are declared. Although such events do not cause runtime errors, they may fail to be caught by associated event handlers. Use the **Shared** keyword to create a shared event if you need to raise an event from a constructor.

Example

The following example uses events to count off seconds during a demonstration of the fastest 100-meter race. The code illustrates all of the event-related methods, properties, and statements, including the **RaiseEvent** statement.

The class that raises an event is the event source, and the methods that process the event are the event handlers. An event source can have multiple handlers for the events it generates. When the class raises the event, that event is raised on every class that has elected to handle events for that instance of the object.

The example also uses a form (`Form1`) with a button (`Command1`), a label (`Label1`), and two text boxes (`Text1` and `Text2`). When you click the button, the first text box displays "From Now" and the second starts to count seconds. When the full time (9.84 seconds) has elapsed, the first text box displays "Until Now" and the second displays "9.84"

The code for `Form1` specifies the initial and terminal states of the form. It also contains the code executed when events are raised.

To use this example, open a new Windows Forms project, add a button named `Button1`, a label named `Label1` and two text boxes, named `TextBox1` and `TextBox2`, to the main form, named `form1`. Then right click the form and click **View Code** to open the code editor.

To simplify access to the Timer property, add an **Imports** statement as the first line of code above the `Class Form1` statement.

```
Imports Microsoft.VisualBasic.DateAndTime
```

Add a **WithEvents** variable to the declarations section of the `Form1` class.

```
Private WithEvents mText As TimerState
```

Add the following code to the code for `Form1`. Replace any duplicate procedures that may exist, such as `Form_Load`, or `Button_Click`.

```
Private Sub Form1_Load(ByVal sender As Object, _
                       ByVal e As System.EventArgs) _
                       Handles MyBase.Load
   Button1.Text = "Click to Start Timer"
   TextBox1.Text = ""
   TextBox2.Text = ""
   Label1.Text = "The fastest 100 meters ever run took this long:"
   mText = New TimerState()
End Sub
Private Sub Button1_Click(ByVal sender As System.Object, _
                          ByVal e As System.EventArgs) _
                          Handles Button1.Click
   TextBox1.Text = "From Now"
   TextBox1.Refresh()
   TextBox2.Text = "0"
   TextBox2.Refresh()
   mText.TimerTask(9.84)
End Sub

Private Sub mText_ChangeText() Handles mText.ChangeText
   TextBox1.Text = "Until Now"
   TextBox2.Text = "9.84"
End Sub

Private Sub mText_UpdateTime(ByVal Jump As Double) _
      Handles mText.UpdateTime
   TextBox2.Text = Format(Jump, "##0.00")
   Application.DoEvents()
End Sub

Class TimerState
   Public Event UpdateTime(ByVal Jump As Double)
   Public Event ChangeText()
   Public Sub TimerTask(ByVal Duration As Double)
      Dim Start As Double
      Dim Second As Double
      Dim SoFar As Double
      Start = Timer
      SoFar = Start
      Do While Timer < Start + Duration
         If Timer - SoFar >= 0.1 Then
            SoFar = SoFar + 0.1
            RaiseEvent UpdateTime(Timer - Start)
         End If
      Loop
      RaiseEvent ChangeText()
   End Sub
End Class
```

Press F5 to run this example, and click the button labeled **Click to start timer**. The text box will count up to 9.84 seconds.

See Also

Events and Delegates | Adding Events to a Class | Event Statement

Randomize Statement

Initializes the random-number generator.

```
Public Shared Sub Randomize [ number ]
```

Part

number
Optional. An **Object** or any valid numeric expression.

Remarks

Randomize uses *number* to initialize the **Rnd** function's random-number generator, giving it a new seed value. If you omit *number*, the value returned by the system timer is used as the new seed value.

If **Randomize** is not used, the **Rnd** function (with no arguments) uses the same number as a seed the first time it is called, and thereafter uses the last-generated number as a seed value.

> **Note** To repeat sequences of random numbers, call **Rnd** with a negative argument immediately before using **Randomize** with a numeric argument. Using **Randomize** with the same value for *number* does not repeat the previous sequence.
>
> **Security Note** Because the **Random** statement and the **Rnd** function start with a seed value and generate numbers that fall within a finite range, the results may be predictable by someone who knows the algorithm used to generate them. Consequently, the **Random** statement and the **Rnd** function should not be used to generate random numbers for use in cryptography.

Example

This example uses the **Randomize** statement to initialize the random-number generator. Because the number argument has been omitted, **Randomize** uses the return value from the **Timer** function as the new seed value.

```
Dim MyValue As Integer
Randomize   ' Initialize random-number generator.
MyValue = CInt(Int((6 * Rnd()) + 1)) ' Generate random value between 1 and 6.
```

Requirements

Namespace: Microsoft.VisualBasic

Module: VBMath

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Rnd Function

ReDim Statement

Used at procedure level to reallocate storage space for an array variable.

```
ReDim [ Preserve ] name(boundlist)
```

Parts

Preserve
Optional. Keyword used to preserve the data in the existing array when you change the size of only the last dimension.

name
Required. Name of the variable. Must be a valid Visual Basic identifier. You can redimension as many variables as you like in the same statement, specifying the *name* and *boundlist* parts for each one. Multiple variables are separated by commas.

boundlist
Required. List of non-negative integers representing the upper bounds of the dimensions of the redefined array. Multiple upper bounds are separated by commas. The number of dimensions in *boundlist* must match the original rank of the array.

Each value in *boundlist* specifies the upper bound of a dimension, not the length. The lower bound is always zero, so the subscript for each dimension can vary from zero through the upper bound.

It is possible to use -1 to declare the upper bound of an array dimension. This signifies that the array is empty but not **Nothing**, a distinction required by certain common language runtime functions. However, Visual Basic code cannot successfully access such an array. If you attempt to do so, an **IndexOutOfRangeException** error occurs during execution.

Remarks

The **ReDim** statement can appear only at procedure level. This means you can redefine arrays inside a procedure but not at class or module level.

The **ReDim** statement is used to change the size of one or more dimensions of an array that has already been formally declared. **ReDim** cannot change the rank (the number of dimensions) of the array.

The **ReDim** statement cannot change the data type of an array variable or provide new initialization values for the array elements.

ReDim releases the existing array and creates a new array with the same rank. The elements of the new array are initialized to the default value for their data type unless you specify **Preserve**.

If you include the **Preserve** keyword, Visual Basic copies the elements from the existing array to the new array. When you use **Preserve**, you can resize only the last dimension of the array, and for every other dimension you must specify the same size it already has in the existing array.

For example, if your array has only one dimension, you can resize that dimension and still preserve the contents of the array, because it is the last and only dimension. However, if your array has two or more dimensions, you can change the size of only the last dimension if you use **Preserve**.

The following example increases the size of the last dimension of a dynamic array without losing any existing data in the array, and then decreases the size with partial data loss:

```
Dim IntArray(10, 10, 10) As Integer
' ...
ReDim Preserve IntArray(10, 10, 20)
' ...
ReDim Preserve IntArray(10, 10, 15)
```

The first **ReDim** creates a new array, copying all the elements from the existing array. It also adds 10 more columns to the end of every row in every layer. The elements in these new columns are initialized to the default value of the element type of the array.

The second **ReDim** creates another new array, copying all the elements that fit. However, five columns are lost from the end of every row in every layer. This is not a problem if you have finished using these columns. Reducing the size of a large array can free up memory that you no longer need.

You can use **ReDim** on a property that holds an array of values.

Example

This example uses the **ReDim** statement to allocate and reallocate storage space for array variables.

```
Dim I, MyArray() As Integer   ' Declare variable and array variable.
ReDim MyArray(5) ' Allocate 6 elements.
For I = 0 To UBound(MyArray)
   MyArray(I) = I   ' Initialize array.
Next I
```

The next statement resizes the array without saving the contents of the elements.

```
ReDim MyArray(10) ' Resize to 11 elements.
For I = 0 To UBound(MyArray)
   MyArray(I) = I   ' Initialize array.
Next I
```

The following statement resizes the array but saves the contents of the elements.

```
ReDim Preserve MyArray(15) ' Resize to 16 elements.
```

See Also

Const Statement | Dim Statement | Erase Statement

REM Statement

Used to include explanatory remarks in a program.

```
REM comment
```

-or-

```
' comment
```

Parts

comment
Optional. The text of any comment you want to include. A space is required between the **REM** keyword and *comment*.

Remarks

You can put a **REM** statement alone on a line, or you can put it on a line following another statement. The **REM** statement must be the last statement on the line. If it follows another statement, the **REM** must be separated from that statement by a space.

You can use a single quote (') instead of **REM**. This is true whether your comment follows another statement on the same line or sits alone on a line.

> **Note** You cannot continue a **REM** statement by using a line-continuation sequence (_). Once a comment begins, the compiler does not examine the characters for special meaning. For a multiple-line comment, use another **REM** statement or a comment symbol (') on each line.

Example

This example illustrates the **REM** statement, which is used to include explanatory remarks in a program. It also shows the alternative of using the single quote character (') instead of **REM**.

```
Dim MyStr1, MyStr2 as String
MyStr1 = "Hello" REM Comment after a statement using REM.
MyStr2 = "Goodbye"   ' Comment after a statement using the ' character.
REM This entire line is a comment.
' This entire line is also a comment.
```

See Also

Comments in Code

RemoveHandler Statement

Removes the association between an event and an event handler.

```
RemoveHandler event, AddressOf eventhandler
```

Parts

event
The name of the event being handled.
eventhandler
The name of the procedure currently handling the event.

Remarks

The **AddHandler** and **RemoveHandler** statements allow you to start and stop event handling for a specific event at any time during program execution.

Example

```
Sub TestEvents()
   Dim Obj As New Class1()
   ' Associate an event handler with an event.
   AddHandler Obj.Ev_Event, AddressOf EventHandler
   Obj.CauseSomeEvent() ' Ask the object to raise an event.
   RemoveHandler Obj.Ev_Event, AddressOf EventHandler ' Stop handling events.
   Obj.CauseSomeEvent() ' This event will not be handled.
End Sub

   Sub EventHandler()
      MsgBox("EventHandler caught event.") ' Handle the event.
   End Sub

Public Class Class1
   Public Event Ev_Event()  ' Declare an event.
   Sub CauseSomeEvent()
      RaiseEvent Ev_Event() ' Raise an event.
   End Sub
End Class
```

See Also

AddHandler Statement | Handles | Events and Event Handlers | AddHandler and RemoveHandler

Resume Statement

Resumes execution after an error-handling routine is finished.

```
Resume [ Next | line ]
```

Parts

Resume
Required. If the error occurred in the same procedure as the error handler, execution resumes with the statement that caused the error. If the error occurred in a called procedure, execution resumes at the statement that last called out of the procedure containing the error-handling routine.

Next
Optional. If the error occurred in the same procedure as the error handler, execution resumes with the statement immediately following the statement that caused the error. If the error occurred in a called procedure, execution resumes with the statement immediately following the statement that last called out of the procedure containing the error-handling routine (or **On Error Resume Next** statement).

line
Optional. Execution resumes at the line specified in the required *line* argument. The *line* argument is a line label or line number and must be in the same procedure as the error handler.

Remarks

If you use a **Resume** statement anywhere except in an error-handling routine, an error will occur.

The **Resume** statement cannot be used in any procedure that contains a **Try...Catch...Finally** statement.

Example

This example uses the **Resume** statement to end error handling in a procedure, and then resume execution with the statement that caused the error. Error number 55 is generated to illustrate using the **Resume** statement.

```
Sub ResumeStatementDemo()
   On Error GoTo ErrorHandler   ' Enable error-handling routine.
   Dim x As Integer = 32
   Dim y As Integer = 0
   Dim z As Integer
   z = x / y   ' Creates a divide by zero error
   Exit Sub   ' Exit Sub to avoid error handler.
ErrorHandler:   ' Error-handling routine.
   Select Case Err.Number   ' Evaluate error number.
      Case 6   ' "Divide by zero" error.
         y = 1 ' Sets the value of y to 1 and tries the calculation again.
      Case Else
         ' Handle other situations here....
   End Select
   Resume   ' Resume execution at same line
   ' that caused the error.
End Sub
```

Requirements

Namespace: Microsoft.VisualBasic

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Try...Catch...Finally Statements | Error Statement | On Error Statement

Return Statement

Returns control to the code that called a **Sub**, **Function**, or **Property** procedure.

```
Return
```

-or-

```
Return expr
```

Part

expr
Required in a **Function** procedure or a **Property** procedure that retrieves the property's value. An expression that represents the value to be returned to the calling code.

Remarks

For a **Sub** procedure or a **Property** procedure that sets the property's value, the **Return** statement is equivalent to an **Exit Sub** statement, and *expr* must not be supplied.

For a **Function** procedure or a **Property** procedure that retrieves the property's value, *expr* must be present and must evaluate to a data type that is convertible to the return type of the function. In this form, **Return** is equivalent to assigning the expression to the function name as the return value and then executing an **Exit Function** statement.

Example

This example uses the **Return** statement several times to return to the calling code when the procedure does not need to do anything else.

```
Public Function GetAgePhrase(Age As Integer) As String
   If Age > 60 Then Return "Senior"
   If Age > 40 Then Return "Middle-aged"
   If Age > 20 Then Return "Adult"
   If Age > 12 Then Return "Teen-aged"
   If Age > 4 Then Return "School-aged"
   If Age > 1 Then Return "Toddler"
   Return "Infant"
End Function
```

See Also

Function Statement | Sub Statement

Select...Case Statements

Executes one of several groups of statements, depending on the value of an expression.

```
Select [ Case ] testexpression
   [ Case expressionlist
      [ statements ] ]
   [ Case Else
      [ elsestatements ] ]
End Select
```

Parts

testexpression
Required. Expression. Must evaluate to one of the elementary data types (**Boolean**, **Byte**, **Char**, **Date**, **Double**, **Decimal**, **Integer**, **Long**, **Object**, **Short**, **Single**, and **String**).

expressionlist
Required in a **Case** statement. List of expression clauses representing match values for *testexpression*. Multiple expression clauses are separated by commas. Each clause can take one of the following forms:

- *expression1* **To** *expression2*
- [**Is**] *comparisonoperator expression*
- *expression*

Use the **To** keyword to specify the boundaries of a range of match values for *testexpression*. The value of *expression1* must be less than or equal to the value of *expression2*.

Use the **Is** keyword with a comparison operator (**=**, **<>**, **<**, **<=**, **>**, or **>=**) to specify a restriction on the match values for *testexpression*. If the **Is** keyword is not supplied, it is automatically inserted before *comparisonoperator*.

The form specifying only *expression* is treated as a special case of the **Is** form where *comparisonoperator* is the equal sign (**=**). This form is evaluated as *testexpression* = *expression*.

The expressions in *expressionlist* can be of any data type, provided they are implicitly convertible to the type of *testexpression* and the appropriate *comparisonoperator* is valid for the two types it is being used with.

statements
Optional. One or more statements following **Case** that are executed if *testexpression* matches any clause in *expressionlist*.

elsestatements
Optional. One or more statements following **Case Else** that are executed if *testexpression* does not match any clause in the *expressionlist* of any of the **Case** statements.

End Select
Terminates **Select...Case** block.

Remarks

If *testexpression* matches any **Case** *expressionlist* clause, the statements following that **Case** statement are executed up to the next **Case** statement or the **End Select** statement. Control then passes to the statement following **End Select**. If *testexpression* matches an *expressionlist* clause in more than one **Case** clause, only the statements following the first match are executed.

The **Case Else** statement is used to introduce the *elsestatements* to be executed if no match is found between the *testexpression* and an *expressionlist* clause in any of the other **Case** statements. Although not required, it is a good idea to have a **Case Else** statement in your **Select Case** block to handle unforeseen *testexpression* values. If no **Case** *expressionlist* clause matches *testexpression* and there is no **Case Else** statement, execution continues at the statement following **End Select**.

You can use multiple expressions or ranges in each **Case** clause. For example, the following line is valid:

```
Case 1 To 4, 7 To 9, 11, 13, Is > MaxNumber
```

Note The **Is** keyword used in the **Case** and **Case Else** statements is not the same as the **Is** comparison operator.

You also can specify ranges and multiple expressions for character strings. In the following example, **Case** matches strings that

are exactly equal to "apples", strings with values between "nuts" and "soup" in alphabetical order, and the current value of `TestItem`:

```
Case "apples", "nuts" To "soup", TestItem
```

The setting of **Option Compare** can affect string comparisons. Under **Option Compare Text**, the strings "Apples" and "apples" compare as equal, but under **Option Compare Binary**, they do not.

Select Case statements can be nested. Each nested **Select Case** statement must have a matching **End Select** statement.

If you do not need to execute any more of the statements in a **Case** or **Case Else** statement block, you can exit the block by using the **Exit Select** statement. This transfers control immediately to the statement following **End Select**.

Example

This example uses the **Select Case** statements to write a line corresponding to the value of the variable `Number`. The second **Case** statement contains the value that matches the current value of `Number`, so the statement that writes "Between 6 and 8" is executed.

```
Dim Number As Integer = 8
' ...
Select Number   ' Evaluate Number.
   Case 1 To 5   ' Number between 1 and 5, inclusive.
      Debug.WriteLine("Between 1 and 5")
   ' The following is the only Case clause that evaluates to True.
   Case 6, 7, 8   ' Number between 6 and 8.
      Debug.WriteLine("Between 6 and 8")
   Case 9 To 10   ' Number is 9 or 10.
      Debug.WriteLine("Greater than 8")
   Case Else   ' Other values.
      Debug.WriteLine("Not between 1 and 10")
End Select
```

See Also

Choose Function | End Statement | If...Then...Else Statements | Select...Case Statements (Conceptual) | Option Compare Statement | Exit Statement

Set Statement

Declares a **Set** property procedure used to assign a value to a property.

```
[ <attrlist> ] Set(ByVal value [ As datatype ])
   [ block ]
End Set
```

Parts

attrlist
Optional. List of attributes that apply to this **Set** block. Multiple attributes are separated by commas.
value
Required. This parameter contains the new value for the property.
datatype
Required if **Option Strict** is **On**. Declares the data type of the value parameter. The data type specified must be the same as the data type of the property where this **Set** statement is declared.
block
Optional. The body of the **Set** procedure contains code to set the property value.
End Set
Terminates a **Set** property procedure.

Each attribute in the *attrlist* part has the following syntax and parts:

```
attrname [({ attrargs | attrinit })]
```

attrlist Parts

attrname
Required. Name of the attribute. Must be a valid Visual Basic identifier.
attrargs
Optional. List of positional arguments for this attribute. Multiple arguments are separated by commas.
attrinit
Optional. List of field or property initializers for this attribute. Multiple initializers are separated by commas.

Remarks

Visual Basic .NET calls the **Set** property procedure when the value of a property is changed in an assignment operation.

Example

This example uses the **Set** statement to set the value of a property.

```
Class PropClass
   Private PropVal As Integer
   Property Prop1() As Integer
      Get
         Return PropVal ' Same As Prop1 = PropVal
      End Get
      Set(ByVal Value As Integer)
         PropVal = Value
      End Set
   End Property
End Class
```

See Also

Get Statement | Property Statement | Adding Fields and Properties to a Class

Stop Statement

Suspends execution.

```
Stop
```

Remarks

You can place **Stop** statements anywhere in procedures to suspend execution. Using the **Stop** statement is similar to setting a breakpoint in the code.

The **Stop** statement suspends execution, but unlike **End**, it does not close any files or clear any variables, unless it is encountered in a compiled executable (.exe) file.

> **Note** If the **Stop** statement is encountered in code that is running outside of the integrated development environment (IDE), the debugger is invoked. This is true regardless of whether the code was compiled in debug or retail mode.

Example

This example uses the **Stop** statement to suspend execution for each iteration through the **For...Next** loop.

```
Dim I As Integer
For I = 1 To 10   ' Start For...Next loop.
   Debug.WriteLine (I)   ' Print I to the Output window.
   Stop   ' Stop during each iteration and wait for user to resume.
Next I
```

See Also

End Statement

Structure Statement

Used at module or class level to declare a structure and define the characteristics of its members.

```
[ <attrlist> ] [{ Public | Protected | Friend | Protected Friend | Private }] [ Shadows ]
Structure name
   [ Implements interfacenames ]
   variabledeclarations
   [ proceduredeclarations ]
End Structure
```

Parts

attrlist
Optional. List of attributes that apply to this structure. Multiple attributes are separated by commas.

Public
Optional. Structures declared with the **Public** keyword have public access. There are no restrictions on the accessibility of public structures.

Protected
Optional. Structures declared with the **Protected** keyword have protected access. They are accessible only from within their own class or from a derived class. Protected access can be specified only on members of classes. It is not a superset of friend access.

Friend
Optional. Structures declared with the **Friend** keyword have friend access. They are accessible from within their declaration context and from anywhere else in the same program.

Protected Friend
Optional. Structures declared with the **Protected Friend** modifiers have the union of protected and friend access. They can be used by code in the same program, as well as by code in derived classes. Protected friend access can be specified only on members of classes.

Private
Optional. Structures declared with the **Private** modifier have private access. They are accessible only from within their declaration context, including from members of any nested types such as procedures.

Shadows
Optional. Indicates that this structure shadows an identically named programming element, or set of overloaded elements, in a base class. You can shadow any kind of declared element with any other kind. A shadowed element is unavailable from within the derived class that shadows it, unless the shadowing element is inaccessible, for example if it is **Private**.

name
Required. Name of the structure. Must be a valid Visual Basic identifier.

Implements
Optional. Indicates that this structure implements the members of one or more interfaces.

interfacenames
Required if the **Implements** statement is used. The names of the interfaces implemented by this structure. If you use the **Implements** statement, it must immediately follow the **Structure** statement, and you must implement every member defined by every interface you specify.

variabledeclarations
Required. One or more **Dim**, **Event**, **Friend**, **Private**, or **Public** statements declaring variables and events that serve as data members of the structure. These declarations follow the same rules as they do outside of a structure.

You can also define constants and properties in the structure, but you must declare at least one nonshared variable or event.

proceduredeclarations
Optional. Zero or more declarations of **Function**, **Property**, or **Sub** procedures that serve as method members of the structure. These declarations follow the same rules as they do outside of a structure.

Each attribute in the *attrlist* part has the following syntax and parts:

```
attrname [({ attrargs | attrinit })]
```

***attrlist* Parts**

attrname
Required. Name of the attribute. Must be a valid Visual Basic identifier.
attrargs
Optional. List of positional arguments for this attribute. Multiple arguments are separated by commas.
attrinit
Optional. List of field or property initializers for this attribute. Multiple initializers are separated by commas.

Remarks

The **Structure** statement can appear only at module, namespace, or file level. This means you can declare structures in a source file or inside a module, interface, or class, but not inside a procedure. You can also define one structure inside another, but you cannot access its members through the outer structure. Instead, you must declare a variable of the inner structure's data type.

Structures can be accessed from anywhere within the module or class in which they are declared. A structure is **Friend** by default. To specify the accessibility in more detail, include **Public**, **Protected**, **Friend**, **Protected Friend**, or **Private** in the **Structure** statement.

You must declare every data member of a structure. This means every statement in the *variabledeclarations* part must use **Dim**, **Friend**, **Private**, or **Public**. A structure member cannot be **Protected** or **Protected Friend** because nothing can inherit from a structure. The structure itself, however, can be **Protected** or **Protected Friend**.

If **Option Strict** is **On**, you must also include the **As** clause in every member declaration. Members declared with **Dim** default to **Public** access, and members declared without the **As** clause default to the **Object** data type.

You must define at least one nonshared variable or event in a structure. You cannot have only constants, properties, and procedures, even if some of them are nonshared.

The scope of every structure member is the entire structure.

You cannot initialize the value of any data member of a structure as part of its declaration. You must either initialize a data member by means of a parameterized constructor on the structure, or assign a value to the member after you have created an instance of the structure.

Structures support many of the same features as classes. For example, structures can have properties and methods, they can implement interfaces, and they can have parameterized constructors. However, there are significant differences between structures and classes in areas such as inheritance, declarations, and usage.

Example

This example uses the **Structure** statement to define a set of related data for an employee. It shows the use of **Public**, **Friend**, and **Private** members to reflect the sensitivity of the data items. It also shows procedure, property, and event members.

```
Public Structure Employee
   ' Public members, accessible throughout declaration region.
   Public FirstName As String
   Public MiddleName As String
   Public LastName As String
   ' Friend members, accessible anywhere within the same program.
   Friend EmployeeNumber As Integer
   Friend WorkPhone As Long
   ' Private members, accessible only within the structure itself.
   Private HomePhone As Long
   Private Level As Integer
   Private Salary As Double
   Private Bonus As Double
   ' Procedure member, which can access structure's private members.
   Friend Sub CalculateBonus(ByVal Rate As Single)
      Bonus = Salary * CDbl(Rate)
   End Sub
   ' Property member to return employee's eligibility.
   Friend ReadOnly Property Eligible() As Boolean
      Get
         Return Level >= 25
      End Get
   End Property
   ' Event member, raised when business phone number has changed.
   Public Event ChangedWorkPhone(ByVal NewPhone As Long)
```

```
End Structure
```

See Also

Dim Statement | Implements Statement | Structures and Classes | Event Statement | Property Statement

Sub Statement

Declares the name, arguments, and code that define a **Sub** procedure.

```
[ <attrlist> ] [{ Overloads | Overrides | Overridable |
NotOverridable | MustOverride | Shadows | Shared }]
[{ Public | Protected | Friend | Protected Friend | Private }]
Sub name [(arglist)] [ Implements interface.definedname ]
   [ statements ]
   [ Exit Sub ]
   [ statements ]
End Sub
```

Parts

attrlist
Optional. List of attributes that apply to this procedure. Multiple attributes are separated by commas.

Overloads
Optional. Indicates that this **Sub** procedure overloads one or more procedures defined with the same name in a base class. The argument list in this declaration must be different from the argument list of every overloaded procedure. The lists must differ in the number of arguments, their data types, or both. This allows the compiler to distinguish which version to use.

You do not have to use the **Overloads** keyword when you are defining multiple overloaded procedures in the same class. However, if you use **Overloads** in one of the declarations, you must use it in all of them.

You cannot specify both **Overloads** and **Shadows** in the same procedure declaration.

Overrides
Optional. Indicates that this **Sub** procedure overrides an identically named procedure in a base class. The number and data types of the arguments must exactly match those of the base class procedure.

Overridable
Optional. Indicates that this **Sub** procedure can be overridden by an identically named procedure in a derived class. **Overridable** is the default setting for a procedure that itself overrides a base class procedure.

NotOverridable
Optional. Indicates that this **Sub** procedure cannot be overridden in a derived class. **NotOverridable** is the default setting for a procedure that does not itself override a base class procedure.

MustOverride
Optional. Indicates that this **Sub** procedure is not implemented in this class and must be implemented in a derived class for that class to be creatable.

Shadows
Optional. Indicates that this **Sub** procedure shadows an identically named programming element, or set of overloaded elements, in a base class. You can shadow any kind of declared element with any other kind. If you shadow a procedure with another procedure, the arguments and the return type do not have to match those in the base class procedure. A shadowed element is unavailable from within the derived class that shadows it, unless the shadowing element is inaccessible, for example if it is **Private**.

You cannot specify both **Overloads** and **Shadows** in the same procedure declaration.

Shared
Optional. Indicates that this **Sub** procedure is a shared procedure. This means it is not associated with a specific instance of a class or structure. You can call a shared procedure by qualifying it either with the class or structure name, or with the variable name of a specific instance of the class or structure.

Public
Optional. Procedures declared with the **Public** keyword have public access. There are no restrictions on the accessibility of public procedures.

Protected
Optional. Procedures declared with the **Protected** keyword have protected access. They are accessible only from within their own class or from a derived class. Protected access can be specified only on members of classes. It is not a superset of friend access.

Friend
Optional. Procedures declared with the **Friend** keyword have friend access. They are accessible from within their declaration context and from anywhere else in the same program.

Protected Friend
Optional. Procedures declared with the **Protected Friend** keywords have the union of protected and friend access. They can be used by code in the same program, as well as by code in derived classes. Protected friend access can be specified only on members of classes.
Private
Optional. Procedures declared with the **Private** keyword have private access. They are accessible only from within their declaration context, including from members of any nested types such as procedures.
name
Required. Name of the **Sub** procedure. Must be a valid Visual Basic identifier.
arglist
Optional. List of variables or expressions representing arguments that are passed to the **Sub** procedure when it is called. Multiple arguments are separated by commas. If you supply an argument list, you must enclose it in parentheses.
Implements
Optional. Indicates that this **Sub** procedure implements a **Sub** procedure defined by an interface.
interface
Required if **Implements** is supplied. The interface implemented by the class or structure containing this **Sub** procedure. The class or structure must specify *interface* in an **Implements** statement immediately following the **Class** or **Structure** statement.
definedname
Required if **Implements** is supplied. The name by which the **Sub** procedure is defined in *interface*. The name of this **Sub** procedure (in *name*) does not have to be the same as *definedname*.
statements
Optional. A block of statements to be executed within the **Sub** procedure.
End Sub
Terminates the definition of this **Sub** procedure.

Each argument in the *arglist* part has the following syntax and parts:

```
[ <attrlist> ] [ Optional ] [{ ByVal | ByRef }] [ ParamArray ]
argname[( )] [ As argtype ] [ = defaultvalue ]
```

***arglist* Parts**

attrlist
Optional. List of attributes that apply to this argument. Multiple attributes are separated by commas.
Optional
Optional. Indicates that this argument is not required when the procedure is called. If this keyword is used, all subsequent arguments in *arglist* must also be optional and be declared using the **Optional** keyword. Every optional argument declaration must supply the *defaultvalue* clause. **Optional** cannot be used for any argument if **ParamArray** is used.
ByVal
Optional. Indicates that the procedure cannot replace or reassign the underlying variable element in the calling code. However, if the argument is a reference type, the procedure can modify the contents or members of the underlying object. **ByVal** is the default in Visual Basic.
ByRef
Optional. Indicates that the procedure can modify the underlying variable in the calling code the same way the calling code itself can.
ParamArray
Optional. Used only as the last argument in *arglist* to indicate that the final argument is an optional array of elements of the specified type. The **ParamArray** keyword allows you to pass an arbitrary number of arguments to the procedure. A **ParamArray** argument is always passed **ByVal**.
argname
Required. Name of the variable representing the argument. Must be a valid Visual Basic identifier.
argtype
Optional unless **Option Strict** is **On**. Data type of the argument passed to the procedure. Can be **Boolean**, **Byte**, **Char**, **Date**, **Decimal**, **Double**, **Integer**, **Long**, **Object**, **Short**, **Single**, or **String**; or the name of an enumeration, structure, class, or interface.
defaultvalue
Required for **Optional** arguments. Any constant or constant expression that evaluates to the data type of the argument. If the type is **Object**, or a class, interface, array, or structure, the default value can only be **Nothing**.

Each attribute in the *attrlist* part has the following syntax and parts:

```
attrname [({ attrargs | attrinit })]
```

attrlist Parts

attrname
Required. Name of the attribute. Must be a valid Visual Basic identifier.
attrargs
Optional. List of positional arguments for this attribute. Multiple arguments are separated by commas.
attrinit
Optional. List of field or property initializers for this attribute. Multiple initializers are separated by commas.

Remarks

All executable code must be in procedures. You can define a **Sub** procedure inside a module, class, interface, or structure, but not inside another procedure.

Sub procedures are **Public** by default. To specify a different accessibility, include **Protected**, **Friend**, **Protected Friend**, or **Private** in the declaration.

When the **Sub** procedure returns to the calling program, execution continues with the statement following the statement that called it.

The **Exit Sub** statement causes an immediate exit from a **Sub** procedure. Any number of **Exit Sub** statements can appear anywhere in the procedure.

You can also use the **Return** statement to immediately exit the procedure, as the following example shows:

```
Sub MySub(ByVal Q As String)
   ' ...
   Return
   ' ...
End Sub
```

Any number of **Return** statements can appear anywhere in the procedure. You can also mix **Exit Sub** and **Return** statements.

A **Sub** procedure, like a **Function** procedure, is a separate procedure that can take arguments and perform a series of statements. However, unlike a **Function** procedure, which returns a value, a **Sub** procedure cannot be used in an expression.

You call a **Sub** procedure by using the procedure name, followed by the argument list in parentheses, in a statement. You can omit the parentheses only if you are not supplying any arguments. You can optionally use the **Call** statement to call a **Sub** procedure. This can improve the readability of your program.

Example

This example uses the **Sub** statement to define the name, arguments, and code that form the body of a **Sub** procedure.

```
' Sub procedure definition.
' Sub procedure with two arguments.
Sub SubComputeArea(ByVal Length As Double, ByVal Width As Double)
   Dim Area As Double   ' Declare local variable.
   If Length = 0 Or Width = 0 Then
      ' If either argument = 0.
      Exit Sub ' Exit Sub immediately.
   End If
   Area = Length * Width   ' Calculate area of rectangle.
   Debug.WriteLine(Area)   ' Print Area to Immediate window.
End Sub
```

See Also

Call Statement | Function Statement | Dim Statement | Implements Statement

SyncLock Statement

Allows statements to be synchronized on a single expression.

```
SyncLock expression
...[ block ]
End SyncLock
```

Parts

expression
Required. A unique collection of operators and values that yield a single result.

block
Optional. The statements that will execute in sequence.

End SyncLock
Terminates a **SyncLock** procedure.

Remarks

The **SyncLock** statement ensures that multiple threads do not execute the same statements at the same time. When the thread reaches the **SyncLock** block, it evaluates the expression and maintains this exclusivity until it has a lock on the object that is returned by the expression. This prevents an expression from changing values during the running of several threads, which can give unexpected results from your code.

> **Note** The type of the expression in a **SyncLock** statement must be a reference type, such as a class, a module, an interface, array or delegate.

Example

```
Class Cache
   Private Shared Sub Add(ByVal x As Object)
      SyncLock GetType(Cache)
      End SyncLock
   End Sub

   Private Shared Sub Remove(ByVal x As Object)
      SyncLock GetType(Cache)
      End SyncLock
   End Sub
End Class
```

See Also

Thread Synchronization | Multithreading in Visual Basic .NET

Throw Statement

Throws an exception within a procedure.

```
Throw expression
```

Part

expression
Required. Provides information about the exception to be thrown.

Remarks

The **Throw** statement throws an exception that you can handle with structured exception-handling code (**Try**...**Catch**...**Finally**) or unstructured exception-handling code (**On Error GoTo**). You can use the **Throw** statement to trap errors within your code, because Visual Basic moves up the call stack until it finds the appropriate exception-handling code.

Example

The following code uses the **Throw** statement to throw an exception:

```
' Throws a new exception.
Throw New System.Exception("An exception has occurred.")
```

Requirements

Namespace: Microsoft.VisualBasic

Module: Interaction

Assembly: Microsoft Visual Basic .NET Runtime (in Microsoft.VisualBasic.dll)

See Also

Try...Catch...Finally Statements | On Error Statement | Structured Exception Handling | Unstructured Exception Handling

Try...Catch...Finally Statements

Provides a way to handle some or all possible errors that may occur in a given block of code, while still running code.

```
Try
   [ tryStatements ]
[ Catch [ exception [ As type ] ] [ When expression ]
   [ catchStatements ] ]
[ Exit Try ]
...
[ Finally
   [ finallyStatements ] ]
End Try
```

Parts

tryStatements
Optional. Statement(s) where an error can occur. Can be a compound statement.

Catch
Optional. Multiple **Catch** blocks permitted. If an exception occurs while processing the **Try** block, each **Catch** statement is examined in textual order to determine if it handles the exception. *Exception* represents the exception that has been thrown.

exception
Optional. Any variable name. The initial value of *exception* is the value of the thrown error. Used with **Catch** to specify the error caught.

type
Optional. Specifies the type of class filter. If the value of *exception* is of the type specified by *type* or of a derived type, the identifier becomes bound to the exception object.

When
Optional. A **Catch** statement with a **When** clause will only catch exceptions when *expression* evaluates to **True**. A **When** clause is only applied after checking the type of the exception, and *expression* may refer to the identifier representing the exception.

expression
Optional. Must be implicitly convertible to **Boolean**. Any expression that describes a generic filter. Typically used to filter by error number. Used with **When** keyword to specify circumstances under which the error is caught.

catchStatements
Optional. Statement(s) to handle errors occurring in the associated **Try** block. Can be a compound statement.

Exit Try
Optional. Keyword that breaks out of the **Try...Catch...Finally** structure. Execution resumes with the **Finally** block if present, otherwise with the code immediately following the **End Try** statement. Not allowed in **Finally** blocks.

Finally
Optional. A **Finally** block is always executed when execution leaves any part of the **Try** statement.

finallyStatements
Optional. Statement(s) that are executed after all other error processing has occurred.

End Try
Terminates the **Try...Catch...Finally** structure.

Remarks

Local variables from a **Try** block are not available in a **Catch** block because they are separate blocks. If you want to use a variable in more than one block, declare the variable outside the **Try...Catch...Finally** structure.

If errors occur that the programmer has not handled, Visual Studio for Applications simply provides its normal error message to a user, as if there was no error handling.

The **Try** block contains code where an error can occur, while the **Catch** block contains code to handle any error that does occur. If an error occurs in the **Try** block, program control is passed to the appropriate **Catch** statement for disposition. The *exception* argument is an instance of the **Exception** class or an instance of a class that derives from the **Exception** class corresponding to the error that occurred in the **Try** block. The **Exception** class instance contains information about the error including, among other things, its number and message.

In partial trust situations, such as an application hosted on a network share, **Try...Catch...Finally** will not catch security exceptions that occur before the method containing the call is invoked. The following example, when placed on a server share and run from there, for example, will produce the error: "Sub System.Security.SecurityException: Request Failed. For more information on

Security Exceptions, see SecurityException Class.

```
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As _
System.EventArgs) Handles Button1.Click
   Try
      Process.Start("http://www.microsoft.com")
   Catch ex As Exception
      MsgBox("Can't load Web page" & vbCrLf & ex.Message)
   End Try
End Sub
```

In such a partial trust situation, you would need to put the Process.Start statement in a separate **Sub**. The initial call to the **Sub** will fail, allowing **Try...Catch** to catch it before the Sub containing Process.Start is launched and the security exception produced.

Note If a **Try** statement does not contain at least one **Catch** block it must contain a **Finally** block.

Example

The following simplified example illustrates the structure of the **Try...Catch...Finally** statement:

```
Public Sub TryExample()
   Dim x As Integer = 5   ' Declare variables.
   Dim y As Integer = 0
   Try   ' Set up structured error handling.
      x /= y   ' Cause a "Divide by Zero" error.
   Catch ex As Exception When y = 0   ' Catch the error.
      MsgBox(ex.toString)   ' Show friendly error message.
   Finally
      Beep()   ' Beep after error processing.
   End Try
End Sub
```

See Also

End Statement | Err Object | Exit Statement | On Error Statement | Try...Catch...Finally Statements (Conceptual) | Exception Class

While...End While Statements

Executes a series of statements as long as a given condition is **True**.

```
While condition
   [ statements ]
End While
```

Parts

condition
Required. Expression. Must evaluate to **True** or **False**. If *condition* is **Nothing**, *condition* is treated as **False**.
statements
Optional. One or more statements following **While** that are executed while *condition* is **True**.
End While
Terminates execution of the **While** block.

Remarks

If *condition* is **True**, all of the *statements* are executed until the **End While** statement is encountered. Control then returns to the **While** statement and *condition* is again checked. If *condition* is still **True**, the process is repeated. If it is **False**, execution resumes with the statement following the **End While** statement.

You can nest **While** loops by placing one loop within another.

Example

This example uses the **While...End While** statement to increment a counter variable. The statements in the loop are executed as long as the condition evaluates to **True**.

```
Dim Counter As Integer = 0
While Counter < 20   ' Test value of Counter.
   Counter += 1   ' Increment Counter.
End While   ' End While loop when Counter > 19.
Debug.WriteLine (Counter)   ' Prints 20 in the Output window.
```

See Also

Do...Loop Statements | Do...Loop Statements (Conceptual) | With...End With Statements | While Statement (Conceptual)

With...End With Statements

Executes a series of statements making repeated reference to a single object or structure.

```
With object
   [ statements ]
End With
```

Parts

object
Required. Expression. Can evaluate to any data type, including elementary types.
statements
Optional. One or more statements following **With** that are executed on *object*.
End With
Terminates execution of the **With** block.

Remarks

With...End With allows you to perform a series of statements on a specified object without requalifying the name of the object. For example, to change a number of different properties on a single object, place the property assignment statements within the **With...End With**, referring to the object once instead of referring to it with each property assignment. The following example illustrates use of **With...End With** to assign values to several properties of the same object:

```
With MyLabel
   .Height = 2000
   .Width = 2000
   .Text = "This is MyLabel"
End With
```

> **Note** Once you have entered a **With...End With**, you cannot reassign *object* until you have passed the **End With**. Therefore, you can access the methods and properties of only the specified object without qualifying them. You can use methods and properties of other objects, but you must qualify them with their object names.

You can nest **With...End With** statements by placing one within another. However, because members of outer statements are masked inside the inner statements, you must provide a fully qualified object reference in an inner **With...End With** to any member of an object in an outer statement.

You cannot use **GoTo** to branch from outside a **With...End With** to a label inside it. If you with to exit before all the statements have been executed, put a label on the **End With** statement and branch to that.

Example

This example uses the **With** statement to execute a series of statements on a single object. The object `MyObject` and its properties are generic names used for illustration purposes only.

```
With MyObject
   .Height = 100   ' Same as MyObject.Height = 100.
   .Text = "Hello World"   ' Same as MyObject.Text = "Hello World".
   .ForeColor = Color.Green   ' Same as MyObject.ForeColor = Color.Green.
   .Font = New Font(.Font, FontStyle.Bold)   ' Same as:
                ' MyObject.Font = New Font(MyObject.Font, FontStyle.Bold).
End With
```

See Also

With...End With Statements (Conceptual)

Error Messages

Error messages can occur while an application is running, either within the Visual Basic .NET environment or as a stand-alone executable. Some of these can also occur during design time or compile time. You can test and respond to trappable errors using the **On Error** statement and the **Err** object's **Number** property. Unused error numbers in the range 1 to 1000 are reserved for future use by Visual Basic.

> **Note** The description "Application-defined or object-defined error" occurs when an error is generated using the **Err** object's **Raise** method or the **Error** statement, but the number does not correspond to an error defined by the Visual Basic language. Such errors may be defined by the host application (for example, Microsoft Excel or Visual Basic), but if you want to generate them from code, you must use the **Raise** method, and fill in all relevant arguments.

While developing in Visual Basic you can get information from Help on a message by pressing F1. Likewise, you press F1 on error messages shown in the Task List window.

In This Section

Visual Basic Compiler Errors
Explains the concept of compiler errors and how to get help for particular error messages.
Visual Basic Run-Time Errors
Explains the concept of run-time errors and how to get help for particular error messages.

Related Sections

Exception Handling
Lists links to topics related to exception handling in Visual Basic .NET.
Walkthrough: Structured Exception Handling
Illustrates how to use the **Try...Catch...Finally** statement to handle exceptions.

Visual Basic Compiler Errors

Visual Basic compiler errors occur when the compiler encounters problems in the code. The code causing the errors is marked with a wavy line underneath it in the Code Editor, and when your mouse pauses over the marked code, the error message appears. The messages also appear in the Task List.

To get help for a particular error message

- Highlight the error message in the **Task List** and press F1.

-or-

1. Click **Search** on the **Help** menu.
2. Type the error message in the **Look for** box in the Search window.

> **Note** Remember that displayed error messages often incorporate identifiers from the code, such as variable or project names, as well as the type names for these elements; these are not part of the underlying error message. Be careful not to include these words as part of your search.

See Also

Visual Basic Run-Time Errors | Debugging Code in the Editor

Visual Basic Run-Time Errors

Visual Basic run-time errors occur when an application attempts to perform an action that the system cannot execute.

The errors that Visual Basic .NET throws are **Error** objects. Visual Basic .NET can generate custom errors of any data type, including **Error** objects, by using the **Throw** statement. A program can display the error number and message of a caught **Error** to identify the error. If an error is not caught, the program ends.

Run-time errors can be trapped and examined by the code. By enclosing the code that produces the error in a **Try** block, any thrown error can be caught within a matching **Catch** block.

To get help for a particular error message

1. Click **Search** on the **Help** menu.
2. Type the error message in the **Look for** box in the Search window.

> **Note** Some error messages include words in single quotes that refer to identifiers from the code, such as a project or variable name, and are not part of the error message. Do not include the quoted words as part of your search.

See Also

Visual Basic Compiler Errors | Try...Catch...Finally Statements | Structured Exception Handling

Visual Basic Compiler Options

The Visual Basic .NET command-line compiler is provided as an alternative to compiling programs from within the Visual Studio .NET integrated development environment (IDE). This section contains descriptions for the Visual Basic compiler options.

In This Section

Building From the Command Line
Describes the Visual Basic .NET command-line compiler, which is provided as an alternative to compiling programs from within the Visual Studio .NET integrated development environment (IDE).

Visual Basic Compiler Options Listed Alphabetically
Lists compiler options in an alphabetical table.

Visual Basic Compiler Options Listed by Category
Presents compiler options in functional groups.

Related Sections

<Projectname> Property Pages Dialog Box
Discusses how to use the **<Projectname>** Property Pages dialog box to specify global settings for your project.

Visual Basic and Visual C#
Covers designing, developing, debugging, testing, and deploying applications using Visual Basic or Visual C#.

Building From the Command Line

A Visual Basic .NET project is made up of one or more separate source files. During the process known as *compilation*, these files are brought together into one package — a single executable file that can be run as an application.

Visual Basic .NET provides a command-line compiler as an alternative to compiling programs from within the Visual Studio .NET integrated development environment (IDE). The command-line compiler is designed for situations in which you do not require the full set of features in the IDE; for example, when you are using or writing for computers with limited system memory or storage space.

When compiling from the command line, you must explicitly reference the Microsoft Visual Basic Run-Time Library through the **/reference** compiler option.

To compile source files from within the Visual Studio .NET IDE, choose the **Build** command from the **Build** menu. To compile project (.vbproj) files at the command line, see Devenv Command Line Switches.

In This Section

Invoking the Command-Line Compiler
Describes how to invoke the command-line compiler at the MS-DOS prompt or from a specific subdirectory.
Sample Compilation Command Lines
Provides a list of sample command lines that you can modify for your own use.

Related Sections

Visual Basic Compiler Options
Provides lists of compiler options, organized alphabetically or by purpose.
Conditional Compilation Overview
Describes how to compile particular sections of code.
Preparing and Managing Builds
Describes how to organize what will be included in different builds, choose project properties, and ensure that projects build in the correct order.

Invoking the Command-Line Compiler

You can invoke the command-line compiler by typing the name of its executable file at the command line, also known as the MS-DOS prompt.

When compiling from the command line, you must explicitly reference the Microsoft Visual Basic Run-Time Library through the **/reference** compiler option.

To invoke the compiler

- At the command line, type **vbc.exe /reference:Microsoft.VisualBasic.dll**, and then press ENTER.

You may need to adjust your path if you want to invoke the compiler from a specific subdirectory on your computer. For example, if you stored your executables in a directory called "SourceFiles", you would open the command prompt, type "cd SourceFiles" to change to that directory, and then type **vbc.exe**.

See Also

Visual Basic Compiler Options | Conditional Compilation Overview | /reference

Sample Compilation Command Lines

As an alternative to compiling Visual Basic .NET programs from within Visual Studio .NET, you can compile from the command line to produce executable (.exe) files or dynamic-link library (.dll) files.

The Visual Basic .NET command-line compiler supports a complete set of options that control input and output files, assemblies, and debug and preprocessor options. Each option is available in two interchangeable forms: **-***option* and */option*. This documentation shows only the */option* form.

The following table lists some sample command lines you can modify for your own use.

To	Use
Compile File.vb and create File.exe	**vbc /reference:Microsoft.VisualBasic.dll File.vb**
Compile File.vb and create File.dll	**vbc /target:library File.vb**
Compile File.vb and create My.exe	**vbc /out:My.exe File.vb**
Compile all Visual Basic files in the current directory, with optimizations on and the DEBUG symbol defined, producing File2.exe	**vbc /define:DEBUG=1 /optimize /out:File2.exe *.vb**
Compile all Visual Basic files in the current directory, producing a debug version of File2.dll without displaying the logo or warnings	**vbc /target:library /out:File2.dll /warn:0 /nologo /debug *.vb**
Compile all Visual Basic files in the current directory to Something.dll	**vbc /target:library /out:Something.dll *.vb**

When compiling from the command line, you must explicitly reference the Microsoft Visual Basic Run-Time Library through the **/reference** compiler option.

See Also

Visual Basic Compiler Options | Conditional Compilation Overview

Visual Basic Compiler Options Listed Alphabetically

The Visual Basic .NET command-line compiler is provided as an alternative to compiling programs from within the Visual Studio .NET integrated development environment (IDE). The following is a list of the Visual Basic command-line compiler options sorted alphabetically.

Option	Purpose
@ (Specify Response File)	Specifies a response file.
/?	Displays compiler options. This command is the same as specifying the **/help** option. No compilation occurs.
/addmodule	Causes the compiler to make all type information from the specified file(s) available to the project you are currently compiling.
/baseaddress	Specifies the base address of a DLL.
/bugreport	Creates a file that contains information that makes it easy to report a bug.
/debug	Produces debugging information.
/define	Defines symbols for conditional compilation.
/delaysign	Specifies whether the assembly will be fully or partially signed.
/help	Displays compiler options. This command is the same as specifying the **/?** option. No compilation occurs.
/imports	Imports a namespace from a specified assembly.
/keycontainer	Specifies a key container name for a key pair to give an assembly a strong name.
/keyfile	Specifies a file containing a key or key pair to give an assembly a strong name.
/libpath	Specifies the location of assemblies referenced via the /reference option.
/linkresource	Creates a link to a managed resource.
/main	Specifies the class that contains the **Sub Main** procedure to use at startup.
/netcf	Sets the compiler to target the .NET Compact Framework.
/nologo	Suppresses compiler banner information.
/nowarn	Suppresses the compiler's ability to generate warnings.
/optimize	Enables/disables code optimization.
/optioncompare	Determines whether string comparisons should be binary or use locale-specific text semantics.
/optionexplicit	Requires explicit declaration of variables.
/optionstrict	Enforces strict language semantics.
/out	Specifies an output file.
/quiet	Prevents the compiler from displaying code for syntax-related errors and warnings.
/recurse	Searches subdirectories for source files to compile.
/reference	Imports metadata from an assembly.
/removeintchecks	Disables integer overflow checking.
/resource	Embeds a managed resource in an assembly.
/rootnamespace	Specifies a namespace for all type declarations.
/sdkpath	Specifies the location of mscorlib.dll and microsoft.visualbasic.dll.
/target	Specifies the format of the output file using one of four options: /target:exe, /target:library, /target:module, or /target:winexe.
/utf8output	Displays compiler output using UTF-8 encoding.
/verbose	Outputs extra information during compilation.
/warnaserror	Promotes warnings to errors.
/win32icon	Inserts a .ico file into the output file.
/win32resource	Inserts a Win32 resource into the output file.

See Also

Visual Basic Compiler Options Listed by Category | <Projectname> Property Pages Dialog Box

/help, /?

Displays the compiler options.

```
/help
```

-or-

```
/?
```

Remarks

If you include this option in a compilation, no output file is created and no compilation takes place.

> **Note** The **/help** option is not available from within the Visual Studio development environment; it is available only when compiling from the command line. It cannot be changed programmatically.

Example

The following code displays help from the command line:

```
vbc /help
```

See Also

Visual Basic Compiler Options | Sample Compilation Command Lines

@ (Specify Response File)

Specifies a file that contains compiler options and source code files to compile.

```
@response_file
```

Arguments

response_file
Required. A file that lists compiler options or source code files to compile.

Remarks

The compiler processes the compiler options and source code files specified in a response file as if they had been specified on the command line.

To specify more than one response file in a compilation, specify multiple response file options. For example:

```
@file1.rsp @file2.rsp
```

In a response file, multiple compiler options and source code files can appear on one line. A single compiler option specification must appear on one line (cannot span multiple lines). Response files can have comments that begin with the # symbol.

You can combine options specified on the command line with options specified in one or more response files. The compiler processes the command options as it encounters them. Therefore, command-line arguments can override previously listed options in response files. Conversely, options in a response file override options listed previously on the command line or in other response files.

> **Note** The **@** option is not available from within the Visual Studio development environment; it is available only when compiling from the command line. It cannot be changed programmatically.

Example

The following are a few lines from a sample response file:

```
# build the first output file
/target:exe
/out:MyExe.exe
source1.vb
source2.vb
```

See Also

Visual Basic Compiler Options | Sample Compilation Command Lines

/addmodule

Causes the compiler to make all type information from the specified file(s) available to the project you are currently compiling.

```
/addmodule:module[,module2]
```

Arguments

module, *module2*
Required. A file that contains metadata but not an assembly manifest. To import more than one file, separate file names with a comma.

Remarks

module is a file created with /target:module, or another compiler's equivalent to **/target:module**.

All modules added with **/addmodule** must be in the same directory as the output file at run time. That is, you can specify a module in any directory at compile time, but the module must be in the application directory at run time. If it is not, you get a **System.TypeLoadException** error.

If you specify (implicitly or explicitly) any /target option other than **/target:module** with **/addmodule**, the files you pass to **/addmodule** become part of the project's assembly. An assembly is required to run an output file that has one or more files added with **/addmodule**.

Use /reference to import metadata from a file that contains an assembly.

> **Note** The **/addmodule** option is not available from within the Visual Studio development environment; it is available only when compiling from the command line. It cannot be changed programmatically.

Example

The following code creates a module:

```
' t1.vb
' Compile with vbc /target:module t1.vb.
' Outputs t1.netmodule.

Public class TestClass
   Public i As Integer
End class
```

The following code imports the module's types:

```
' t2.vb
' Compile with vbc /addmodule:t1.netmodule t2.vb.
Option Strict Off
Module Module1
   Sub Main()
      Dim x As TestClass
      x = New TestClass
      x.i = 802
      System.Console.WriteLine(x.i)
   End Sub
End Module
```

When you run `t1`, it outputs `802`.

See Also

Visual Basic Compiler Options | /target | Sample Compilation Command Lines

/baseaddress

Specifies a default base address when creating a DLL.

```
/baseaddress:address
```

Arguments

address
Required. The base address for the DLL. This address must be specified as a hexadecimal number.

Remarks

The default base address for a DLL is set by the .NET Framework.

Be aware that the lower-order word in this address is rounded. For example, if you specify 0x11110001, it is rounded to 0x11110000.

To complete the signing process for a DLL, use the **–R** option of the Strong Naming tool (Sn.exe).

This option is ignored if the target is not a DLL.

To set /baseaddress in the Visual Studio integrated development environment

1. Open the project's Property Pages dialog box. For details, see <Projectname> Property Pages Dialog Box.
2. Click the Configuration Properties folder.
3. Click the Optimizations property page.
4. Modify the DLL base address property.

To set /baseaddress programmatically

- See BaseAddress Property.

See Also

Visual Basic Compiler Options | /target | Sample Compilation Command Lines

/bugreport

Creates a file you can use when filing a bug report.

```
/bugreport:file
```

Arguments

file
Required. The name of the file that will contain your bug report.

Remarks

The following information is placed in *file*:

- A copy of all source code files in the compilation.
- A list of the compiler options used in the compilation.
- Version information about your compiler, common language runtime, and operating system.
- Compiler output, if any.
- A description of the problem, for which you are prompted.
- A description of how you think the problem should be fixed, for which you are prompted.

Because a copy of all source code files is placed in *file*, you may want to reproduce the (suspected) code defect in the shortest possible program.

> **Note** The **/bugreport** option is not available from within the Visual Studio development environment; it is available only when compiling from the command line. It cannot be changed programmatically.
>
> **Security Note** The **/bugreport** option produces a file that contains potentially sensitive information including: current time, compiler version, .NET Framework version, OS version, user name, the command-line arguments with which the compiler was run, all source code, and the binary form of any referenced assembly. This option can be accessed by specifying command-line options in the web.config file for a server-side compilation of an ASP.NET application. To prevent this from happening, disallow users from compiling on the server via the machine.config file.

Example

The following example compiles `t2.vb` and puts all bug reporting information in the file `problem.txt`:

```
vbc /bugreport:problem.txt t2.vb
```

See Also

Visual Basic Compiler Options | /debug | Sample Compilation Command Lines | Machine Configuration Files | <trustlevel>

/debug

Causes the compiler to generate debugging information and place it in the output file(s).

```
/debug[+ | -]
```

-or-

```
/debug:[full | pdbonly]
```

Arguments

+ | -
Optional. Specifying +, or just **/debug**, causes the compiler to generate debugging information and place it in a .pdb file. Specifying -, which is in effect if you do not specify **/debug**, causes no debug information to be created.

full | pdbonly
Optional. Specifies the type of debugging information generated by the compiler. *full*, which is in effect if you do not specify **/debug:pdbonly**, enables attaching a debugger to the running program. *pdbonly* allows source code debugging when the program is started in the debugger, but displays assembly-language code only when the running program is attached to the debugger.

Remarks

Use this option to create debug builds. If you do not specify **/debug**, **/debug+**, or **/debug:full**, you will be unable to debug the output file of your program.

By default, debugging information is not emitted (**/debug**-). To emit debugging information, specify **/debug** or **/debug+**.

For information on how to configure the debug performance of an application, see Making an Image Easier to Debug.

To set /debug in the Visual Studio integrated development environment

1. Open the project's Property Pages dialog box. For details, see <Projectname> Property Pages Dialog Box.
2. Click the Configuration Properties folder.
3. Click the Build property page.
4. Select the Generate debugging information check box.

To set /debug programmatically

- See DebugSymbols Property.

Example

The following example puts debugging information in output file `app.exe`:

```
vbc /debug /out:app.exe test.vb
```

See Also

Visual Basic Compiler Options | /bugreport | Sample Compilation Command Lines

/define

Defines conditional compiler constants.

```
/define:symbol=value[,symbol=value]
```

-or-

```
/d:symbol=value[,symbol=value]
```

Arguments

symbol
Required. The symbol to define.
value
Required. The value to assign *symbol*.

Remarks

The **/define** option has the same effect as using a **#Const** preprocessor directive in your source file.

You can use symbols created by this option with the **#If...Then...#Else** directive to compile source files conditionally.

/d is the short form of **/define**.

You can define multiple symbols with **/define** by using a comma to separate symbol definitions.

To set /define in the Visual Studio integrated development environment

1. Open the project's Property Pages dialog box. For details, see <Projectname> Property Pages Dialog Box.
2. Click the Configuration Properties folder.
3. Click the Build property page.
4. Modify the Conditional compilation constants property.

To set /define programmatically

- See DefineConstants Property.

Example

The following code defines and then uses two conditional compiler constants:

```
Vbc /define:DEBUGMODE=True,TRAPERRORS=False test.vb
Sub mysub()
   #If debugmode Then
      ' Place debug statements here.
       MsgBox("debug mode")
   #Else
      ' Place default statements here.
      '
   #End If
End Sub
```

See Also

Visual Basic Compiler Options | #If...Then...#Else Directives | #Const Directive | Sample Compilation Command Lines

/delaysign

Specifies whether the assembly will be fully or partially signed.

```
/delaysign[+ | -]
```

Arguments

+ | -
Use **/delaysign-** if you want a fully signed assembly. Use **/delaysign+** if you only want to place the public key in the assembly. The default is **/delaysign-**.

Remarks

The **/delaysign** option has no effect unless used with /keyfile or /keycontainer.

When you request a fully signed assembly, the compiler hashes the file that contains the manifest (assembly metadata) and signs that hash with the private key. The resulting digital signature is stored in the file that contains the manifest. When an assembly is delay signed, the compiler does not compute and store the signature, but reserves space in the file so the signature can be added later.

For example, using **/delaysign**+ allows a tester to put the assembly in the global cache. After testing, you can fully sign the assembly by placing the private key in the assembly.

See Creating and Using Strong-Named Assemblies for more information on signing an assembly.

This option is not available within the development environment.

To set /delaysign programmatically

- See DelaySign Property.

See Also

Visual Basic Compiler Options | /keyfile | /keycontainer | Sample Compilation Command Lines

/help, /?

Displays the compiler options.

```
/help
```

-or-

```
/?
```

Remarks

If you include this option in a compilation, no output file is created and no compilation takes place.

> **Note** The **/help** option is not available from within the Visual Studio development environment; it is available only when compiling from the command line. It cannot be changed programmatically.

Example

The following code displays help from the command line:

```
vbc /help
```

See Also

Visual Basic Compiler Options | Sample Compilation Command Lines

/imports

Imports a namespace from a specified assembly.

```
/imports:namespace[, namespace]
```

Arguments

namespace
Required. The namespace to be imported.

Remarks

The **/imports** option imports any namespace defined within the current set of source files or from any referenced assembly.

The members in a namespace specified with **/imports** are available to all source code files in the compilation. Use the Imports Statement to use a namespace in a single source code file.

To set /imports in the Visual Studio integrated development environment

1. Open the project's Property Pages dialog box. For details, see <Projectname> Property Pages Dialog Box.
2. Click the Common Properties folder.
3. Click the Imports property page.
4. Modify the Namespace property.

To set /import programmatically

- See Add Method (Imports Object).

Example

The following code compiles when **/imports:system** is specified:

```
Module MyModule
   Sub Main
      Console.WriteLine("test")
      ' Otherwise, would need
      ' System.Console.WriteLine("test")
   End Sub
End Module
```

See Also

Visual Basic Compiler Options | References and the Imports Statement | Sample Compilation Command Lines

/keycontainer

Specifies a key container name for a key pair to give an assembly a strong name.

```
/keycontainer:container
```

-or-

```
/keycontainer:"container"
```

Arguments

container
Required. Container that contains the key. Place the string in double quotation marks (" ") if it contains a space.

Remarks

The compiler creates the sharable component by inserting a public key into the assembly manifest and signing the final assembly with the private key. To generate a key file, type sn -k *file* at the command line. **sn -i** installs the key pair into a container.

If you compile with **/target:module**, the name of the key file is held in the module and incorporated into the assembly that is created when you compile an assembly with /addmodule.

You can also specify this option as a custom attribute (System.Reflection.AssemblyKeyNameAttribute) in the source code for any MSIL module.

You can also pass your encryption information to the compiler with /keyfile. Use /delaysign if you want a partially signed assembly.

See Creating and Using Strong-Named Assemblies for more information on signing an assembly.

This option is not available within the development environment.

To set /keycontainer programmatically

- See AssemblyKeyContainerName Property and the AssemblyOriginatorKeyMode Property.

Example

The following code compiles source file `input.vb` and specifies a key container:

```
vbc /keycontainer:key1 input.vb
```

See Also

Assemblies | Visual Basic Compiler Options | /keyfile | Sample Compilation Command Lines

/keyfile

Specifies a file containing a key or key pair to give an assembly a strong name.

```
/keyfile:file
```

-or-

```
/keyfile:"file"
```

Arguments

file
Required. File that contains the key. Place the string in double quotation marks (" ") if it contains a space.

Remarks

The compiler inserts the public key into the assembly manifest and then signs the final assembly with the private key. To generate a key file, type sn -k *file* at the command line.

If you compile with **/target:module**, the name of the key file is held in the module and incorporated into the assembly that is created when you compile an assembly with /addmodule.

You can also pass your encryption information to the compiler with /keycontainer. Use /delaysign if you want a partially signed assembly.

You can also specify this option as a custom attribute (System.Reflection.AssemblyKeyFileAttribute) in the source code for any MSIL module.

In case both /keyfile and /keycontainer are specified (either by command line option or by custom attribute) in the same compilation, the compiler will first try the key container. If that succeeds, then the assembly is signed with the information in the key container. If the compiler does not find the key container, it will try the file specified with /keyfile. If that succeeds, the assembly is signed with the information in the key file and the key information will be installed in the key container (similar to sn -i) so that on the next compilation, the key container will be valid.

Note that a key file might contain only the public key.

See Creating and Using Strong-Named Assemblies for more information on signing an assembly.

This option is not available within the development environment.

To set /keyfile programmatically

- See AssemblyOriginatorKeyFile Property and AssemblyOriginatorKeyMode Property.

Example

The following code compiles source file `input.vb` and specifies a key file:

```
vbc /keyfile:myfile.sn input.vb
```

See Also

Assemblies | Visual Basic Compiler Options | /target:module | /reference | Sample Compilation Command Lines

/libpath

```
/libpath:dir1[,dir2]
```

Arguments

dir1
A directory for the compiler to look in if a referenced assembly is not found in either the current working directory (the directory from which you are invoking the compiler) or the common language runtime's system directory.

dir2
Additional directories to search for assembly references. Separate additional directory names with a comma.

Remarks

The **/libpath** option specifies the location of assemblies referenced via the /reference option.

The compiler searches for assembly references that are not fully qualified in the following order:

1. Current working directory. This is the directory from which the compiler is invoked.
2. The common language runtime system directory.
3. Directories specified by **/libpath**.
4. Directories specified by the LIB environment variable.

/libpath is additive; specifying it more than once appends to any prior values.

Use **/reference** to specify an assembly reference.

To set /libpath in the Visual Studio integrated development environment

1. Open the project's Property Pages dialog box. For details, see <Projectname> Property Pages Dialog Box.
2. Click the Common Properties folder.
3. Click the Reference Path property page.
4. Modify the Folder property.

To set /libpath programmatically

- See ReferencePath Property.

Example

The following code compiles `t2.vb` to create a .exe file. The compiler looks in the working directory and in the root directory of the C drive for assembly references:

```
vbc /libpath:c:\ /reference:t2.dll t2.vb
```

See Also

Assemblies | Visual Basic Compiler Options | Sample Compilation Command Lines

/linkresource

Creates a link to a managed resource.

```
/linkresource:filename[,identifier[,public|private]]
```

-or-

```
/linkres:filename[,identifier[,public|private]]
```

Arguments

filename
Required. The resource file to link to the assembly.

identifier[,public|private]
Optional. The logical name for the resource; the name used to load the resource. The default is the name of the file. Optionally, you can specify whether the file is public or private in the assembly manifest; for example, `/linkres:filename.res,myname.res,public`. By default, *filename* is public in the assembly.

Remarks

The **/linkresource** option does not embed the resource file in the output file; use the **/resource** option to do this.

/linkresource requires one of the **/target** options other than **/target:module**.

If *filename* is a .NET Framework resource file created, for example, by the Resource File Generator (Resgen.exe) or in the development environment, it can be accessed with members in the **System.Resources** namespace (see ResourceManager Class for more information). To access all other resources at run time, use the methods that begin with **GetManifestResource** in the Assembly Class class .

filename can be any file format. For example, you may want to make a native DLL part of the assembly, so it can be installed into the Global Assembly Cache and accessed from managed code in the assembly.

The short form of **/linkresource** is **/linkres**.

> **Note** The **/linkresource** option is not available from within the Visual Studio development environment; it is available only when compiling from the command line. It cannot be changed programmatically.

Example

The following code compiles `in.vb` and links to resource file `rf.resource`:

```
vbc /linkresource:rf.resource in.vb
```

See Also

Visual Basic Compiler Options | /target | /target:module | /resource | Sample Compilation Command Lines

/main

Specifies the class or module that contains the **Sub Main** procedure.

```
/main:location
```

Arguments

location
Required. A full qualification to the **Sub Main** procedure to be called when the program starts. This may be in the form **/main:module** or **/main:namespace.module**. *location* also can be a class that inherits from **System.Windows.Forms.Form**.

Remarks

Use this option when creating an executable file or Windows executable program. If the **/main** option is omitted, the compiler searches for a valid shared **Sub Main** in all public classes and modules.

See The Visual Basic Version of Hello World! for a discussion of the various forms of the **Main** procedure.

When *location* is a class that inherits from **System.Windows.Forms.Form**, vbc.exe lets you compile code at the command line that was created in the development environment:

```
' Compile with /r:System.dll,SYSTEM.WINDOWS.FORMS.DLL /main:MyC.
Public Class MyC
   Inherits System.Windows.Forms.Form
End Class
```

To set /main in the Visual Studio integrated development environment

1. Open the project's Property Pages dialog box. For details, see <Projectname> Property Pages Dialog Box.
2. Click the Common Properties folder.
3. Click the General property page.
4. Modify the Startup object property.

To set /main programmatically

- See StartupObject Property.

Example

The following code compiles `t2.vb` and `t3.vb`, specifying that the **Sub Main** procedure will be found in the `Test2` class:

```
vbc t2.vb t3.vb /main:Test2
```

See Also

Visual Basic Compiler Options | /target:winexe | /target:exe | Sample Compilation Command Lines

/netcf

Sets the compiler to target the .NET Compact Framework.

```
/netcf
```

Remarks

/netcf causes the Visual Basic compiler to target the .NET Compact Framework rather than the full .NET Framework. Language functionality that is present only in the full .NET Framework is disabled.

/netcf is designed to be used with /sdkpath. The language features disabled by **/netcf** are the same language features not present in the files targeted with **/sdkpath**.

> **Note** **/netcf** is not available from within the Visual Studio .NET integrated development environment (IDE); it is available only when compiling from the command line. It cannot be changed programmatically. **/netcf** is set when a Visual Basic .NET device project is loaded.

/netcf disables the following features:

- The End keyword to terminate execution of a program:

```
Module Module1
Sub Main()
   End   ' not valid to terminate execution with /netcf
End Sub
End Module
```

- Late binding, in all forms, is disabled. Compile-time errors are generated when recognized late-binding scenarios are encountered. The following program will compile and run without **/netcf** but will fail at compile time with **/netcf**.

```
Class LateBoundClass
   Sub S1()
   End Sub

   Default Property P1(ByVal s As String) As Integer
      Get
      End Get
      Set(ByVal Value As Integer)
      End Set
   End Property
End Class

Module Module1
   Sub Main()
      Dim o1 As Object
      Dim o2 As Object
      Dim o3 As Object
      Dim IntArr(3) As Integer

      o1 = New LateBoundClass
      o2 = 1
      o3 = IntArr

      ' Latebound calls
      o1.S1()
      o1.P1("member") = 1

      ' Dictionary member access
```

```
        o1!member = 1

        ' Latebound overload resolution
        LateBoundSub(o2)

        ' Latebound array
        o3(1) = 1
    End Sub

    Sub LateBoundSub(ByVal n As Integer)
    End Sub

    Sub LateBoundSub(ByVal s As String)
    End Sub
End Module
```

- The Auto, Ansi, and Unicode modifiers are disabled. The syntax of the Declare statement is also modified to: **Declare Sub|Function** *name* **Lib** "*library*" [**Alias** "*alias*"] [([*arglist*])]. The following code sample shows the effect of **/netcf** on a compilation:

```
' compile with: /target:library
Module Module1
   ' valid with or without /netcf
   Declare Sub DllSub Lib "SomeLib.dll" ()

   ' invalid with /netcf
   Declare Auto Sub DllSub1 Lib "SomeLib.dll" ()
   Declare Ansi Sub DllSub2 Lib "SomeLib.dll" ()
   Declare Unicode Sub DllSub3 Lib "SomeLib.dll" ()
End Module
```

- The use of Visual Basic 6.0 keywords that were removed from Visual Basic .NET generate a different error when **/netcf** is used. This affects the error messages seen for the following keywords:
  - OPEN
  - CLOSE
  - GET
  - PUT
  - PRINT
  - WRITE
  - INPUT
  - LOCK
  - UNLOCK
  - SEEK
  - WIDTH
  - NAME
  - FREEFILE
  - EOF
  - LOC
  - LOF
  - LINE
  - GET

Example

The following code compiles `myfile.vb` with the .NET Compact Framework:

```
vbc /netcf myfile.vb
```

See Also

Visual Basic Compiler Options | Sample Compilation Command Lines

/nologo

Suppresses display of the copyright banner and informational messages during compilation.

```
/nologo
```

Remarks

If you specify **/nologo**, the compiler does not display a copyright banner. By default, **/nologo** is not in effect.

> **Note** The **/nologo** option is not available from within the Visual Studio development environment; it is available only when compiling from the command line. It cannot be changed programmatically.

Example

The following code compiles `t2.vb` and does not display a copyright banner:

```
vbc /nologo t2.vb
```

See Also

Visual Basic Compiler Options | Sample Compilation Command Lines

/nowarn

Suppresses the compiler's ability to generate warnings.

```
/nowarn
```

To set /nowarn in the Visual Studio integrated development environment

1. Open the project's Property Pages dialog box. For details, see <Projectname> Property Pages Dialog Box.
2. Click the Configuration Properties folder.
3. Click the Build property page.
4. Modify the Enable build warnings property.

To set /nowarn programmatically

- See WarningLevel Property.

Example

The following code compiles `t2.vb` and does not display warnings:

```
vbc /nowarn t2.vb
```

See Also

Visual Basic Compiler Options | Sample Compilation Command Lines

/optimize

Enables or disables compiler optimizations.

```
/optimize[ + | - ]
```

Arguments

+ | -
/optimize- disables compiler optimizations. **/optimize+** enables optimizations. By default, optimizations are disabled.

Remarks

Compiler optimizations make your output file smaller, faster, and more efficient. However, because optimizations result in code rearrangement in the output file, **/optimize+** can make debugging difficult.

All modules used by an assembly must have been built using the same /optimize settings as the assembly's /optimize setting.

You can combine the **/optimize** and **/debug** options.

To set /baseaddress in the Visual Studio integrated development environment

1. Open the project's Property Pages dialog box. For details, see <Projectname> Property Pages Dialog Box.
2. Click the Configuration Properties folder.
3. Click the Optimizations property page.
4. Modify the Enable optimizations property.

To set /optimize programmatically

- See Optimize Property.

Example

The following code compiles `t2.vb` and enables compiler optimizations:

```
vbc t2.vb /optimize
```

See Also

Visual Basic Compiler Options | /debug | Sample Compilation Command Lines

/optioncompare

Specifies how string comparisons are made.

```
/optioncompare:[binary | text]
```

Remarks

You can specify **/optioncompare** in one of two forms: /optioncompare:binary to use binary string comparisons, and /optioncompare:text to use text string comparisons.

/optioncompare:binary is the default.

See Also

Visual Basic Compiler Options | /optionexplicit | /optionstrict | Sample Compilation Command Lines

/optioncompare:binary

Causes strings to be compared based on their binary representations.

```
/optioncompare:binary
```

Remarks

In Microsoft Windows, the code page being used determines the binary sort order. A typical binary sort order is shown in the following example:

```
A < B < E < Z < a < b < e < z < À < Ê < Ø < à < ê < ø
```

To set /optioncompare:binary in the Visual Studio integrated development environment

1. Open the project's Property Pages dialog box. For details, see <Projectname> Property Pages Dialog Box.
2. Click the Common Properties folder.
3. Click the Build property page.
4. Modify the Option Compare property.

To set /optioncompare:binary programmatically

- See OptionCompare Property.

Example

The following code compiles `myfile.vb` and uses binary string comparisons:

```
vbc /optioncompare:binary myfile.vb
```

See Also

Visual Basic Compiler Options | /optioncompare:text | /optionstrict | /optionexplicit | Sample Compilation Command Lines

/optioncompare:text

Causes strings to be compared based on their text representations.

```
/optioncompare:text
```

Remarks

Text-based string comparisons are based on a case-insensitive text sort order determined by your system's locale. A typical text sort order is shown in the following example:

```
(A=a) < (À = à) < (B=b) < (E=e) < (Ê = ê) < (Z=z) < (Ø = ø)
```

To set /optioncompare:text in the Visual Studio integrated development environment

1. Open the project's Property Pages dialog box. For details, see <Projectname> Property Pages Dialog Box.
2. Click the Common Properties folder.
3. Click the Build property page.
4. Modify the Option Compare property.

To set /optioncompare:text programmatically

- See OptionCompare Property.

Example

The following code compiles `myfile.vb` and uses text-based string comparisons:

```
vbc /optioncompare:text myfile.vb
```

See Also

Visual Basic Compiler Options | /optioncompare:binary | /optionstrict | /optionexplicit | Sample Compilation Command Lines

/optionexplicit

Causes the compiler to report errors if variables are not declared prior to use.

```
/optionexplicit[+ | -]
```

Arguments

+ | -
Specify **/optionexplicit+** to require explicit declaration of variables. **/optionexplicit+**, which is the same as **/optionexplicit**, is the default. **/optionexplicit-** allows implicit declaration of variables.

To set /optionexplicit in the Visual Studio integrated development environment

1. Open the project's Property Pages dialog box. For details, see <Projectname> Property Pages Dialog Box.
2. Click the Common Properties folder.
3. Click the Build property page.
4. Modify the Option Explicit property.

To set /optionexplicit programmatically

- See OptionExplicit Property.

Example

The following code compiles when **/optionexplicit-** is used:

```
Module Module1
   Sub Main()
      i = 99
      System.Console.WriteLine(i)
   End Sub
End Module
```

See Also

Visual Basic Compiler Options | /optionstrict | Sample Compilation Command Lines | Option Explicit Statement

/optionstrict

Enforces strict type semantics to restrict implicit type conversions.

```
/optionstrict[+ | -]
```

Arguments

+ | -
/optionstrict+ restricts implicit type conversion. The default for this option is **/optionstrict-**; **/optionstrict-** is the same as **/optionstrict**. Both let you use permissive type semantics.

Remarks

When **/optionstrict+** is in effect, only widening type conversions can be done implicitly. Implicit narrowing type conversions, such as assigning a **Decimal** type object to an integer type object, are reported as errors.

To set /optionstrict in the Visual Studio integrated development environment

1. Open the project's Property Pages dialog box. For details, see <Projectname> Property Pages Dialog Box.
2. Click the Common Properties folder.
3. Click the Build property page.
4. Modify the Option Strict property.

To set /optionstrict programmatically

- See OptionStrict Property.

Example

The following code compiles `test.vb` using strict type semantics:

```
vbc /optionstrict+ test.vb
```

See Also

Visual Basic Compiler Options | /optionexplicit | Sample Compilation Command Lines | Option Strict Statement

/out

Specifies the name of the output file.

```
/out:filename
```

Arguments

filename
Required. The name of the output file the compiler creates.

Remarks

The compiler expects to find one or more source code files following the **/out** option.

Specify the full name and extension of the file to create. If you do not, the .exe file takes its name from the source code file containing the **Sub Main** procedure, and the .dll file takes its name from the first source code file.

To set /out in the Visual Studio integrated development environment

1. Open the project's Property Pages dialog box. For details, see <Projectname> Property Pages Dialog Box.
2. Click the Common Properties folder.
3. Click the General property page.
4. Modify the Assembly name property.

To set /out programmatically

- See OutputFileName Property.

Example

The following code compiles `t2.vb` and creates output file `t2.exe`:

```
vbc t2.vb /out:t3.exe
```

See Also

Visual Basic Compiler Options | /target | Sample Compilation Command Lines

/quiet

Prevents the compiler from displaying code for syntax-related errors and warnings.

```
/quiet
```

Remarks

By default, **/quiet** is not in effect. When the compiler reports a syntax-related error or warning, it also outputs the line from source code. For applications that parse compiler output, it may be more convenient for the compiler to output only the text of the diagnostic.

In the following example, `Module1` outputs an error that includes source code when compiled without **/quiet**:

```
Module Module1
   Sub Main()
      x
   End Sub
End Module
```

Output:

```
E:\test\t2.vb(3) : error BC30451: Name 'x' is not declared.

        x
        ~
```

Compiled with **/quiet**, the compiler outputs only the following:

```
E:\test\t2.vb(3) : error BC30451: Name 'x' is not declared.
```

> **Note** The **/quiet** option is not available from within the Visual Studio development environment; it is available only when compiling from the command line. It cannot be accessed programmatically.

Example

The following code compiles `t2.vb` and does not display code for syntax-related compiler diagnostics:

```
vbc /quiet t2.vb
```

See Also

Visual Basic Compiler Options | Sample Compilation Command Lines

/recurse

Compiles source code files in all child directories of either the specified directory or the project directory.

```
/recurse:[dir\]file
```

Arguments

dir
Optional. The directory in which you want the search to begin. If not specified, the search begins in the project directory.
file
The file(s) to search for. Wildcard characters are allowed.

Remarks

You can use wildcards in a file name to compile all matching files in the project directory without using **/recurse**. If no output file name is specified, the compiler bases the output file name on the first input file processed. This is generally the first file in the list of files compiled when viewed alphabetically. For this reason, it is best to specify an output file using the **/out** option.

> **Note** The **/recurse** option is not available from within the Visual Studio development environment; it is available only when compiling from the command line. It cannot be changed programmatically.

Example

The following code compiles all Visual Basic files in the current directory:

```
vbc *.vb
```

The following code compiles all Visual Basic files in the Test\ABC directory and any directories below it, and then generates Test.ABC.dll:

```
vbc /target:library /out:Test.ABC.dll /recurse:Test\ABC\*.vb
```

See Also

Visual Basic Compiler Options | /out | Sample Compilation Command Lines

/reference

Causes the compiler to make type information in the specified assemblies available to the project you are currently compiling.

```
/reference:file[,file2]
```

-or-

```
/r:file[,file2]
```

Arguments

file, *file2*
One or more assemblies. To import more than one file, separate file names with a comma.

Remarks

The file(s) you import must contain assembly metadata. Only public types are visible outside the assembly. /addmodule imports metadata from a module.

At run time, you should anticipate that only one .exe assembly can be loaded per process, even though there may be times when more than one .exe might be loaded in the same process. Therefore, it is not recommended to pass an assembly built with **/target:exe** or **/target:winexe** to **/reference** if you are compiling with **/target:winexe** or **/target:exe**.

If you reference an assembly (Assembly A) which itself references another assembly (Assembly B), you need to reference Assembly B if:

- A type you use from Assembly A inherits from a type or implements an interface from Assembly B.
- You invoke a field, property, event, or method that has a return type or parameter type from Assembly B.

Use /libpath to specify the directory in which one or more of your assembly references is located.

For the compiler to recognize a type in an assembly (not a module), it must be forced to resolve the type, which you can do, for example, by defining an instance of the type. Other ways are available to resolve type names in an assembly for the compiler. For example, if you inherit from a type in an assembly, the type name then becomes known to the compiler.

/r is the short form of **/reference**.

To set /reference in the Visual Studio integrated development environment

- See the Add Reference Dialog Box.

To set /reference programmatically

- See ReferencePath Property and Add Method (References Object).

Example

The following code compiles source file `input.vb` and reference assemblies from `metad1.dll` and `metad2.dll` to produce `out.exe`:

```
vbc /reference:metad1.dll,metad2.dll /out:out.exe input.vb
```

See Also

Visual Basic Compiler Options | /target | /target:module | Public | Sample Compilation Command Lines

/removeintchecks

Turns overflow error checking for integer operations on or off.

```
/removeintchecks[+ | -]
```

Arguments

+ | -

/removeintchecks- causes the compiler to check all integer calculations for errors such as overflow or division by zero. **/removeintchecks-** is the default for this option.

Specifying **/removeintchecks** or **/removeintchecks+** prevents error checking and can make integer calculations faster. However, without error checking and if data type capacities are overflowed, incorrect results may be stored without raising an error.

To set /removeintchecks in the Visual Studio integrated development environment

1. Open the project's Property Pages dialog box. For details, see <Projectname> Property Pages Dialog Box.
2. Click the Configuration Properties folder.
3. Click the Optimizations property page.
4. Modify the Remove integer overflow checks property.

To set /removeintchecks programmatically

- See RemoveIntegerChecks Property.

Example

The following code compiles `test.vb` and turns off integer overflow error checking:

```
vbc /removeintchecks+ test.vb
```

See Also

Visual Basic Compiler Options | Sample Compilation Command Lines

/resource

Embeds a managed resource in an assembly.

```
/resource:filename[,identifier[,public|private]]
```

-or-

```
/res:filename[,identifier[,public|private]]
```

Arguments

filename
The resource file to embed in the output file. By default, *filename* is public in the assembly.

identifier[,public|private]
Optional. The logical name for the resource; the name used to load it. The default is the name of the file. Optionally, you can specify whether the resource is public or private in the assembly manifest; for example:

```
/res:filename.res,myname.res,public.
```

Remarks

Use **/resource** to link a resource to an assembly without placing the resource file in the output file.

If *filename* is a .NET Framework resource file created, for example, by the Resource File Generator (Resgen.exe) or in the development environment, it can be accessed with members in the **System.Resources** namespace (see ResourceManager Class for more information). For all other resources, use the **GetManifestResource** methods in **System.Reflection.Assembly** class to access the resource at run time.

/res is the short form of **/resource**.

To set /resource in the Visual Studio integrated development environment

1. Add a resource file to your project.
2. Select the file to embed in Solution Explorer.
3. Select Build Action for the file in the Properties Window.
4. Set Build Action to Embedded Resource.

To set /resource programmatically

- See BuildAction Property (Visual Basic and C# ProjectItem Object).

Example

The following code compiles `in.vb` and attaches resource file `rf.resource`:

```
vbc /res:rf.resource in.vb
```

See Also

Visual Basic Compiler Options | /win32resource | /linkresource | /target | /target:module | Sample Compilation Command Lines

/rootnamespace

Specifies a namespace for all type declarations.

```
/rootnamespace:namespace
```

Arguments

namespace
The name of the namespace in which to enclose all type declarations for the current project.

Remarks

If you use the Visual Studio executable (Devenv) to compile a project created in the Visual Studio integrated development environment, **/rootnamespace** lets you specify the value of the **RootNamespace** property. See Devenv Command Line Switches for more information.

Use the common language runtime MSIL Disassembler (ildasm.exe) to view the namespace names in your output file.

To set /rootnamespace in the Visual Studio integrated development environment

1. Open the project's Property Pages dialog box. For details, see <Projectname> Property Pages Dialog Box.
2. Click the Common Properties folder.
3. Click the General property page.
4. Modify the Root Namespace property.

To set /rootnamespace programmatically

- See DefaultNamespace Property (Visual Basic and C# Project Object) and DefaultNamespace Property (Visual Basic and Visual C# ProjectItem Object).

Example

The following code compiles `in.vb` and encloses all type declarations in the namespace `mynamespace`:

```
vbc /rootnamespace:mynamespace in.vb
```

See Also

Visual Basic Compiler Options | MSIL Disassembler (Ildasm.exe) | Sample Compilation Command Lines

/sdkpath

Specifies the location of mscorlib.dll and microsoft.visualbasic.dll.

```
/sdkpath path
```

Arguments

path
The directory containing the versions of mscorlib.dll and microsoft.visualbasic.dll to use for compilation. This path is not verified until it is loaded.

Remarks

This option tells the Visual Basic compiler to load the mscorlib.dll and microsoft.visualbasic.dll files from a non-default location. **/sdkpath** was designed to be used with /netcf. The .NET Compact Framework uses different versions of these support libraries to avoid the use of types and language features not found on the devices.

> **Note** **/sdkpath** is not available from within the Visual Studio .NET integrated development environment (IDE); it is available only when compiling from the command line. It cannot be changed programmatically. **/sdkpath** is set when a Visual Basic .NET device project is loaded.

Example

The following code compiles `myfile.vb` using the versions of mscorlib.dll and microsoft.visualbasic.dll found in the default installation directory of the .NET Compact Framework on the C drive:

```
vbc /sdkpath:"c:\Program Files\Microsoft Visual Studio .NET 2003\CompactFrameworkSDK\v1.0.500
0\Windows CE " myfile.vb
```

See Also

Visual Basic Compiler Options | Sample Compilation Command Lines

/target

Specifies the format of compiler output.

```
/target:[exe | library | module | winexe]
```

Remarks

Use /target:exe to create a .exe console application. This is the default if you do not specify the **/target** option.

Use /target:library to create a dynamic-link library (DLL).

Use /target:module to create a module.

Use /target:winexe to create a Windows program.

Unless you specify **/target:module**, **/target** causes a .NET Framework assembly manifest to be placed in an output file.

Each invocation of vbc.exe produces, at most, one output file. If you specify a compiler option such as **/out** or **/target** more than once, the last one the compiler sees is put into effect. Information about all files in a compilation is placed in the manifest. All output files except those created with **/target:module** contain assembly metadata in the manifest. Use MSIL Disassembler (Ildasm.exe) to view the metadata in an output file.

Example

If you create an assembly, you can indicate that all or part of your code is CLS-compliant with the CLSCompliantAttribute Class attribute, as in the following code:

```
<Assembly: System.CLSCompliant(True)>
<System.CLSCompliant(False)> _
Public Class AClass
End Class

Module Module1
   Sub Main()
   End Sub
End Module
```

See Also

Visual Basic Compiler Options | Assemblies | Sample Compilation Command Lines

/target:exe

Causes the compiler to create an executable console application.

```
/target:exe
```

-or-

```
/t:exe
```

Remarks

The **/target:exe** option is the default when no **/target** option is specified. The executable file is created with a .exe extension.

Use /target:winexe to create a Windows program executable.

Unless otherwise specified with the **/out** option, the output file name takes the name of the input file that contains the **Sub Main** procedure.

Only one **Sub Main** procedure is required in the source code files that are compiled into a .exe file. The **/main** compiler option lets you specify which class contains the **Sub Main** procedure.

/t:exe is the short form of **/target:exe**.

To set /target:exe in the Visual Studio integrated development environment

1. Open the project's Property Pages dialog box. For details, see <Projectname> Property Pages Dialog Box.
2. Click the Common Properties folder.
3. Click the General property page.
4. Modify the Output type property.

To set /target:exe programmatically

- See OutputType Property.

Example

Both of the following two command lines compile `in.vb`, creating a console application called `in.exe`:

```
vbc /target:exe in.vb
vbc in.vb
```

See Also

/main | /target | /target:library | /target:module | /target:winexe | Visual Basic Compiler Options | /out | Sample Compilation Command Lines

/target:library

Causes the compiler to create a dynamic-link library (DLL) rather than an executable (EXE) file.

```
/target:library
```

-or-

```
/t:library
```

Remarks

The dynamic-link library file is created with a .dll extension.

Unless otherwise specified with the **/out** option, the output file name takes the name of the first input file.

When building a DLL, a **Sub Main** procedure is not required.

/t:library is the short form of **/target:library**.

To set /target:library in the Visual Studio integrated development environment

1. Open the project's Property Pages dialog box. For details, see <Projectname> Property Pages Dialog Box.
2. Click the Common Properties folder.
3. Click the General property page.
4. Modify the Output type property.

To set /target:library programmatically

- See OutputType Property.

Example

The following code compiles `in.vb`, creating `in.dll`:

```
vbc /target:library in.vb
```

See Also

/target | /target:exe | /target:module | /target:winexe | Visual Basic Compiler Options | /out | Sample Compilation Command Lines

/target:module

Causes the compiler to generate a module that can be added to an assembly.

```
/target:module
```

-or-

```
/t:module
```

Remarks

The output file is created with an extension of .netmodule.

The .NET common language runtime cannot load a file that does not have an assembly. However, you can incorporate such a file into the assembly manifest of an assembly by using **/reference**.

When code in one module references internal types in another module, both modules must be incorporated into an assembly manifest by using **/reference**.

/addmodule imports metadata from a module.

The short form of **/target:module** is **/t:module**.

> **Note** The **/target:module** option is not available from within the Visual Studio development environment; it is available only when compiling from the command line. It cannot be changed programmatically.

Example

The following code compiles `in.vb`, creating `in.netmodule`:

```
vbc /target:module in.vb
```

See Also

/reference | /target | /target:exe | /target:library | /target:winexe | Visual Basic Compiler Options | Sample Compilation Command Lines

/target:winexe

Causes the compiler to create an executable Windows program.

```
/target:winexe
```

-or-

```
/t:winexe
```

Remarks

The executable file is created with a .exe extension. A Windows program is one that provides a user interface from either the Services Framework class library or with the Win32 APIs.

Use /target:exe to create a console application.

Unless otherwise specified with the **/out** option, the output file name takes the name of the input file that contains the **Sub Main** procedure.

Only one **Sub Main** procedure is required in the source code files that are compiled into a .exe file. The **/main** compiler option lets you specify which class contains the **Sub Main** procedure in cases where your code has more than one class with a **Sub Main** procedure.

The short form of **/target:winexe** is **/t:winexe**.

To set /target:winexe in the Visual Studio integrated development environment

1. Open the project's Property Pages dialog box. For details, see <Projectname> Property Pages Dialog Box.
2. Click the Common Properties folder.
3. Click the General property page.
4. Modify the Output type property.

To set /target:winexe programmatically

- See OutputType Property.

Example

The following code compiles `in.vb` into a Windows program:

```
vbc /target:winexe in.vb
```

See Also

/main | /target | /target:exe | /target:library | /target:module | /out | Visual Basic Compiler Options | Sample Compilation Command Lines

/utf8output

Displays compiler output using UTF-8 encoding.

```
/utf8output[+ | -]
```

Arguments

+ | -
The default for this option is **/utf8output-**, which means compiler output does not use UTF-8 encoding. Specifying **/utf8output** is the same as specifying **/utf8output+**.

Remarks

In some international configurations, compiler output cannot be displayed correctly in the console. In such situations, use **/utf8output** and redirect compiler output to a file.

> **Note** The **/utf8output** option is not available from within the Visual Studio development environment; it is available only when compiling from the command line. It cannot be changed programmatically.

Example

The following code compiles `in.vb` and directs the compiler to display output using UTF-8 encoding:

```
vbc /utf8output in.vb
```

See Also

Visual Basic Compiler Options | Sample Compilation Command Lines

/verbose

Causes the compiler to produce verbose status and error messages.

```
/verbose[+ | -]
```

Arguments

+ | -
Specifying **/verbose** is the same as specifying **/verbose+**, which causes the compiler to emit verbose messages. The default for this option is **/verbose-**.

Remarks

/verbose displays information about the total number of errors issued by the compiler, reports which assemblies are being loaded by a module, and displays which files are currently being compiled.

> **Note** The **/verbose** option is not available from within the Visual Studio development environment; it is available only when compiling from the command line. It cannot be changed programmatically.

Example

The following code compiles `in.vb` and directs the compiler to display verbose status information:

```
vbc /verbose in.vb
```

See Also

Visual Basic Compiler Options | Sample Compilation Command Lines

/warnaserror

Causes the compiler to treat the first occurrence of a warning as an error.

```
/warnaserror[+ | -]
```

Arguments

+ | -
By default, **/warnaserror-** is in effect; warnings do not prevent the compiler from producing an output file. **/warnaserror**, which is the same as **/warnaserror+**, causes warnings to be treated as errors.

Remarks

Subsequent occurrences of the same warning are reported as warnings. The compiler produces no output file.

To set /warnaserror in the Visual Studio integrated development environment

1. Open the project's Property Pages dialog box. For details, see <Projectname> Property Pages Dialog Box.
2. Click the Configuration Properties folder.
3. Click the Build property page.
4. Modify the Treat compiler warnings as errors property.

To set /warnaserror programmatically

- See TreatWarningsAsErrors Property.

Example

The following code compiles `in.vb` and directs the compiler to display an error for the first occurrence of every warning it finds:

```
vbc /warnaserror in.vb
```

See Also

Visual Basic Compiler Options | Sample Compilation Command Lines

/win32icon

Inserts an .ico file in the output file. This .ico file represents the output file in the Windows Explorer.

```
/win32icon:filename
```

Arguments

filename
The .ico file to add to your output file.

Remarks

You can create an .ico file with the Microsoft® Windows® Resource Compiler (RC). The Resource Compiler is invoked when you compile a Visual C++ program; an .ico file is created from the .rc file. The **/win32icon** and **/win32resource** options are mutually exclusive.

See /linkresource to reference a .NET Framework resource file, or /resource to attach a .NET Framework resource file. See /win32resource to import a .res file.

To set /win32icon in the Visual Studio integrated development environment

1. Open the project's Property Pages dialog box. For details, see <Projectname> Property Pages Dialog Box.
2. Click the Common Properties folder.
3. Click the Build property page.
4. Modify the Application icon property.

To set /win32icon programmatically

- See ApplicationIcon Property.

Example

The following code compiles `in.vb` and attaches an .ico file, `rf.ico`:

```
vbc /win32icon:rf.ico in.vb
```

See Also

Visual Basic Compiler Options | Sample Compilation Command Lines

/win32resource

Inserts a Win32 resource file in the output file.

```
/win32resource:filename
```

Arguments

filename
The resource file to add to your output file.

Remarks

You can create a Win32 resource file with the Microsoft Windows Resource Compiler (RC).

A Win32 resource can contain version or bitmap (icon) information that helps identify your application in the Windows Explorer. If you do not specify **/win32resource**, the compiler generates version information based on the assembly version. The **/win32resource** and **/win32icon** options are mutually exclusive.

See /linkresource to reference a .NET Framework resource file, or /resource to attach a .NET Framework resource file.

> **Note** The **/nologo** option is not available from within the Visual Studio development environment; it is available only when compiling from the command line. It cannot be changed programmatically.

Example

The following code compiles `in.vb` and attaches a Win32 resource file, `rf.res`:

```
vbc /win32resource:rf.res in.vb
```

See Also

Visual Basic Compiler Options | Sample Compilation Command Lines

Visual Basic Compiler Options Listed by Category

The Visual Basic .NET command-line compiler is provided as an alternative to compiling programs from within the Visual Studio .NET integrated development environment (IDE). The following is a list of the Visual Basic command-line compiler options sorted by functional category.

Compiler Output

Option	Purpose
/nologo	Suppresses compiler banner information.
/utf8output	Displays compiler output using UTF-8 encoding.
/verbose	Outputs extra information during compilation.

Optimization

Option	Purpose
/optimize	Enables/disables optimizations.

Output Files

Option	Purpose
/netcf	Sets the compiler to target the .NET Compact Framework.
/out	Specifies an output file.
/target	Specifies the format of the output file using one of four options: /target:exe, /target:library, /target:module, or /target:winexe.

.NET Assemblies

Option	Purpose
/addmodule	Causes the compiler to make all type information from the specified file(s) available to the project you are currently compiling.
/delaysign	Specifies whether the assembly will be fully or partially signed.
/imports	Imports a namespace from a specified assembly.
/keycontainer	Specifies a key container name for a key pair to give an assembly a strong name.
/keyfile	Specifies a file containing a key or key pair to give an assembly a strong name.
/libpath	Specifies the location of assemblies referenced via the /reference option.
/reference	Imports metadata from an assembly.

Debugging/Error Checking

Option	Purpose
/bugreport	Creates a file that contains information that makes it easy to report a bug.
/debug	Produces debugging information.
/nowarn	Suppresses the compiler's ability to generate warnings.
/quiet	Prevents the compiler from displaying code for syntax-related errors and warnings.
/removeintchecks	Disables integer overflow checking.
/warnaserror	Promotes warnings to errors.

Help

Option	Purpose
/?	Displays the compiler options. This command is the same as specifying the **/help** option. No compilation occurs.
/help	Displays the compiler options. This command is the same as specifying the **/?** option. No compilation occurs.

Language

Option	Purpose

/optionexplicit	Requires explicit declaration of variables.
/optionstrict	Enforces strict type semantics.
/optioncompare	Determines whether string comparisons should be binary or use locale-specific text semantics.

Preprocessor

Option	Purpose
/define	Defines symbols for conditional compilation.

Resources

Option	Purpose
/linkresource	Creates a link to a managed resource.
/resource	Embeds a managed resource in an assembly.
/win32icon	Inserts an .ico file into the output file.
/win32resource	Inserts a Win32 resource into the output file.

Miscellaneous

Option	Purpose
@ (Specify Response File)	Specifies a response file.
/baseaddress	Specifies the base address of a DLL.
/main	Specifies the class that contains the **Sub Main** procedure to use at startup.
/recurse	Searches subdirectories for source files to compile.
/rootnamespace	Specifies a namespace for all type declarations.
/sdkpath	Specifies the location of mscorlib.dll and microsoft.visualbasic.dll.

See Also

Visual Basic Compiler Options Listed Alphabetically | <Projectname> Property Pages Dialog Box

Visual Basic .NET Language Specification

This document describes the Visual Basic .NET language. It is designed to be a complete language description, not a conceptual overview or a user's reference manual. For a conceptual overview, see *Visual Basic Language Features*. For language reference information, see *Visual Basic Language Reference*.

In This Section

1. Introduction
   Describes the evolution of Visual Basic and its role within .NET.
2. Lexical Grammar
   Discusses the syntax of characters, lines, identifiers, keywords, literals, and separators.
3. Preprocessing Directives
   Summarizes the role of conditional compilation, external source directives, and region directives.
4. General Concepts
   Provides an overview of the core concepts of object-oriented programming in Visual Basic: declaration spaces, inheritance, implementation, polymorphism, accessibility, scope, namespace and type names, and attributes.
5. Attributes
   Summarizes the syntax of attributes, which modify entities when being declared.
6. Source Files and Namespaces
   Explains compilation options, the **Imports** statement, and namespaces.
7. Types
   Provides an overview of Visual Basic .NET as a type-safe system: the taxonomy of value and reference types.
8. Conversions
   Discusses the grammar of changing value types: implicit and explicit conversions, widening and narrowing conversions, value and reference conversions, and cast expressions.
9. Type Members
   Discusses methods, constructors, events, constants, variables, and properties.
10. Statements
   Summarizes the syntax of executable code in Visual Basic .NET.
11. Expressions
   Summarizes the syntax of computational processes in Visual Basic .NET.
12. Visual Basic .NET Grammar Summary
   Provides an overview of the entire Visual Basic .NET grammar.

Related Sections

Visual Basic Language and Run-Time Reference

Visual Basic Language Features

1. Introduction

The Microsoft® Visual Basic® .NET programming language is a high-level programming language for the Microsoft .NET Framework. Although it is designed to be an approachable and easy-to-learn language, it is also powerful enough to satisfy the needs of experienced programmers. The Visual Basic .NET programming language is closely related to the Visual Basic programming language but the two languages are not the same. A discussion of the differences between Visual Basic .NET and Visual Basic 6.0 is beyond the scope of this document.

The Visual Basic .NET programming language has a syntax that is similar to English, which promotes the clarity and readability of Visual Basic .NET code. Wherever possible, meaningful words or phrases are used instead of abbreviations, acronyms, or special characters. Extraneous or unneeded syntax is generally allowed but not required.

The Visual Basic .NET programming language can be either a strongly typed or a loosely typed language. Loose typing defers much of the burden of type checking until a program is already running. This includes not only type checking of conversions but also of method calls, meaning that the binding of a method call can be deferred until run-time. This is useful when building prototypes or other programs in which speed of development is more important than execution speed. The Visual Basic .NET programming language also provides strongly typed semantics that performs all type checking at compile-time and disallows run-time binding of method calls. This guarantees maximum performance and helps ensure that type conversions are correct. This is useful when building production applications in which speed of execution and execution correctness is important.

This document describes the Visual Basic .NET language. It is meant to be a complete language description rather than a language tutorial or a user's reference manual.

See Also

1.1 Grammar Notation

1.1 Grammar Notation

This specification describes two grammars: a lexical grammar and a syntactic grammar. The lexical grammar defines how characters can be combined to form tokens; the syntactic grammar defines how the tokens can be combined to form Visual Basic .NET programs. There are also several secondary grammars used for preprocessing operations like conditional compilation.

> **Note** The grammars in this specification are designed to be human readable, not formal (that is, usable by LEX or YACC).

All of the grammars use a modified BNF notation, which **consists** of a set of productions made up of terminal and non-terminal names. A terminal name represents one or more Unicode characters. Each nonterminal name is defined by one or more productions. In a production, nonterminal names are shown in *italic type*, and terminal names are shown in a **bold type**. Text in normal type and surrounded by angle-bracket metasymbols are informal terminals (for example, "< all Unicode characters >"). Each grammar starts with the nonterminal *Start*.

Case is unimportant in Visual Basic .NET programs. For simplicity, all terminals will be given in standard casing, but any casing will match them. Terminals that are printable elements of the ASCII character set are represented by their corresponding ASCII characters. Visual Basic .NET is also width insensitive when matching terminals, allowing full-width Unicode characters to match their half-width Unicode equivalents.

A set of productions begins with the name of a nonterminal, followed by two colons and an equal sign. The right side contains a terminal or nonterminal production. A nonterminal may have multiple productions that are separated by the vertical-bar metasymbol (|). Items included in square-bracket metasymbols ([]) are optional. A plus metasymbol (+) following an item means the item may occur one or more times.

Line breaks and indentation may be added for readability and are not part of the production.

See Also

1. Introduction

2. Lexical Grammar

Compilation of a Visual Basic .NET program first involves translating the raw stream of Unicode characters into an ordered set of lexical tokens. Because the Visual Basic .NET language is not free-format, the set of tokens is then further divided into a series of logical lines. A *logical line* spans from either the start of the stream or a line terminator through to the next line terminator that is not preceded by a line continuation or through to the end of the stream.

```
Start ::= [ LogicalLine+ ]
LogicalLine ::= [ LogicalLineElement+ ] [ Comment ] LineTerminator
LogicalLineElement ::= WhiteSpace | LineContinuation | Token
Token ::= Identifier | Keyword | Literal | Separator | Operator
```

See Also

2.1 Characters and Lines | 2.2 Identifiers | 2.3 Keywords | 2.4 Literals | 2.5 Separators | 2.6 Operator Characters

2.1 Characters and Lines

Visual Basic .NET programs are composed of characters from the Unicode character set.

```
Character ::= < any Unicode character except a LineTerminator >
```

See Also

2.2 Identifiers | 2.3 Keywords | 2.4 Literals | 2.5 Separators | 2.6 Operator Characters | 2. Lexical Grammar

2.1.1 Line Terminators

Unicode line break characters separate logical lines.

```
LineTerminator ::=
   < Unicode carriage return character (0x000D) > |
   < Unicode linefeed character (0x000A) > |
   < Unicode carriage return character > < Unicode linefeed character > |
   < Unicode line separator character (0x2028) > |
   < Unicode paragraph separator character (0x2029) >
```

See Also

2.1.2 Line Continuation | 2.1.3 White Space | 2.1.4 Comments | 2.1 Characters and Lines

2.1.2 Line Continuation

A *line continuation* consists of at least one white-space character that immediately precedes a single underscore character as the last character (other than white space) in a text line. A line continuation allows a logical line to span more than one physical line. Line continuations are treated as if they were white space, even though they are not.

The following program shows some line continuations:

```
Imports System

Module Test
    Function Func( _
        ByVal Param1 As Integer, _
        ByVal Param2 As Integer )

        If (Param1 < Param2) Or _
            (Param1 > Param2) Then
            Console.WriteLine("Not equal")
        End If
    End Function
End Module
```

```
LineContinuation ::= WhiteSpace _ [ WhiteSpace+ ] LineTerminator
```

See Also

2.1.1 Line Terminators | 2.1.3 White Space | 2.1.4 Comments | 2.1 Characters and Lines |
Breaking and Combining Statements in Code (Visual Basic Language Concepts)

2.1.3 White Space

White space serves only to separate tokens and is otherwise ignored. Logical lines containing only white space are ignored.

> **Note** Line terminators are not considered white space.

```
WhiteSpace ::=
   < Unicode blank characters (class Zs) > |
   < Unicode tab character (0x0009) >
```

See Also

2.1.1 Line Terminators | 2.1.2 Line Continuation | 2.1.4 Comments | 2.1 Characters and Lines

2.1.4 Comments

A *comment* begins with a single-quote character or the keyword **REM**. A single-quote character is either an ASCII single-quote character, a Unicode left single-quote character, or a Unicode right single-quote character. Comments can begin anywhere on a source line, and the end of the physical line ends the comment. The compiler ignores the characters between the beginning of the comment and the line terminator. Consequently, comments cannot extend across multiple lines by using line continuations.

```
Comment ::= CommentMarker [ Character+ ]
CommentMarker ::= SingleQuoteCharacter | REM
SingleQuoteCharacter ::=
   ' |
   < Unicode left single-quote character (0x2018) |
   < Unicode right single-quote character (0x2019)
```

See Also

2.1.1 Line Terminators | 2.1.2 Line Continuation | 2.1.3 White Space | 2.1 Characters and Lines |
REM Statement (Visual Basic Language Reference)

2.2 Identifiers

An *identifier* is a name. Visual Basic .NET identifiers conform to the Unicode Standard Annex 15 with one exception: identifiers may begin with an underscore (connector) character. If an identifier begins with an underscore, it must contain at least one other valid identifier character to disambiguate it from a line continuation.

Regular identifiers may not match keywords, but escaped identifiers can. An *escaped identifier* is an identifier delimited by square brackets. Escaped identifiers follow the same rules as regular identifiers except that they may match keywords and may not have type characters.

This example defines a class named `class` with a shared method named `shared` that takes a parameter named `boolean` and then calls the method.

```
Imports System

Class [class]
    Shared Sub [shared](ByVal [boolean] As Boolean)
        If [boolean] Then
            Console.WriteLine("true")
        Else
            Console.WriteLine("false")
        End If
    End Sub
End Class

Module [module]
    Sub Main()
        [class].[shared](True)
    End Sub
End Module
```

Identifiers are case insensitive, so two identifiers are considered to be the same identifier if they differ only in case.

> **Note** The Unicode Standard one-to-one case mappings are used when comparing identifiers, and any locale-specific case mappings are ignored.

```
Identifier ::=
   NonEscapedIdentifier [ TypeCharacter ] |
   EscapedIdentifier
NonEscapedIdentifier ::= < IdentifierName but not Keyword >
EscapedIdentifier ::= [ IdentifierName ]
IdentifierName ::= IdentifierStart [ IdentifierCharacter+ ]
IdentifierStart ::=
   AlphaCharacter |
   UnderscoreCharacter IdentifierCharacter
IdentifierCharacter ::=
   UnderscoreCharacter |
   AlphaCharacter |
   NumericCharacter |
   CombiningCharacter |
   FormattingCharacter
AlphaCharacter ::=
   < Unicode alphabetic character (classes Lu, Ll, Lt, Lm, Lo, Nl) >
NumericCharacter ::= < Unicode decimal digit character (class Nd) >
CombiningCharacter ::= < Unicode combining character (classes Mn, Mc) >
FormattingCharacter ::= < Unicode formatting character (class Cf) >
UnderscoreCharacter ::= < Unicode connection character (class Pc) >
IdentifierOrKeyword ::= Identifier | Keyword
```

See Also

2.1 Characters and Lines | 2.3 Keywords | 2.4 Literals | 2.5 Separators | 2.6 Operator Characters | 2. Lexical Grammar | Declared Element Names (Visual Basic Language Concepts)

2.2.1 Type Characters

A *type character* denotes the type of the preceding identifier. The type character is not considered part of the identifier. If a declaration includes a type character, the type character must agree with the type specified in the declaration itself; otherwise, a compile-time error occurs. If the declaration omits the type (for example, if it does not specify an **As** clause), the type character is implicitly substituted as the type of the declaration.

No white space may come between an identifier and its type character. There are no type characters for **Byte** or **Short**, due to a lack of suitable characters.

Appending a type character to an identifier that conceptually does not have a type (for example, a namespace name) or to an identifier whose type disagrees with the type of the type character causes a compile-time error.

The following example shows the use of type characters:

```
' The follow line will cause an error: standard modules have no type.
Module Test1#
End Module

Module Test2
    ' This function takes a Long parameter and returns a String.
    Function Func$(ByVal Param&)
        ' The following line causes an error because the type character
        ' conflicts with the declared type of StringFunc and LongParam.
        Func# = CStr(Param@)

        ' The following line is valid.
        Func$ = CStr(Param&)
    End Function
End Module
```

The type character **!** presents a special problem in that it can be used both as a type character and as a separator in the language. To remove ambiguity, a **!** character is a type character as long as the character that follows it cannot start an identifier. If it can, then the **!** character is a separator, not a type character.

```
TypeCharacter ::=
   IntegerTypeCharacter |
   LongTypeCharacter |
   DecimalTypeCharacter |
   SingleTypeCharacter |
   DoubleTypeCharacter |
   StringTypeCharacter
IntegerTypeCharacter ::= %
LongTypeCharacter ::= &
DecimalTypeCharacter ::= @
SingleTypeCharacter ::= !
DoubleTypeCharacter ::= #
StringTypeCharacter ::= $
```

See Also

2.2 Identifiers | 2.1 Characters and Lines | 2.3 Keywords | 2.4 Literals | 2.5 Separators | 2.6 Operator Characters | 2. Lexical Grammar | Type Characters (Visual Basic Language Concepts)

2.3 Keywords

A *keyword* is a word that has special meaning in a language construct. All keywords are reserved by the language and may not be used as identifiers unless the identifiers are escaped.

> **Note** **EndIf**, **GoSub**, **Let**, **Variant**, and **Wend** are retained as keywords, although they are no longer used in Visual Basic .NET.

```
Keyword ::= < member of keyword table >
```

AddHandler	**AddressOf**	**Alias**	**And**
AndAlso	**Ansi**	**As**	**Assembly**
Auto	**Boolean**	**ByRef**	**Byte**
ByVal	**Call**	**Case**	**Catch**
CBool	**CByte**	**CChar**	**CDate**
CDbl	**CDec**	**Char**	**CInt**
Class	**CLng**	**CObj**	**Const**
CShort	**CSng**	**CStr**	**CType**
Date	**Decimal**	**Declare**	**Default**
Delegate	**Dim**	**DirectCast**	**Do**
Double	**Each**	**Else**	**ElseIf**
End	**EndIf**	**Enum**	**Erase**
Error	**Event**	**Exit**	**False**
Finally	**For**	**Friend**	**Function**
Get	**GetType**	**GoSub**	**GoTo**
Handles	**If**	**Implements**	**Imports**
In	**Inherits**	**Integer**	**Interface**
Is	**Let**	**Lib**	**Like**
Long	**Loop**	**Me**	**Mod**
Module	**MustInherit**	**MustOverride**	**MyBase**
MyClass	**Namespace**	**New**	**Next**
Not	**Nothing**	**NotInheritable**	**NotOverridable**
Object	**On**	**Option**	**Optional**
Or	**OrElse**	**Overloads**	**Overridable**
Overrides	**ParamArray**	**Preserve**	**Private**
Property	**Protected**	**Public**	**RaiseEvent**
ReadOnly	**ReDim**	**REM**	**RemoveHandler**
Resume	**Return**	**Select**	**Set**
Shadows	**Shared**	**Short**	**Single**
Static	**Step**	**Stop**	**String**
Structure	**Sub**	**SyncLock**	**Then**
Throw	**To**	**True**	**Try**
TypeOf	**Unicode**	**Until**	**Variant**
Wend	**When**	**While**	**With**
WithEvents	**WriteOnly**	**Xor**	

See Also

2.1 Characters and Lines | 2.2 Identifiers | 2.4 Literals | 2.5 Separators | 2.6 Operator Characters | 2. Lexical Grammar | Visual Basic Language Keywords (Visual Basic Language Reference)

2.4 Literals

A *literal* is a textual representation of a particular value of a type. Literal types include Boolean, integer, floating point, string, character, and date.

```
Literal ::=
   BooleanLiteral |
   IntegerLiteral |
   FloatingPointLiteral |
   StringLiteral |
   CharacterLiteral |
   DateLiteral |
   Nothing
```

See Also

2.4.1 Boolean Literals | 2.4.2 Integer Literals | 2.4.3 Floating-Point Literals | 2.4.4 String Literals | 2.4.5 Character Literals | 2.4.6 Date Literals | 2.1 Characters and Lines | 2.2 Identifiers | 2.3 Keywords | 2.5 Separators | 2.6 Operator Characters | 2. Lexical Grammar

2.4.1 Boolean Literals

True and **False** are literals of the **Boolean** type that map to the true and false state, respectively.

```
BooleanLiteral ::= True | False
```

See Also

2.4.2 Integer Literals | 2.4.3 Floating-Point Literals | 2.4.4 String Literals | 2.4.5 Character Literals | 2.4.6 Date Literals | 2.4.7 Nothing | 2.4 Literals | Boolean Data Type (Visual Basic Language Reference) | Boolean Type (Visual Basic Language Concepts)

2.4.2 Integer Literals

Integer literals can be decimal (base 10), hexadecimal (base 16), or octal (base 8). A decimal integer literal is a string of decimal digits (0-9). A hexadecimal literal is **&H** followed by a string of hexadecimal digits (0-9, A-F). An octal literal is **&O** followed by a string of octal digits (0-7). Decimal literals directly represent the decimal value of the integral literal, whereas octal and hexadecimal literals represent the binary value of the integer literal (thus, `&H8000S` is –32768, not an overflow error).

The type of a literal is determined by its value or by the following type character. If no type character is specified, values in the range of the **Integer** type are typed as **Integer**; values outside the range for **Integer** are typed as **Long**. If an integer literal's type is of insufficient size to hold the integer literal, a compile-time error results.

```
IntegerLiteral ::= IntegralLiteralValue [ IntegralTypeCharacter ]
IntegralLiteralValue ::= IntLiteral | HexLiteral | OctalLiteral
IntegralTypeCharacter ::=
   ShortCharacter |
   IntegerCharacter |
   LongCharacter |
   IntegerTypeCharacter |
   LongTypeCharacter
ShortCharacter ::= S
IntegerCharacter ::= I
LongCharacter ::= L
IntLiteral ::= Digit+
HexLiteral ::= & H HexDigit+
OctalLiteral ::= & O OctalDigit+
Digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
HexDigit ::=
   0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F
OctalDigit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
```

See Also

2.4.1 Boolean Literals | 2.4.3 Floating-Point Literals | 2.4.4 String Literals | 2.4.5 Character Literals | 2.4.6 Date Literals | 2.4.7 Nothing | 2.4 Literals | Integer Data Type (Visual Basic Language Reference)

2.4.3 Floating-Point Literals

A floating-point literal is an integer literal followed by an optional decimal point (the ASCII period character) and mantissa, and an optional base 10 exponent. By default, a floating-point literal is of type **Double**. If the **Single**, **Double**, or **Decimal** type character is specified, the literal is of that type. If a floating-point literal's type is of insufficient size to hold the floating-point literal, a compile-time error results.

```
FloatingPointLiteral ::=
   FloatingPointLiteralValue [ FloatingPointTypeCharacter ] |
   IntLiteral FloatingPointTypeCharacter
FloatingPointTypeCharacter ::=
   SingleCharacter |
   DoubleCharacter |
   DecimalCharacter |
   SingleTypeCharacter |
   DoubleTypeCharacter |
   DecimalTypeCharacter
SingleCharacter ::= F
DoubleCharacter ::= R
DecimalCharacter ::= D
FloatingPointLiteralValue ::=
   IntLiteral . IntLiteral [ Exponent ] |
   . IntLiteral [ Exponent ] |
   IntLiteral Exponent
Exponent ::= E [ Sign ] IntLiteral
Sign ::= + | -
```

See Also

2.4.1 Boolean Literals | 2.4.2 Integer Literals | 2.4.4 String Literals | 2.4.5 Character Literals | 2.4.6 Date Literals | 2.4.7 Nothing | 2.4 Literals

2.4.4 String Literals

A string literal is a sequence of zero or more Unicode characters beginning and ending with an ASCII double-quote character, a Unicode left double-quote character, or a Unicode right double-quote character. Within a string, a sequence of two double-quote characters is an escape sequence representing a double quote in the string. A string constant is of the **String** type.

```
Imports System

Module Test
    Sub Main()
        ' This prints out: ".
        Console.WriteLine("""")

        ' This prints out: a"b.
        Console.WriteLine("a""b")

        ' This causes a compile error due to mismatched double-quotes.
        Console.WriteLine("a"b")
    End Sub
End Module
```

Each string literal does not necessarily result in a new string instance. When two or more string literals that are equivalent according to the string equality operator using binary comparison semantics appear in the same program, these string literals refer to the same string instance. For instance, the output of the following program is **True** because the two literals refer to the same string instance.

```
Imports System

Module Test
    Sub Main()
        Dim a As Object = "hello"
        Dim b As Object = "hello"
        Console.WriteLine(a Is b)
    End Sub
End Module
```

```
StringLiteral ::=
   DoubleQuoteCharacter [ StringCharacter+ ] DoubleQuoteCharacter
DoubleQuoteCharacter ::=
   " |
   < Unicode left double-quote character (0x201C) > |
   < Unicode right double-quote character (0x201D) >
StringCharacter ::=
   < Character except for DoubleQuoteCharacter > |
   DoubleQuoteCharacter DoubleQuoteCharacter
```

See Also

2.4.1 Boolean Literals | 2.4.2 Integer Literals | 2.4.3 Floating-Point Literals | 2.4.5 Character Literals | 2.4.6 Date Literals | 2.4.7 Nothing | 2.4 Literals | String Data Type (Visual Basic Language Reference) | Strings (Visual Basic Language Concepts)

2.4.5 Character Literals

A character literal represents a single Unicode character of the **Char** type. Two double-quote characters is an escape sequence representing the double-quote character.

```
Imports System

Module Test
    Sub Main()
        ' This prints out: a.
        Console.WriteLine("a"c)

        ' This prints out: ".
        Console.WriteLine(""""c)
    End Sub
End Module
```

```
CharacterLiteral ::=
   DoubleQuoteCharacter StringCharacter DoubleQuoteCharacter C
```

See Also

2.4.1 Boolean Literals | 2.4.2 Integer Literals | 2.4.3 Floating-Point Literals | 2.4.4 String Literals | 2.4.6 Date Literals | 2.4.7 Nothing | 2.4 Literals | Character Data Types (Visual Basic Language Concepts)

2.4.6 Date Literals

A date literal represents a particular moment in time expressed as a value of the **Date** type. The literal may specify both a date and a time, just a date, or just a time. If the date value is omitted, then January 1 of the year 1 in the Gregorian calendar is assumed. If the time value is omitted, then 12:00:00 AM is assumed.

To avoid problems with interpreting the year value in a date value, the year value cannot be two digits. When expressing a date in the first century AD/CE, leading zeros must be specified.

A time value may be specified either using a 24-hour value or a 12-hour value; time values that omit an **AM** or **PM** are assumed to be 24-hour values. If a time value omits the minutes, the literal **0** is used by default. If a time value omits the seconds, the literal **0** is used by default. If both minutes and second are omitted, then **AM** or **PM** must be specified. If the date value specified is outside the range of the **Date** type, a compile-time error occurs.

The following example contains several date literals.

```
Dim d As Date
d = # 8/23/1970 3:45:39AM #
d = # 8/23/1970 #              ' Date value: 8/23/1970 12:00:00AM.
d = # 3:45:39AM #              ' Date value: 1/1/1 3:45:39AM.
d = # 3:45:39 #                ' Date value: 1/1/1 3:45:39AM.
d = # 13:45:39 #               ' Date value: 1/1/1 1:45:39PM.
d = # 1AM #                    ' Date value: 1/1/1 1:00:00AM.
d = # 13:45:39PM #             ' This date value is not valid.
```

```
DateLiteral ::= # [ Whitespace+ ] DateOrTime [ Whitespace+ ] #
DateOrTime ::=
   DateValue Whitespace+ TimeValue |
   DateValue |
   TimeValue
DateValue ::=
   MonthValue / DayValue / YearValue |
   MonthValue – DayValue - YearValue
TimeValue ::=
   HourValue : MinuteValue [ : SecondValue ] [ WhiteSpace+ ] [ AMPM ]
MonthValue ::= IntLiteral
DayValue ::= IntLiteral
YearValue ::= IntLiteral
HourValue ::= IntLiteral
MinuteValue ::= IntLiteral
SecondValue ::= IntLiteral
AMPM ::= AM | PM
```

See Also

2.4.1 Boolean Literals | 2.4.2 Integer Literals | 2.4.3 Floating-Point Literals | 2.4.4 String Literals | 2.4.5 Character Literals | 2.4.7 Nothing | 2.4 Literals | Date Data Type (Visual Basic Language Reference) | Date Type (Visual Basic Language Concepts)

2.4.7 Nothing

Nothing is a special literal; it does not have a type and is convertible to all types in the type system. When converted to a particular type, it is the equivalent of the default value of that type.

```
Nothing ::= Nothing
```

See Also

2.4.1 Boolean Literals | 2.4.2 Integer Literals | 2.4.3 Floating-Point Literals | 2.4.4 String Literals | 2.4.5 Character Literals | 2.4.6 Date Literals | 2.4 Literals | Nothing (Visual Basic Language Reference)

2.5 Separators

The following ASCII characters are separators:

```
Separator ::= ( | ) | { | } | ! | # | , | . | :
```

See Also

2.1 Characters and Lines | 2.2 Identifiers | 2.3 Keywords | 2.4 Literals | 2.6 Operator Characters | 2. Lexical Grammar

2.6 Operator Characters

The following ASCII characters denote operators and may be combined to denote other operators:

```
Operator ::= & | * | + | - | / | \ | ^ | < | = | >
```

See Also

2.1 Characters and Lines | 2.2 Identifiers | 2.3 Keywords | 2.4 Literals | 2.5 Separators | 2. Lexical Grammar |
11.12 Operator Expressions | Operators Summary (Visual Basic Language Reference) |
Operators (Visual Basic Language Concepts)

3. Preprocessing Directives

Once a file has been lexically analyzed, several kinds of source preprocessing occur. The most important, conditional compilation, determines which source is processed by the syntactic grammar; two other types of directives — external source directives and region directives — provide meta-information about the source but have no effect on compilation.

See Also

3.1 Conditional Compilation | 3.2 External Source Directives | 3.3 Region Directives | Directives (Visual Basic Language Reference)

3.1 Conditional Compilation

Conditional compilation controls whether sequences of logical lines are translated into actual code. At the beginning of conditional compilation, all logical lines are enabled; however, enclosing lines in conditional compilation statements may selectively disable those lines within the file, causing them to be ignored during the rest of the compilation process. Because the conditional compilation process is done after lexical analysis, even disabled lines must be lexically valid.

For example, the program

```
#Const A = True
#Const B = False
Class C
#If A Then
    Sub F()
    End Sub
#Else
    Sub G()
    End Sub
#End If
#If B Then
    Sub H()
    End Sub
#Else
    Sub I()
    End Sub
#End If
End Class
```

produces the exact same sequence of tokens as the program

```
Class C
    Sub F()
    End Sub

    Sub I()
    End Sub
End Class
```

```
Start ::= [ ConditionalElement+ ]
ConditionalElement ::=
   ConditionalConstantDeclaration |
   ConditionalIfGroup |
   LogicalLine
```

See Also

3.1.1 Conditional Constant Directives | 3.2 External Source Directives | 3.3 Region Directives |
Compiler Directive Summary (Visual Basic Language Reference) |
Conditional Compilation Overview (Visual Basic Language Concepts)

3.1.1 Conditional Constant Directives

Conditional constant statements define constants that exist in a separate conditional compilation declaration space scoped to the source file. The declaration space is special in that no explicit declaration of conditional compilation constants is necessary – conditional constants can be implicitly defined in a conditional compilation directive.

Prior to being assigned a value, a conditional compilation constant has the value **Nothing**. When a conditional compilation constant is assigned a value, which must be a constant expression, the type of the constant becomes the type of the value being assigned to it. A conditional compilation constant may be redefined multiple times throughout a source file.

For example, the following code prints only the string `about to print value` and the value of `Test`.

```
Imports System

Module M1
    Sub PrintValue(ByVal Test As Integer)
#Const DebugCode = True
#If DebugCode Then
        Console.WriteLine("about to print value")
#End If
#Const DebugCode = False
        Console.WriteLine(Test)
#If DebugCode Then
        Console.WriteLine("printed value")
#End If
    End Sub
End Module
```

The compilation environment may also define conditional constants in a conditional compilation declaration space.

```
ConditionalConstantDeclaration ::=
   # Const Identifier = ConstantExpression LineTerminator
```

See Also

3.1.2 Conditional Compilation Directives | 3.1 Conditional Compilation | #Const Directive (Visual Basic Language Reference) | Declaring Conditional Compilation Constants and Enumerations (Visual Basic Language Concepts)

3.1.2 Conditional Compilation Directives

Conditional compilation directives control conditional compilation and can only reference constant expressions and conditional compilation constants. Each of the constant expressions within a single conditional compilation group is evaluated and converted to the **Boolean** type in textual order from first to last until one of the conditional expressions evaluates to **True**. If an expression is not convertible to **Boolean**, a compile-time error results. Permissive semantics are always used when evaluating conditional compilation constant expressions, regardless of any **Option** directives or compilation environment settings.

All lines enclosed by the group, including nested conditional compilation directives, are disabled except for lines between the statement containing the **True** expression and the next conditional statement of the group, or lines between the **Else** statement and the **End If** statement if an **Else** appears in the group and all of the expressions evaluate to **False**.

In this example, the call to `WriteToLog` in the `Trace` conditional compilation directive is not processed because the surrounding `Debug` conditional compilation directive evaluates to **False**.

```
#Const Debug = False   ' Debugging off
#Const Trace = True    ' Tracing on

Class PurchaseTransaction
    Sub Commit()
#If Debug Then
        CheckConsistency()
#If Trace Then
        WriteToLog(Me.ToString())
#End If
#End If
        CommitHelper()
    End Sub
End Class
```

```
ConditionalIfGroup ::=
   # If ConstantExpression [ Then ] LineTerminator
   [ ConditionalElement+ ]
   [ ConditionalElseIfGroup+ ]
   [ ConditionalElseGroup ]
   # End If LineTerminator
ConditionalElseIfGroup ::=
   # ElseIf ConstantExpression [ Then ] LineTerminator
   [ ConditionalElement+ ]
ConditionalElseGroup ::=
   # Else LineTerminator
   [ ConditionalElement+ ]
```

See Also

3.1.1 Conditional Constant Directives | 3.1 Conditional Compilation | #If...Then...#Else Directives (Visual Basic Language Reference) | Conditional Compilation Overview (Visual Basic Language Concepts)

3.2 External Source Directives

A source file may include external source directives that indicate a mapping between source lines and text external to the source. External source directives have no effect on compilation and may not be nested. For example:

```
Imports System

Module Test
    Sub Main()
#ExternalSource("c:\wwwroot\inetpub\test.aspx", 30)
        Console.WriteLine("In test.aspx")
#End ExternalSource
    End Sub
End Module
```

```
Start ::= [ ExternalSourceElement+ ]
ExternalSourceElement ::= ExternalSourceGroup | LogicalLine
ExternalSourceGroup ::=
   # ExternalSource ( StringLiteral , IntLiteral ) LineTerminator
   [ LogicalLine+ ]
   # End ExternalSource LineTerminator
```

See Also

3.1 Conditional Compilation | 3.3 Region Directives | #ExternalSource Directive (Visual Basic Language Reference)

3.3 Region Directives

Region directives group lines of source code but have no other effect on compilation. The entire group can be collapsed and hidden, or expanded and viewed, in the integrated development environment (IDE). These directives are special in that they can neither start nor terminate within a method body. For example:

```
Module Test
#Region "Startup code – do not edit"
    Sub Main()
    End Sub
#End Region
End Module
```

```
Start ::= [ RegionElement+ ]
RegionElement ::= RegionGroup | LogicalLine
RegionGroup ::=
   # Region StringLiteral LineTerminator
   [ LogicalLine+ ]
   # End Region LineTerminator
```

See Also

3.1 Conditional Compilation | 3.2 External Source Directives | #Region Directive | Collapsing and Hiding Sections of Code (Visual Basic Language Concepts)

4. General Concepts

This chapter covers a number of concepts that are required to understand the semantics of the Microsoft Visual Basic .NET language. Many of the concepts should be familiar to Visual Basic programmers or C/C++ programmers, but their precise definitions may differ.

See Also

4.1 Declarations | 4.3 Inheritance | 4.4 Implementation | 4.5 Polymorphism | 4.6 Accessibility | 4.2 Scope | 4.7 Type and Namespace Names | 4.7.1 Qualified Name Resolution | 5. Attributes

4.1 Declarations

A Visual Basic .NET program is made up of named entities. These entities are introduced through *declarations* and represent the "meaning" of the program.

At a top level, *namespaces* are entities that organize other entities, such as nested namespaces and types. *Types* are entities that describe values and define executable code. Types may contain nested types and type members. *Type members* are constants, variables, methods, properties, events, enumeration values, and constructors.

An entity that can contain other entities defines a *declaration space*. Entities are introduced into a declaration space either through declarations or inheritance; the containing declaration space is called the entities' *declaration context*. Declaring an entity in a declaration space in turn defines a new declaration space that can contain further nested entity declarations; thus, the declarations in a program form a hierarchy of declaration spaces.

Except in the case of overloaded type members, it is invalid for declarations to introduce identically named entities of the same kind into the same declaration context. Additionally, a declaration space may never contain different kinds of entities with the same name; for example, a declaration space may never contain a variable and a method by the same name.

The declaration space of a namespace is "open ended," so two namespace declarations with the same fully qualified name contribute to the same declaration space. In the example below, the two namespace declarations contribute to the same declaration space, in this case declaring two classes with the fully qualified names `Data.Customer` and `Data.Order`:

```
Namespace Data
    Class Customer
    End Class
End Namespace

Namespace Data
    Class Order
    End Class
End Namespace
```

Because the two declarations contribute to the same declaration space, a compile-time error would occur if each contained a declaration of a class with the same name.

See Also

4.1.1 Overloading and Signatures | 4.3 Inheritance | 4.4 Implementation | 4.5 Polymorphism | 4.6 Accessibility | 4.2 Scope | 4.7 Type and Namespace Names | 4.7.1 Qualified Name Resolution | 5. Attributes

4.1.1 Overloading and Signatures

The only way to declare identically named entities of the same kind in a declaration space is through *overloading*. Only methods, instance constructors, and properties may be overloaded.

Overloaded type members must possess unique signatures. The signature of a type member consists of the name of the type member and the number and types of the member's parameters.

The following are not part of a member's signature, and hence cannot be overloaded on:

- Modifiers to a type member (for example, **Shared** or **Private**).
- Modifiers to a parameter (for example, **ByVal** or **ByRef**).
- The names of the parameters.
- The return type of a method or the element type of a property.

The following example shows a set of overloaded method declarations along with their signatures. This declaration would not be valid since several of the method declarations have identical signatures.

```
Interface ITest
    Sub F()                              ' Signature is F().
    Sub F(x As Integer)                  ' Signature is F(Integer).
    Sub F(ByRef x As Integer)            ' Signature is F(Integer).
    Sub F(x As Integer, y As Integer)    ' Signature is F(Integer, Integer).
    Function F(s As String) As Integer   ' Signature is F(String).
    Function F(x As Integer) As Integer  ' Signature is F(Integer).
    Sub F(a() As String)                 ' Signature is F(String()).
    Sub F(ParamArray ByVal a() As String) ' Signature is F(String()).
End Interface
```

A method with optional parameters is considered to have multiple signatures, one for each set of parameters that can be passed in by the caller. For example, the following method has three corresponding signatures:

```
Overloads Sub F(ByVal x As Short, _
    ByVal Optional y As Integer = 10, _
    ByVal Optional z As Long = 20)
```

These are the method's signatures:

- F(Short)
- F(Short, Integer)
- F(Short, Integer, Long)

See Also

4.1 Declarations | Overloads (Visual Basic Language Reference) | Procedure Overloading (Visual Basic Language Concepts)

4.2 Scope

The *scope* of an entity's name is the set of all declaration spaces within which it is possible to refer to that name without qualification. In general, the scope of an entity's name is its entire declaration context; however, an entity's declaration may contain nested declarations of entities with the same name. In that case, the nested entity *shadows*, or hides, the outer entity, and access to the shadowed entity is only possible through qualification.

Shadowing through nesting occurs in namespaces or types nested within namespaces, in types nested within other types, and in the bodies of type members. Shadowing through the nesting of declarations always occurs implicitly; no explicit syntax is required.

In the following example, within the `F` method, the instance variable `i` is shadowed by the local variable `i`, but within the `G` method, `i` still refers to the instance variable.

```
Class Test
    Private i As Integer = 0
    Sub F()
        Dim i As Integer = 1
    End Sub

    Sub G()
        i = 1
    End Sub
End Class
```

When a name in an inner scope hides a name in an outer scope, it shadows all overloaded occurrences of that name. In the following example, the call `F(1)` invokes the `F` declared in `Inner` because all outer occurrences of `F` are hidden by the inner declaration. For the same reason, the call `F("Hello")` is in error.

```
Class Outer
    Shared Sub F(i As Integer)
    End Sub

    Shared Sub F(s As String)
    End Sub

    Class Inner
        Shared Sub F(l As Long)
        End Sub

        Sub G()
            F(1) ' Invokes Outer.Inner.F.
            F("Hello") ' Error.
        End Sub
    End Class
End Class
```

See Also

4.1 Declarations | 4.3 Inheritance | 4.4 Implementation | 4.5 Polymorphism | 4.6 Accessibility | 4.7 Type and Namespace Names | 5. Attributes | Scope (Visual Basic Language Concepts)

4.3 Inheritance

An inheritance relationship is one in which one type (the *derived* type) derives from another (the *base* type), such that the derived type's declaration space implicitly contains the accessible nonconstructor type members and nested types of its base type. In the following example, class `A` is the base class of `B`, and `B` is derived from `A`.

```
Class A
End Class

Class B
    Inherits A
End Class
```

Since `A` does not explicitly specify a base class, its base class is implicitly **Object**.

The following are important aspects of inheritance:

- Inheritance is transitive. If type *C* is derived from type *B*, and type *B* is derived from type *A*, type *C* inherits the type members declared in type *B* as well as the type members declared in type *A*.
- A derived type extends, but cannot narrow, its base type. A derived type can add new type members, and it can shadow inherited type members, but it cannot remove the definition of an inherited type member.
- Because an instance of a type contains all of the type members of its base type, a conversion always exists from a derived type to its base type.
- All types must have a base type, except for the type **Object**. Thus, **Object** is the ultimate base type of all types, and all types can be converted to it.
- Circularity in derivation is not permitted. That is, when a type `B` derives from a type `A`, it is an error for type `A` to derive directly or indirectly from type `B`.

The following example produces a compile-time error because the classes circularly depend on each other.

```
Class A
    Inherits B
End Class

Class B
    Inherits C
End Class

Class C
    Inherits A
End Class
```

It is valid for a class to derive from classes that are nested within it or within other classes. In the following example, `A` derives from `B.C` and `B` derives from `A`, but `C` does not derive from `B`.

```
Class A
    Inherits B.C
End Class

Class B
    Inherits A

    Public Class C
    End Class
End Class
```

In the next example, class `B` derives from class `A`, but class `A` does not derive from class `B`.

```
Class A
    Class B
        Inherits A
```

```
    End Class
End Class
```

See Also

4.3.1 MustInherit and NotInheritable Classes | 4.3.2 Interfaces and Multiple Inheritance | 4.3.3 Shadowing | 4.1 Declarations | 4.4 Implementation | 4.5 Polymorphism | 4.6 Accessibility | 4.2 Scope | 4.7 Type and Namespace Names | 5. Attributes | Inheritance (Visual Basic Language Concepts)

4.3.1 MustInherit and NotInheritable Classes

A **MustInherit** class is an incomplete type that can act only as a base type. A **MustInherit** class cannot be instantiated, so it is an error to use the **New** operator on one. It is valid to declare variables of **MustInherit** classes; such variables can only be assigned **Nothing** or a value that is of a class derived from the **MustInherit** class.

When a regular class is derived from a **MustInherit** class, the regular class must override all inherited **MustOverride** members. For example:

```
MustInherit Class A
    Public MustOverride Sub F()
End Class

MustInherit Class B
    Inherits A

    Public Sub G()
    End Sub
End Class

Class C
    Inherits B

    Public Overrides Sub F()
    End Sub
End Class
```

The **MustInherit** class `A` introduces a **MustOverride** method `F`. Class `B` introduces an additional method `G`, but does not provide an implementation of `F`. Class `B` must therefore also be declared **MustInherit**. Class `C` overrides `F` and provides an actual implementation. Since there are no outstanding **MustOverride** members in class `C`, it is not required to be **MustInherit**.

A **NotInheritable** class is a class from which another class cannot be derived. **NotInheritable** classes are primarily used to prevent unintended derivation.

In this example, class `B` is in error because it attempts to derive from the **NotInheritable** class `A`. A class can not be marked both **MustInherit** and **NotInheritable**.

```
NotInheritable Class A
End Class

Class B
    ' Error, a class cannot derive from a NotInheritable class.
    Inherits A
End Class
```

See Also

4.3.1 MustInherit and NotInheritable Classes | 4.3.2 Interfaces and Multiple Inheritance | 4.3.3 Shadowing | 4.3 Inheritance | MustInherit | NotInheritable (Visual Basic Language Reference) |
Inheritance and the Base Object Class (Visual Basic Language Concepts)

4.3.2 Interfaces and Multiple Inheritance

Unlike other types, which only derive from a single base type, an interface may derive from multiple base interfaces. Because of this, an interface can inherit an identically named type member from different base interfaces. In such a case, the multiply-inherited name is not available in the derived interface, and referring to any of those type members through the derived interface causes a compile-time error, regardless of signatures or overloading. Instead, conflicting type members must be referenced through a base interface name.

In the following example, the first two statements cause compile-time errors because the multiply-inherited member `Count` is not available in interface `IListCounter`:

```
Interface IList
    Property Count() As Integer
End Interface

Interface ICounter
    Sub Count(ByVal i As Integer)
End Interface

Interface IListCounter
    Inherits IList
    Inherits ICounter
End Interface

Module Test
    Sub F(ByVal x As IListCounter)
        x.Count(1)                  ' Error, Count is not available.
        x.Count = 1                 ' Error, Count is not available.
        CType(x, IList).Count = 1   ' Ok, invokes IList.Count.
        CType(x, ICounter).Count(1) ' Ok, invokes ICounter.Count.
    End Sub
End Module
```

As illustrated by the example, the ambiguity is resolved by casting `x` to the appropriate base interface type. Such casts have no run-time costs; they merely consist of viewing the instance as a less-derived type at compile time.

When a single type member is inherited from the same base interface through multiple paths, the type member is treated as if it were only inherited once. In other words, the derived interface only contains one instance of each type member inherited from a particular base interface. For example:

```
Interface IBase
    Sub F(ByVal i As Integer)
End Interface

Interface ILeft
    Inherits IBase
End Interface

Interface IRight
    Inherits IBase
End Interface

Interface IDerived
    Inherits ILeft, IRight
End Interface

Class Derived
    Implements IDerived

    ' Only have to implement F once.
    Sub F(ByVal i As Integer) Implements IDerived.F
    End Sub
End Class
```

If a type member name is shadowed in one path through the inheritance hierarchy but not through all of them, the name is still unavailable in derived interfaces. In the following example, the `IBase.F` member is shadowed by the `ILeft.F` member, but is not shadowed in `IRight`:

```
Interface IBase
    Sub F(ByVal i As Integer)
End Interface

Interface ILeft
    Inherits IBase
    Shadows Sub F(ByVal i As Integer)
End Interface

Interface IRight
    Inherits IBase
    Sub G()
End Interface

Interface IDerived
    Inherits ILeft, IRight
End Interface

Class Test
    Sub H(ByVal d As IDerived)
        d.F(1)                  ' Error: F not available in IDerived.
        CType(d, IBase).F(1)    ' Invokes IBase.F.
        CType(d, ILeft).F(1)    ' Invokes ILeft.F.
        CType(d, IRight).F(1)   ' Invokes IBase.F.
    End Sub
End Class
```

The invocation `d.F(1)` produces an error because in `IDerived` the name `F` is visible both from `ILeft.F` and from `IBase.F` through `IRight`.

See Also

4.3.1 MustInherit and NotInheritable Classes | 4.3.3 Shadowing | 4.3 Inheritance

4.3.3 Shadowing

A derived type shadows the name of an inherited type member by redeclaring it. Shadowing a name does not remove the inherited type members with that name; it merely makes all of the inherited type members with that name unavailable in the derived class. The shadowing declaration may be any type of entity.

Entities than can be overloaded can choose one of two forms of shadowing. *Shadowing by name* is specified using the **Shadows** keyword. An entity that shadows by name hides everything by that name in the base class, including all overloads. *Shadowing by name and signature* is specified using the **Overloads** keyword. An entity that shadows by name and signature hides everything by that name with the same signature as the entity. For example:

```
Imports System

Class Base
    Sub F()
    End Sub

    Sub F(ByVal i As Integer)
    End Sub

    Sub G()
    End Sub

    Sub G(ByVal i As Integer)
    End Sub
End Class

Class Derived
    Inherits Base

    ' Only hides F(Integer).
    Overloads Sub F(ByVal i As Integer)
    End Sub

    ' Hides G() and G(Integer).
    Shadows Sub G(ByVal i As Integer)
    End Sub
End Class

Module Test
    Sub Main()
        Dim x As Derived = New Derived()

        x.F() ' Calls Base.F().
        x.G() ' Error: No such method.
    End Sub
End Module
```

A shadowing method or property that does not specify **Shadows** or **Overloads** assumes **Shadows**. If one member of a set of overloaded entities specifies the **Shadows** or **Overloads** keyword, they all must specify it. The **Shadows** and **Overloads** keywords cannot be specified at the same time.

It is valid to shadow the name of a type member that has been multiply-inherited through interface inheritance (and which is thereby unavailable), thus making the name available in the derived interface.

For example:

```
Interface ILeft
    Sub F()
End Interface

Interface IRight
    Sub F()
End Interface
```

```
Interface ILeftRight
    Inherits ILeft, IRight
    Shadows Sub F()
End Interface

Module Test
    Sub G(ByVal i As ILeftRight)
        i.F() ' Calls ILeftRight.F.
        CType(i, ILeft).F() ' Calls ILeft.F.
        CType(i, IRight).F() ' Calls IRight.F.
    End Sub
End Module
```

Because methods are allowed to shadow inherited methods, it is possible for a class to contain several **Overridable** methods with the same signature. This does not present an ambiguity problem, since only the most-derived method is visible. In the following example, the `C` and `D` classes contain two **Overridable** methods with the same signature:

```
Imports System

Class A
    Public Overridable Sub F()
        Console.WriteLine("A.F")
    End Sub
End Class

Class B
    Inherits A

    Public Overrides Sub F()
        Console.WriteLine("B.F")
    End Sub
End Class

Class C
    Inherits B

    Public Shadows Overridable Sub F()
        Console.WriteLine("C.F")
    End Sub
End Class

Class D
    Inherits C

    Public Overrides Sub F()
        Console.WriteLine("D.F")
    End Sub
End Class

Module Test
    Sub Main()
        Dim d As New D()
        Dim a As A = d
        Dim b As B = d
        Dim c As C = d
        a.F()
        b.F()
        c.F()
        d.F()
    End Sub
End Module
```

There are two **Overridable** methods here: one introduced by class `A` and the one introduced by class `C`. The method introduced by class `C` hides the method inherited from class `A`. Thus, the **Overrides** declaration in class `D` overrides the method introduced by class `C`, and it is not possible for class `D` to override the method introduced by class `A`. The example produces the output:

```
B.F
B.F
D.F
D.F
```

It is possible to invoke the hidden **Overridable** method by accessing an instance of class `D` through a less-derived type in which the method is not hidden.

It is not valid to shadow a **MustOverride** method, because in most cases this would make the class unusable. For example:

```
MustInherit Class Base
    Public MustOverride Sub F()
End Class

MustInherit Class Derived
    Inherits Base

    Public Shadows Sub F()
    End Sub
End Class

Class MoreDerived
    Inherits Derived

    ' Error: MustOverride method Base.F is not overridden.
End Class
```

In this case, the class `MoreDerived` is required to override the **MustOverride** method `Base.F`, but because the class `Derived` shadows `Base.F`, this is not possible. There is no way to declare a valid descendent of `Derived`.

In contrast to shadowing a name from an outer scope, shadowing an accessible name from an inherited scope causes a warning to be reported, as in the following example:

```
Class Base
    Public Sub F()
    End Sub

    Private Sub G()
    End Sub
End Class

Class Derived
    Inherits Base

    Public Sub F() ' Warning: shadowing an inherited name.
    End Sub

    Public Sub G() ' No warning, Base.G is not accessible here.
    End Sub
End Class
```

The declaration of method `F` in class `Derived` causes a warning to be reported. Shadowing an inherited name is specifically not an error, since that would preclude separate evolution of base classes. For example, the above situation might have come about because a later version of class `Base` introduced a method `F` that was not present in an earlier version of the class. Had the above situation been an error, *any* change made to a base class in a separately versioned class library could potentially cause derived classes to become invalid.

The warning caused by shadowing an inherited name can be eliminated through use of the **Shadows** or **Overloads** modifier:

```
Class Base
    Public Sub F()
    End Sub
End Class
```

```
Class Derived
    Inherits Base

    Public Shadows Sub F() 'OK.
    End Sub
End Class
```

The **Shadows** modifier indicates the intention to shadow the inherited member. It is not an error to specify the **Shadows** or **Overloads** modifier if there is no type member name to shadow.

A declaration of a new member shadows an inherited member only within the scope of the new member, as in the following example:

```
Class Base
    Public Shared Sub F()
    End Sub
End Class

Class Derived
    Inherits Base

    Private Shared Shadows Sub F() ' Shadows Base.F in class Derived only.
    End Sub
End Class

Class MoreDerived
    Inherits Derived

    Shared Sub G()
        F() ' Invokes Base.F.
    End Sub
End Class
```

In the example above, the declaration of method `F` in class `Derived` shadows the method `F` that was inherited from class `Base`, but since the new method `F` in class `Derived` has **Private** access, its scope does not extend to class `MoreDerived`. Thus, the call `F()` in `MoreDerived.G` is valid and will invoke `Base.F`. In the case of overloaded type members, the entire set of overloaded type members is treated as if they all had the most permissive access for the purposes of shadowing.

```
Class Base
    Public Sub F()
    End Sub
End Class

Class Derived
    Inherits Base

    Private Shadows Sub F()
    End Sub

    Public Shadows Sub F(ByVal i As Integer)
    End Sub
End Class

Class MoreDerived
    Inherits Derived

    Public Sub G()
        F()   ' Error. No accessible member with this signature.
    End Sub
End Class
```

In this example, even though the declaration of `F()` in `Derived` is declared with **Private** access, the overloaded `F(Integer)` is declared with **Public** access. Therefore, for the purpose of shadowing, the name `F` in `Derived` is treated as if it was **Public**, so both methods shadow `F` in `Base`.

See Also

4.3.1 MustInherit and NotInheritable Classes | 4.3.2 Interfaces and Multiple Inheritance | 4.3 Inheritance | Shadows | Shadowing

4.4 Implementation

An *implementation* relationship exists when a type declares that it implements an interface and the type implements all the type members of the interface. A type that implements a particular interface is convertible to that interface. Interfaces cannot be instantiated, but it is valid to declare variables of interfaces; such variables can only be assigned a value that is of a class that implements the interface. For example:

```
Interface ITestable
    Function Test(ByVal Value As Byte) As Boolean
End Interface

Class TestableClass
    Implements ITestable

    Function Test(ByVal Value As Byte) _
             As Boolean Implements ITestable.Test
        Return (Value > 128)
    End Function
End Class

Module Test
    Sub F()
        Dim x As ITestable = New TestableClass
        Dim b As Boolean

        b = x.Test(34)
    End Sub
End Module
```

A type implementing an interface with multiply-inherited type members must still implement those methods, even though they cannot be accessed directly from the derived interface being implemented. For example:

```
Interface ILeft
    Sub Test()
End Interface

Interface IRight
    Sub Test()
End Interface

Interface ILeftRight
    Inherits ILeft, IRight
End Interface

Class LeftRight
    Implements ILeftRight

    ' Has to reference ILeft explicitly.
    Sub TestLeft() Implements ILeft.Test
    End Sub

    ' Has to reference IRight explicitly.
    Sub TestRight() Implements IRight.Test
    End Sub

    ' This causes an error because Test is not available in ILeftRight.
    Sub TestLeftRight() Implements ILeftRight.Test
    End Sub
End Class
```

Even **MustInherit** classes must provide implementations of all the members of implemented interfaces; however, they can defer implementation of these methods by declaring them as **MustOverride**. For example:

```
Interface ITest
```

```
    Sub Test1()
    Sub Test2()
End Interface

MustInherit Class TestBase
    Implements ITest

    ' Provides an implementation.
    Sub Test1() Implements ITest.Test1
    End Sub

    ' Defers implementation.
    MustOverride Sub Test2() Implements ITest.Test2
End Class

Class TestDerived
    Inherits TestBase

    ' Have to implement MustOverride method.
    Overrides Sub Test2()
    End Sub
End Class
```

See Also

4.4.1 Implementing Methods | 4.1 Declarations | 4.3 Inheritance | 4.5 Polymorphism | 4.6 Accessibility | 4.2 Scope | 4.7 Type and Namespace Names | 5. Attributes | 8. Conversions | Implements Statement (Visual Basic Language Reference) | Implements Keyword (Visual Basic Language Concepts)

4.4.1 Implementing Methods

A type *implements* a type member of an implemented interface by supplying a method with an **Implements** clause. The two type members must have the same number of parameters, and all of the types and modifiers of the parameters must match, including the default value of optional parameters. For example:

```
Interface ITest
    Sub F(ByRef x As Integer)
    Sub G(ByVal Optional y As Integer = 20)
    Sub H(ByVal Paramarray z() As Integer)
End Interface

Class Test
    Implements ITest

    ' Error: ByRef/ByVal mismatch.
    Sub F(ByVal x As Integer) Implements ITest.F
    End Sub

    ' Error: Defaults do not match.
    Sub G(ByVal Optional y As Integer = 10) Implements ITest.G
    End Sub

    ' Error: Paramarray does not match.
    Sub H(ByVal z() As Integer) Implements ITest.H
    End Sub
End Class
```

A single method may implement any number of interface type members if they all meet the above criteria. For example:

```
Interface ITest
    Sub F(ByVal i As Integer)
    Sub G(ByVal i As Integer)
End Interface

Class Test
    Implements ITest

    Sub F(ByVal i As Integer) Implements ITest.F, ITest.G
    End Sub
End Class
```

A derived type may not implement an interface type member that is already implemented by any of its base types; however, when an interface method is implemented using an overridable method in a base type, a derived type may override the overridable method and alter the implementation of the interface. For example:

```
Interface ITest
    Sub Test1()
    Sub Test2()
End Interface

Class Base
    Implements ITest

    Public Sub Test1() Implements ITest.Test1
    End Sub

    Public Overridable Sub Test2() Implements ITest.Test2
    End Sub
End Class

Class Derived
    Inherits Base
```

```
    ' Error: ITest.Test1 already implemented.
    Public Sub Test1() Implements ITest.Test1
    End Sub

    ' Valid: Overrides base implementation.
    Public Overrides Sub Test2()
    End Sub
End Class
```

A derived type may implement an interface whose base interfaces are implemented by the derived type's base types. In that case, the derived type only implements the interface's type members not already implemented by the base types. For example:

```
Interface IBase
    Sub Base()
End Interface

Interface IDerived
    Inherits IBase

    Sub Derived()
End Interface

Class Base
    Implements IBase

    Public Sub Base() Implements IBase.Base
    End Sub
End Class

Class Derived
    Inherits Base
    Implements IDerived

    ' Error: IBase.Base already implemented by Base.
    Public Sub Base() Implements IDerived.Base
    End Sub

    ' Valid: IDerived.Derived not implemented by Base.
    Public Sub Derived() Implements IDerived.Derived
    End Sub
End Class
```

See Also

4.5 Polymorphism | Implements Statement (Visual Basic Language Reference) |
Implements Keyword (Visual Basic Language Concepts)

4.5 Polymorphism

Polymorphism provides the ability to vary the implementation of a method or property. With polymorphism, the same method or property can perform different actions depending on the run-time type of the instance that invokes it. Methods or properties that are polymorphic are called *overridable*. By contrast, the implementation of a non-overridable method or property is invariant; the implementation is the same whether the method or property is invoked on an instance of the class in which it is declared or an instance of a derived class. When a non-overridable method or property is invoked, the compile-time type of the instance is the determining factor. For example:

```
Class Base
    Public Overridable Property X() As Integer
        Get
        End Get

        Set
        End Set
    End Property
End Class

Class Derived
    Public Overrides Property X() As Integer
        Get
        End Get

        Set
        End Set
    End Property
End Class

Module Test
    Sub F()
        Dim Z As Base

        Z = New Base()
        Z.X = 10            ' Calls Base.X
        Z = New Derived()
        Z.X = 10            ' Calls Derived.X
    End Sub
End Module
```

An overridable method may also be **MustOverride**, which means that it provides no method body and must be overridden. **MustOverride** methods are only allowed in **MustInherit** classes.

In the following example, the class `Shape` defines the abstract notion of a geometrical shape object that can paint itself:

```
MustInherit Public Class Shape
    Public MustOverride Sub Paint(ByVal g As Graphics, _
                                  ByVal r As Rectangle)
End Class

Public Class Ellipse
    Inherits Shape

    Public Overrides Sub Paint(ByVal g As Graphics, ByVal r As Rectangle)
        g.drawEllipse(r)
    End Sub
End Class

Public Class Box
    Inherits Shape

    Public Overrides Sub Paint(ByVal g As Graphics, ByVal r As Rectangle)
        g.drawRect(r)
    End Sub
End Class
```

The `Paint` method is **MustOverride** because there is no meaningful default implementation. The `Ellipse` and `Box` classes are concrete `Shape` implementations. Because these classes are not **MustInherit**, they are required to override the `Paint` method and provide an actual implementation.

It is an error for a base access to reference a **MustOverride** method, as the following example demonstrates:

```
Class A
    Public MustOverride Sub F()
End Class

Class B
    Inherits A

    Public Overrides Sub F()
        MyBase.F() ' Error, MyBase.F is MustOverride.
    End Sub
End Class
```

An error is reported for the `MyBase.F()` invocation because it references a **MustOverride** method.

See Also

4.5.1 Overriding Methods | 4.1 Declarations | 4.3 Inheritance | 4.4 Implementation | 4.6 Accessibility | 4.2 Scope | 4.7 Type and Namespace Names | 4.7.1 Qualified Name Resolution | 5. Attributes | Polymorphism

4.5.1 Overriding Methods

A type may *override* an inherited overridable method by declaring a method with the same name and signature, and marking the declaration with the **Overrides** modifier. Whereas an **Overridable** method declaration introduces a new method, an **Overrides** method declaration replaces the inherited implementation of the method.

An overriding method may also be declared **NotOverridable**, which prevents any further overriding of the method in derived types. In effect, **NotOverridable** methods become non-overridable in any further derived classes.

Consider the following example:

```
Imports System

Class A
    Public Overridable Sub F()
        Console.WriteLine("A.F")
    End Sub

    Public Overridable Sub G()
        Console.WriteLine("A.G")
    End Sub
End Class

Class B
    Inherits A

    Public Overrides NotOverridable Sub F()
        Console.WriteLine("B.F")
    End Sub

    Public Overrides Sub G()
        Console.WriteLine("B.G")
    End Sub
End Class

Class C
    Inherits B

    Public Overrides Sub G()
        Console.WriteLine("C.G")
    End Sub
End Class
```

In the example, class `B` provides two **Overrides** methods: a method `F` that has the **NotOverridable** modifier and a method `G` that does not. Use of the **NotOverridable** modifier prevents class `C` from further overriding method `F`.

A compile-time error occurs unless all of the following are true of an overriding method:

- The declaration context contains an accessible inherited method with the same signature as the overriding method.
- The inherited method being overridden is overridable. In other words, the inherited method being overridden is not **Shared** or **NotOverridable**.
- The accessibility domain of the method being declared is the same as the accessibility domain of the inherited method being overridden. There is one exception: a **Protected Friend** method must be overridden by a **Protected** method if the two methods are not in the same program.
- The parameters of the overriding method match the overridden method's parameters in regards to usage of the **ByVal**, **ByRef**, **ParamArray,** and **Optional** modifiers, including the values provided for optional parameters.

An override declaration can access the overridden base method using a base access, as in the following example:

```
Imports System

Class Base
    Private x As Integer
```

```
    Public Overridable Sub PrintVariables()
        Console.WriteLine("x = " & x)
    End Sub
End Class

Class Derived
    Inherits Base

    Private y As Integer

    Public Overrides Sub PrintVariables()
        MyBase.PrintVariables()
        Console.WriteLine("y = " & y)
    End Sub
End Class
```

In the example, the invocation of `MyBase.PrintVariables()` in class `Derived` invokes the `PrintVariables` method declared in class `Base`. A base access disables the overridable invocation mechanism and simply treats the base method as a non-overridable method. Had the invocation in `Derived` been written `CType(Me, Base).PrintVariables()`, it would recursively invoke the `PrintVariables` method declared in `Derived`, not the one declared in `Base`.

Only when it includes an **Overrides** modifier can a method override another method. In all other cases, a method with the same signature as an inherited method simply shadows the inherited method, as in the example below:

```
Class Base
    Public Overridable Sub F()
    End Sub
End Class

Class Derived
    Inherits Base

    Public Overridable Sub F() ' Warning, shadowing inherited F().
    End Sub
End Class
```

In the example, the method `F` in class `Derived` does not include an **Overrides** modifier and therefore does not override method `F` in class `Base`. Rather, method `F` in class `Derived` shadows the method in class `Base`, and a warning is reported because the declaration does not include a **Shadows** or **Overloads** modifier.

In the following example, method `F` in class `Derived` shadows the overridable method `F` inherited from class `Base`:

```
Class Base
    Public Overridable Sub F()
    End Sub
End Class

Class Derived
    Inherits Base

    Private Shadows Sub F() ' Shadows Base.F within Derived.
    End Sub
End Class

Class MoreDerived
    Inherits Derived

    Public Overrides Sub F() ' Ok, overrides Base.F.
    End Sub
End Class
```

Since the new method `F` in class `Derived` has **Private** access, its scope only includes the class body of `Derived` and does not extend to class `MoreDerived`. The declaration of method `F` in class `MoreDerived` is therefore permitted to override the method `F` inherited from class `Base`.

When an **Overridable** method is invoked, the most derived implementation of the instance method is called, based on the type of

the instance, regardless of whether the call is to the method in the base class or the derived class. The most derived implementation of an **Overridable** method `M` with respect to a class `R` is determined as follows:

- If `R` contains the introducing **Overridable** declaration of `M`, this is the most derived implementation of `M`.
- Otherwise, if `R` contains an override of `M`, this is the most derived implementation of `M`.
- Otherwise, the most derived implementation of `M` is the same as that of the direct base class of `R`.

See Also

4.5 Polymorphism | 4.1 Declarations | 4.6 Accessibility | Overrides (Visual Basic Language Reference) | Overriding Properties and Methods (Visual Basic Language Concepts)

4.6 Accessibility

A declaration specifies the *accessibility* of the entity it declares. An entity's accessibility does not change the scope of an entity's name. The *accessibility domain* of a declaration is the set of all declaration spaces in which the declared entity is accessible.

The five access types are **Public**, **Protected**, **Friend**, **Protected Friend**, and **Private**. **Public** is the most permissive access type, and the four other types are all subsets of **Public**. The least permissive access type is **Private**, and the four other access types are all supersets of **Private**.

The access type for a declaration is specified via an optional access modifier, which can be **Public**, **Protected**, **Friend**, **Private**, or the combination of **Protected** and **Friend**. If no access modifier is specified, the default access type depends on the declaration context; the permitted access types also depend on the declaration context.

- Entities declared with the **Public** modifier have **Public** access. There are no restrictions on the use of **Public** entities.
- Entities declared with the **Protected** modifier have **Protected** access. **Protected** access can only be specified on members of classes (both regular type members and nested classes) or on **Overridable** members of standard modules and structures (which must, by definition, be inherited from **System.Object** or **System.ValueType**). A **Protected** member is accessible to a derived class, provided that either the member is not an instance member, or the access takes place through an instance of the derived class. **Protected** access is not a superset of **Friend** access.
- Entities declared with the **Friend** modifier have **Friend** access. An entity with **Friend** access is accessible only within the program that contains the entity declaration.
- Entities declared with the **Protected Friend** modifiers have the union of **Protected** and **Friend** access.
- Entities declared with the **Private** modifier have **Private** access. A **Private** entity is accessible only within its declaration context, including any nested entities.

The accessibility in a declaration does not depend on the accessibility of the declaration context. For example, a type declared with **Private** access may contain a type member with **Public** access.

The following code demonstrates various accessibility domains:

```
Public Class A
    Public Shared X As Integer
    Friend Shared Y As Integer
    Private Shared Z As Integer
End Class

Friend Class B
    Public Shared X As Integer
    Friend Shared Y As Integer
    Private Shared Z As Integer

    Public Class C
        Public Shared X As Integer
        Friend Shared Y As Integer
        Private Shared Z As Integer
    End Class

    Private Class D
        Public Shared X As Integer
        Friend Shared Y As Integer
        Private Shared Z As Integer
    End Class
End Class
```

The classes and members have the following accessibility domains:

- The accessibility domain of `A` and `A.X` is unlimited.
- The accessibility domain of `A.Y`, `B`, `B.X`, `B.Y`, `B.C`, `B.C.X`, and `B.C.Y` is the containing program.
- The accessibility domain of `A.Z` is `A`.
- The accessibility domain of `B.Z` and `B.D` is `B`, including `B.C` and `B.D`.
- The accessibility domain of `B.C.Z` is `B.C`.
- The accessibility domain of `B.D.X`, `B.D.Y`, and `B.D.Z` is `B.D`.

As the example illustrates, the accessibility domain of a member is never larger than that of a containing type. For example, even though all `X` members have **Public** declared accessibility, all but `A.X` have accessibility domains that are constrained by a containing type.

Access to **Protected** instance members must be through an instance of the derived type so that unrelated types cannot gain access to each other's protected members. For example:

```
Public Class User
    Protected Password As String
End Class

Public Class Employee
    Inherits User
End Class

Public Class Guest
    Inherits User

    Public Function GetPassword(ByVal U As User) As String
        ' Error: protected access has to go through derived type.
        Return U.Password
    End Function
End Class
```

In the above example, the class `Guest` only has access to the protected `Password` field if it is qualified with an instance of `Guest`. This prevents `Guest` from gaining access to the `Password` field of an `Employee` object simply by casting it to `User`.

```
AccessModifier ::= Public | Protected | Friend | Private
```

See Also

4.1 Declarations | 4.3 Inheritance | 4.4 Implementation | 4.5 Polymorphism | 4.2 Scope | 4.7 Type and Namespace Names | 4.7.1 Qualified Name Resolution | 5. Attributes | Accessibility (Visual Basic Language Concepts)

4.6.1 Constituent Types

The *constituent types* of a declaration are the types that are referenced by the declaration. For example, the type of a constant, the return type of a method and the parameter types of a constructor are all constituent types. The accessibility domain of a constituent type of a declaration must be the same as or a superset of the accessibility domain of the declaration itself. For example:

```
Public Class X
    Private Class Y
    End Class

    ' Error: Exposing private class Y outside of X.
    Public Function Z() As Y
    End Function

    ' Valid: Not exposing outside of X.
    Private Function A() As Y
    End Function
End Class

Private Class B
    Private Class C
    End Class

    ' Error: Exposing private class Y outside of B.
    Public Function D() As C
    End Function
End Class
```

See Also

4.6 Accessibility | 4.1 Declarations | 4.3 Inheritance | 4.4 Implementation | 4.5 Polymorphism | 4.2 Scope | 4.7 Type and Namespace Names | 4.7.1 Qualified Name Resolution | 5. Attributes | Accessibility (Visual Basic Language Concepts)

4.7 Type and Namespace Names

Many language constructs require a namespace or type to be specified; these can be specified by using a qualified form of the namespace or type's name. A *qualified name* consists of a series of identifiers separated by periods; the identifier on the right side of a period is resolved in the declaration space specified by the identifier on the left side of the period.

The *fully qualified name* of a namespace or type is a qualified name that contains the name of all containing namespaces and types. In other words, the fully qualified name of a namespace or type is `N.T`, where `T` is the name of the entity and `N` is the fully qualified name of its containing entity.

The example below shows several namespace and type declarations together with their associated fully qualified names in in-line comments.

```
Class A            ' A.
End Class

Namespace X        ' X.
    Class B        ' X.B.
        Class C    ' X.B.C.
        End Class
    End Class

    Namespace Y    ' X.Y.
        Class D    ' X.Y.D.
        End Class
    End Namespace
End Namespace

Namespace X.Y      ' X.Y.
    Class E        ' X.Y.E.
    End Class
End Namespace
```

Because other languages may introduce types and namespaces that match keywords in the language, Visual Basic .NET recognizes keywords to be part of a qualified name as long as they follow a period. Keywords used in this way are treated as identifiers. For example, the qualified identifier `X.Default.Class` is a valid qualified identifier, while `Default.Class` is not.

```
QualifiedIdentifier ::=
   Identifier |
   QualifiedIdentifier . IdentifierOrKeyword
NamespaceOrTypeName ::= QualifiedIdentifier
```

See Also

4.7.1 Qualified Name Resolution | 4.1 Declarations | 4.3 Inheritance | 4.4 Implementation | 4.5 Polymorphism | 4.6 Accessibility | 4.2 Scope | 5. Attributes | Namespace Statement (Visual Basic Language Reference) | Namespaces (Visual Basic Language Concepts)

4.7.1 Qualified Name Resolution

Given a qualified namespace or type name of the form `N.R`, where `R` is the rightmost identifier in the qualified name, the following steps describe how to determine to which namespace or type the qualified name refers:

1. Resolve `N`, which may be either a qualified or unqualified name.
2. If resolution of `N` fails or does not resolve to a namespace or type, a compile-time error occurs. If `R` matches the name of a namespace or type in `N`, then the qualified name refers to that namespace or type.
3. If `N` contains one or more standard modules, and `R` matches the name of a type in exactly one standard module, then the qualified name refers to that type. If `R` matches the name of types in more than one standard module, a compile-time error occurs.
4. Otherwise, a compile-time error occurs.

> **Note** An implication of this resolution process is that type members do not shadow namespaces or types when resolving namespace or type names.

See Also

4.7 Type and Namespace Names | 11.12.3 Operator Resolution | 11.8.1 Overloaded Method Resolution

4.7.2 Unqualified Name Resolution

Given an unqualified name `R`, the following steps describe how to determine to which namespace or type an unqualified name refers:

1. For each nested type containing the name reference, starting from the innermost type and going to the outermost, if `R` matches the name of an accessible nested type in the current type, then the unqualified name refers to that type.
2. For each nested namespace containing the name reference, starting from the innermost namespace and going to the outermost namespace, do the following:
   a. If `R` matches the name of an accessible type or nested namespace in the current namespace, then the unqualified name refers to that type or nested namespace.
   b. If the namespace contains one or more accessible standard modules, and `R` matches the name of an accessible nested type in exactly one standard module, then the unqualified name refers to that nested type. If `R` matches the name of accessible nested types in more than one standard module, a compile-time error occurs.
3. If the source file has one or more import aliases, and `R` matches the name of one of them, then the unqualified name refers to that import alias.
4. If the source file containing the name reference has one or more imports:
   a. If `R` matches the name of a namespace or an accessible type in exactly one import, then the unqualified name refers to that namespace or type. If `R` matches the name of a namespace or accessible type members in more than one import and all are not the same entity, a compile-time error occurs.
   b. If the imports contain one or more accessible standard modules, and `R` matches the name of an accessible nested type in exactly one standard module, then the unqualified name refers to that type. If `R` matches the name of accessible nested types in more than one standard module, a compile-time error occurs.
5. If the compilation environment defines one or more import aliases, and `R` matches the name of one of them, then the unqualified name refers to that import alias.
6. If the compilation environment defines one or more imports:
   a. If `R` matches the name of a namespace or an accessible type in exactly one import, then the unqualified name refers to that namespace or type. If `R` matches the name of namespace or an accessible type member in more than one import, a compile-time error occurs.
   b. If the imports contain one or more accessible standard modules, and `R` matches the name of an accessible nested type in exactly one standard module, then the unqualified name refers to that type. If `R` matches the name of accessible nested types in more than one standard module, a compile-time error occurs.
7. Otherwise, a compile-time error occurs.

> **Note** An implication of this resolution process is that type members do not shadow namespaces or types when resolving namespace or type names.

See Also

4.7 Type and Namespace Names | 11.12.3 Operator Resolution | 11.8.1 Overloaded Method Resolution | 6.3 Imports Statement

4.8 Variables

A *variable* represents a storage location. Every variable has a type that determines what values can be stored in the variable. Because Visual Basic .NET is a type-safe language, every variable in a program has a type and the language guarantees that values stored in variables are always of the appropriate type. Variables are always initialized to the default value of their type before any reference to the variable can be made. It is not possible to access uninitialized memory.

See Also

4. General Concepts | 4.1 Declarations | 4.2 Scope | 4.6 Accessibility

5. Attributes

The Visual Basic .NET language enables the programmer to specify modifiers on declarations, which represent information about the entities being declared. For example, affixing a class method with the modifiers **Public**, **Protected**, **Friend**, **Protected Friend**, or **Private** specifies its accessibility.

In addition to the modifiers defined by the language, Visual Basic .NET also enables programmers to create new modifiers, called *attributes*, and to use them when declaring new entities. These new modifiers, which are defined through the declaration of attribute classes, are then assigned to entities through *attribute blocks*.

> **Note** Attributes may be retrieved at run time through the .NET Framework's reflection APIs. These APIs are outside the scope of this specification.

For instance, a framework might define a **Help** attribute that can be placed on program elements such as classes and methods to provide a mapping from program elements to documentation, as the following example demonstrates:

```
Imports System

<AttributeUsage(AttributeTargets.All)> _
Public Class HelpAttribute
    Inherits Attribute

    Public Sub New(ByVal UrlValue As String)
        Me.UrlValue = UrlValue
    End Sub

    Public Topic As String
    Private UrlValue As String

    Public ReadOnly Property Url() As String
        Get
            Return UrlValue
        End Get
    End Property
End Class
```

The example defines an attribute class named **HelpAttribute**, or **Help** for short, that has one positional parameter (`UrlValue`) and one named argument (`Topic`).

The next example shows several uses of the attribute:

```
<Help("http://www.example.com/.../Class1.htm")> _
Public Class Class1
    <Help("http://www.example.com/.../Class1.htm", Topic := "F")> _
        Public Sub F()
    End Sub
End Class
```

The next example checks to see if `Class1` has a **Help** attribute, and writes out the associated `Topic` and `Url` values if the attribute is present.

```
Imports System

Module Test
    Sub Main()
        Dim type As Type = GetType(Class1)
        Dim arr() As Object() = _
            type.GetCustomAttributes(GetType(HelpAttribute), True)

        If arr.Length = 0 Then
            Console.WriteLine("Class1 has no Help attribute.")
        Else
            Dim ha As HelpAttribute = CType(arr(0), HelpAttribute)
            Console.WriteLine("Url = " & ha.Url & "Topic = " & ha.Topic)
```

```
        End If
    End Sub
End Module
```

See Also

4.1 Declarations | 4.3 Inheritance | 4.4 Implementation | 4.5 Polymorphism | 4.6 Accessibility | 4.2 Scope | 4.7 Type and Namespace Names | Attributes (Visual Basic Language Concepts)

5.1 Attribute Classes

An *attribute class* is a class that derives from **System.Attribute** and is not **MustInherit**. The attribute class must have a **System.AttributeUsage** attribute that declares what the attribute is valid on, whether it may be used multiple times in a declaration, and whether it is inherited. The following example defines an attribute class named `SimpleAttribute` that can be placed on class declarations and interface declarations:

```
Imports System

<AttributeUsage(AttributeTargets.Class Or AttributeTargets.Interface)> _
Public Class SimpleAttribute
    Inherits System.Attribute
End Class
```

The next example shows a few uses of the `Simple` attribute. Although the attribute class is named `SimpleAttribute`, uses of this attribute may omit the *Attribute* suffix, thus shortening the name to `Simple`:

```
<Simple()> Class Class1
End Class
<Simple()> Interface Interface1
End Interface
```

The **System.AttributeUsage** attribute has a variable initializer, `AllowMultiple`, which specifies whether the indicated attribute can be specified more than once for a given declaration. If `AllowMultiple` for an attribute is **True**, it is a *multiple-use attribute class*, and can be specified more than once on a declaration. If `AllowMultiple` for an attribute is **False** or unspecified for an attribute, it is a *single-use attribute class*, and can be specified at most once on a declaration.

The following example defines a multiple-use attribute class named `AuthorAttribute`:

```
Imports System

<AttributeUsage(AttributeTargets.Class, AllowMultiple := True)> _
Public Class AuthorAttribute
    Inherits System.Attribute

    Public Sub New(ByVal Value As String)
    End Sub

    Public ReadOnly Property Value() As String
        Get
        End Get
    End Property
End Class
```

The example shows a class declaration with two uses of the `Author` attribute:

```
<Author("Maria Hammond"), Author("Ramesh Meyyappan")> _
Class Class1
End Class
```

The **System.AttributeUsage** attribute has a public instance variable, `Inherited`, that specifies whether the attribute, when specified on a base type, is also inherited by types that derive from this base type. If the `Inherited` public instance variable is not initialized, a default value of **False** is used. Properties and events do not inherit attributes, although the methods defined by properties and events do. Interfaces do not inherit attributes.

If a single-use attribute is both inherited and specified on a derived type, the attribute specified on the derived type overrides the inherited attribute. If a multiple-use attribute is both inherited and specified on a derived type, both attributes are specified on the derived type. For example:

```
Imports System
```

```
<AttributeUsage(AttributeTargets.Class, AllowMultiple := True, _
                Inherited := True > _
Class MultiUseAttribute
    Inherits System.Attribute

    Public Sub New(ByVal Value As Boolean)
    End Sub
End Class

<AttributeUsage(AttributeTargets.Class, Inherited := True)> _
Class SingleUseAttribute
    Inherits Attribute

    Public Sub New(ByVal Value As Boolean)
    End Sub
End Class

<SingleUse(True), MultiUse(True)> Class Base
End Class

' Derived has three attributes defined on it: SingleUse(False),
' MultiUse(True) and MultiUse(False)
<SingleUse(False), MultiUse(False)> _
Class Derived
    Inherits Base
End Class
```

The positional parameters of the attribute are defined by the parameters of the public constructors of the attribute class. Positional parameters must be **ByVal** and may not specify **ByRef**, **Optional** or **Paramarray**. Public instance variables and properties are defined by public read-write properties or instance variables of the attribute class. The types that can be used in positional parameters and public instance variables and properties are restricted to attribute types. A type is an attribute type if it is one of the following:

- Any primitive type except for **Date** and **Decimal**.
- The type **System.Type**.
- An enumerated type, provided that it and the types in which it is nested (if any) have **Public** accessibility.

See Also

5.2 Attribute Blocks | 5.2.1 Attribute Names | 5.2.2 Attribute Arguments

5.2 Attribute Blocks

Attributes are specified in *attribute blocks*. Each attribute block is delimited by angle brackets ("< >"), with multiple attributes specified in a comma-separated list. The order in which attributes are specified is not significant. For example, the attribute blocks `<A, B>` and `<B, A>` are equivalent.

An attribute may not be specified on a kind of declaration it does not support, and single-use attributes may not be specified more than once in an attribute block. The example below causes errors both because it attempts to use `HelpString` on the interface `Interface1` and more than once on the declaration of `Class1`.

```
Imports System

<AttributeUsage(AttributeTargets.Class)> _
Public Class HelpStringAttribute
    Inherits System.Attribute

    Private InternalValue As String

    Public Sub New(ByVal Value As String)
        Me.InternalValue = Value
    End Sub

    Public ReadOnly Property Value() As String
        Get
            Return InternalValue
        End Get
    End Property
End Class

' Error: HelpString only applies to classes.
<HelpString("Description of Interface1")> _
Interface Interface1
    Sub Sub1()
End Interface

' Error: HelpString is single-use.
<HelpString("Description of Class1"), _
    HelpString("Another description of Class1")> _
Public Class Class1
End Class
```

An attribute consists of an optional attribute modifier, an attribute name, an optional list of positional arguments, and variable/property initializers. If there are no parameters or initializers, the parentheses may be omitted. If an attribute has a modifier, it must be in an attribute block at the top of a source file.

If a source file contains an attribute block at the top of the file that specifies attributes for the assembly or module that will contain the source file, each attribute in the attribute block must be prefixed by either the **Assembly** or **Module** modifier and a colon.

```
Attributes ::= < AttributeList >
AttributeList ::=
   Attribute |
   AttributeList , Attribute
Attribute ::=
   [ AttributeModifier : ] TypeName [ ( [ AttributeArguments ] ) ]
AttributeModifier ::= Assembly | Module
```

See Also

5.1 Attribute Classes | 5.2.1 Attribute Names | 5.2.2 Attribute Arguments | 5. Attributes |
Attributes (Visual Basic Language Concepts)

5.2.1 Attribute Names

The name of an attribute specifies an attribute class. By convention, attribute classes are named with the suffix `Attribute`. Uses of an attribute may either include or omit this suffix. Consequently the name of an attribute class that corresponds to an attribute identifier is either the identifier itself or the concatenation of the qualified identifier and `Attribute`. When the compiler resolves an attribute name, it appends `Attribute` to the name and tries the lookup. If that lookup fails, the compiler tries the lookup without the suffix. For example, uses of an attribute class `SimpleAttribute` may omit the `Attribute` suffix, thus shortening the name to `Simple`:

```
<Simple()> _
Class Class1
End Class

<Simple()> _
Interface Interface1
End Interface
```

The example above is semantically equivalent to the following:

```
<SimpleAttribute()> _
Class Class1
End Class

<SimpleAttribute()> _
Interface Interface1
End Interface
```

In general, attributes named with the suffix `Attribute` are preferred. The following example shows two attribute classes named `X` and `XAttribute`.

```
<AttributeUsage(AttributeTargets.All)> _
Public Class X
    Inherits System.Attribute
End Class

<AttributeUsage(AttributeTargets.All)> _
Public Class XAttribute
    Inherits System.Attribute
End Class

' Refers to XAttribute.
<X()> _
Class Class1
End Class

' Refers to XAttribute.
<XAttribute()> _
Class Class2
End Class
```

Both the attribute block `<X>` and the attribute block `<XAttribute>` refer to the attribute class named `XAttribute`. It is not possible to use `X` as an attribute until you remove the declaration for class `XAttribute`.

See Also

5.1 Attribute Classes | 5.2 Attribute Blocks | 5.2.2 Attribute Arguments | 5. Attributes | Attributes (Visual Basic Language Concepts)

5.2.2 Attribute Arguments

Arguments to an attribute may take two forms: positional arguments and instance variable/property initializers. Any positional arguments to the attribute must precede the instance variable/property initializers. A positional argument consists of a constant expression or a **GetType** expression. An instance variable/property initializer consists of an identifier, which can match keywords, followed by a colon and equal sign, and terminated by a constant expression or a **GetType** expression.

Given an attribute with attribute class `T`, positional argument list `P`, and instance variable/property initializer list `N`, these steps determine whether the arguments are valid:

1. Follow the compile-time processing steps for compiling an expression of the form `New T(P)`. This either results in a compile-time error or determines a constructor on `T` that is most applicable to the argument list.
2. If the constructor determined in step 1 has parameters that are not attribute types or is inaccessible at the declaration site, a compile-time error occurs.
3. For each instance variable/property initializer `Arg` in `N`:
   a. Let `Name` be the identifier of the instance variable/property initializer `Arg`.
   b. `Name` must identify a non-**Shared**, writeable, **Public** instance variable or parameterless property on `T` whose type is an attribute type. If `T` has no such instance variable or property, a compile-time error occurs.

For example:

```
Imports System

<AttributeUsage(AttributeTargets.All)> _
Public Class GeneralAttribute
    Inherits Attribute

    Public Sub New(ByVal x As Integer)
    End Sub

    Public Sub New(Byval x As Double)
    End Sub

    Public y As Type

    Public Property z As Integer
        Get
        End Get

        Set
        End Set
    End Property
End Class

' Calls the first constructor.
<General(10, z := 30, y := GetType(Integer))> _
Class Foo
End Class

' Calls the second constructor.
<General(10.5, z := 10)> _
Class Bar
End Class
```

```
AttributeArguments ::=
   AttributePositionalArgumentList |
   AttributePositionalArgumentList , VariablePropertyInitializerList |
   VariablePropertyInitializerList
AttributePositionalArgumentList ::=
   ConstantExpression |
   AttributePositionalArgumentList , ConstantExpression
VariablePropertyInitializerList ::=
   VariablePropertyInitializer |
```

```
    VariablePropertyInitializerList , VariablePropertyInitializer
VariablePropertyInitializer ::=
    IdentifierOrKeyword : = ConstantExpression
```

See Also

5.1 Attribute Classes | 5.2 Attribute Blocks | 5.2.1 Attribute Names | 5. Attributes | Attributes (Visual Basic Language Concepts)

6. Source Files and Namespaces

A Visual Basic .NET program consists of one or more source files. When a program is compiled, all of the source files are processed together; thus, source files can depend on each other, possibly in a circular fashion, without any forward-declaration requirement. The textual order of declarations in the program text is generally of no significance.

A source file consists of an optional set of option statements, import statements, and attributes, which are followed by a namespace body. The attributes, which must each have either the **Assembly** or **Module** modifier, apply to the .NET assembly or module produced by the compilation. The body of the source file functions as an implicit namespace declaration for the global namespace, meaning that all declarations at the top level of a source file are placed in the global namespace. For example:

```
File A.vb:
    Class A
    End Class

File B.vb:
    Class B
    End Class
```

The two source files contribute to the global namespace, in this case declaring two classes with the fully qualified names `A` and `B`. Because the two source files contribute to the same declaration space, it would have been an error if each contained a declaration of a member with the same name.

The compilation environment may override the namespace declarations into which a source file is implicitly placed.

```
Start ::=
   [ OptionStatement+ ]
   [ ImportsStatement+ ]
   [ Attributes ]
   [ NamespaceMemberDeclaration+ ]
```

See Also

6.2 Compilation Options | 6.3 Imports Statement | 6.4 Namespaces | Namespace Statement (Visual Basic Language Reference) | Namespaces (Visual Basic Language Concepts)

6.1 Program Startup and Termination

Program startup occurs when the execution environment executes a designated method, which is referred to as the program's *entry point*. This entry point method, which must always be named `Main`, must be shared and must have one of the following signatures:

```
Sub Main()
Sub Main(ByVal Args() As String)
Function Main() As Integer
Function Main(ByVal Args() As String) As Integer
```

The accessibility of the entry point method is irrelevant. If a program contains more than one suitable entry point, the compilation environment must designate one as the entry point. Otherwise, a compile-time error occurs. The compilation environment may also create an entry point method if one does not exist.

When a program begins, if the entry point has a parameter, the argument supplied by the execution environment contains the command-line arguments to the program represented as strings. If the entry point has a return type of `Integer`, then the value returned from the function is returned to the execution environment as the result of the program.

In all other respects, entry point methods behave in the same manner as other methods. When execution leaves the invocation of the entry point method made by the execution environment, the program terminates.

See Also

6. Source Files and Namespaces | 6.2 Compilation Options | 6.3 Imports Statement | 6.4 Namespaces

6.2 Compilation Options

A source file can specify compilation options in the source code using *option statements*. An **Option** statement applies only to the source file in which it appears, and only one of each type of **Option** statement may appear in a source file. For example:

```
Option Strict On
Option Compare Text
Option Strict Off    ' Not allowed, Option Strict is already specified.
Option Compare Text  ' Not allowed, Option Compare is already specified.
```

There are three compilation options: strict type semantics, explicit declaration semantics, and comparison semantics. If a source file does not include a particular **Option** statement, then the compilation environment determines which particular set of semantics will be used. There is also a fourth compilation option, integer overflow 0checks, which can only be specified through the compilation environment.

```
OptionStatement ::=
   OptionExplicitStatement |
   OptionStrictStatement |
   OptionCompareStatement
```

See Also

6.2.1 Option Explicit Statement | 6.2.2 Option Strict Statement | 6.2.3 Option Compare Statement | 6.3 Imports Statement | 6.4 Namespaces | Visual Basic Compiler Options (Visual Basic Language Reference)

6.2.1 Option Explicit Statement

The **Option Explicit** statement determines whether local variables may be implicitly declared. The keywords **On** or **Off** may follow the statement; if neither is specified, the default is **On**. If no statement is specified in a file, the compilation environment determines which will be used.

> **Note** **Explicit** and **Off** are not reserved words.

```
Option Explicit Off

Module Test
    Sub Main()
        x = 5 ' Valid because Option Explicit is off.
    End Sub
End Module
```

In this example, the local variable `x` is implicitly declared by assigning to it. The type of `x` is `Object`.

```
OptionExplicitStatement ::= Option Explicit [ OnOff ] LineTerminator
OnOff ::= On | Off
```

See Also

6.2.2 Option Strict Statement | 6.2.3 Option Compare Statement | 6.2.4 Integer Overflow Checks | 6.2 Compilation Options | 10.1.1 Local Variables and Parameters | Option Explicit Statement (Visual Basic Language Reference)

6.2.2 Option Strict Statement

The **Option Strict** statement determines whether conversions and operations on **Object** are governed by strict or permissive type semantics and whether types are implicitly typed as **Object** if no **As** clause is specified. The statement may be followed by the keywords **On** or **Off**; if neither is specified, the default is **On**. If no statement is specified in a file, the compilation environment determines which will be used.

> **Note** **Strict** and **Off** are not reserved words.

```
Option Strict On

Module Test
    Sub Main()
        Dim x ' Error, no type specified.
        Dim o As Object
        Dim b As Byte = o ' Error, narrowing conversion.

        o.Foo() ' Error, late binding disallowed.
        o = o + 1 ' Error, addition is not defined on Object.
    End Sub
End Module
```

Under strict semantics, the following are disallowed:

- Narrowing conversions without an explicit cast operator.
- Late binding.
- Operations on type **Object** other than **=**, **<>**, **TypeOf**...**Is**, and **Is**.
- Omitting the **As** clause in a declaration.

```
OptionStrictStatement ::= Option Strict [ OnOff ] LineTerminator
```

See Also

6.2.1 Option Explicit Statement | 6.2.3 Option Compare Statement | 6.2.4 Integer Overflow Checks | 6.2 Compilation Options | Option Strict Statement (Visual Basic Language Reference)

6.2.3 Option Compare Statement

The **Option Compare** statement determines the semantics of string comparisons. String comparisons are carried out either using binary comparisons (in which the binary Unicode value of each character is compared) or text comparisons (in which the lexical meaning of each character is compared using the current culture). If no statement is specified in a file, the compilation environment controls which type of comparison will be used.

> **Note** **Compare**, **Binary**, and **Text** are not reserved words.

```
Option Compare Text

Imports System

Module M
    Sub Main()
        Console.WriteLine("a" = "A")    ' Prints True.
    End Sub
End Module
```

In this case, the string comparison is done using a text comparison that ignores case differences. If **Option Compare Binary** had been specified, then this would have printed **False**.

```
OptionCompareStatement ::= Option Compare CompareOption LineTerminator
CompareOption ::= Binary | Text
```

See Also

6.2.1 Option Explicit Statement | 6.2.2 Option Strict Statement | 6.2.4 Integer Overflow Checks | 6.2 Compilation Options | 11.15 Like Operator | Option Compare Statement (Visual Basic Language Reference)

6.2.4 Integer Overflow Checks

Integer operations can either be checked or not checked for overflow conditions at run time. If overflow conditions are checked and an integer operation overflows, a **System.OverflowException** exception is thrown. If overflow conditions are not checked, integer operation overflows do not throw an exception. The compilation environment determines whether this option is on or off.

See Also

6.2.1 Option Explicit Statement | 6.2.2 Option Strict Statement | 6.2.3 Option Compare Statement | 6.2 Compilation Options | 11.12 Operator Expressions | Visual Basic Compiler Options (Visual Basic Language Reference)

6.3 Imports Statement

Imports statements import the names of entities into a source file, allowing the names to be referenced without qualification.

Within member declarations in a source file that contains an **Imports** statement, the types contained in the given namespace can be referenced directly, as seen in the following example:

```
Imports N1.N2

Namespace N1.N2
    Class A
    End Class
End Namespace

Namespace N3
    Class B
        Inherits A
    End Class
End Namespace
```

Here, within the source file, the type members of namespace `N1.N2` are directly available, and thus class `N3.B` derives from class `N1.N2.A`.

Imports statements must appear after any **Option** statements but before any type declarations. The compilation environment may also define implicit **Imports** statements.

Imports statements make names available in a source file, but do not declare anything in the global namespace's declaration space. The scope of the names imported by an **Imports** statement extends over the namespace member declarations contained in the source file. The scope of an **Imports** statement specifically does not include other **Imports** statements, nor does it include other source files. **Imports** statements may not refer to one another.

In this example, the last **Imports** statement is in error because it is not affected by the first import alias.

```
Imports R1 = N1 ' OK.
Imports R2 = N1.N2 ' OK.
Imports R3 = R1.N2 ' Error: Can't refer to R1.

Namespace N1.N2
End Namespace
```

> **Note** The namespace or type names that appear in **Imports** statements are always treated as if they are fully qualified. That is, the leftmost identifier in a namespace or type name always resolves in the global namespace and the rest of the resolution proceeds according to normal name resolution rules. This is the only place in the language that applies such a rule; the rule ensures that a name cannot be completely hidden from qualification. Without the rule, if a name in the global namespace were hidden in a particular source file, it would be impossible to specify any names from that namespace in a qualified way.

In this example, the **Imports** statement always refers to the global `System` namespace, and not the class in the source file.

```
Imports System   ' Imports the namespace, not the class.

Class System
End Class
```

```
ImportsStatement ::= Imports ImportsClauses LineTerminator
ImportsClauses ::=
   ImportsClause |
   ImportsClauses , ImportsClause
ImportsClause ::= ImportsAliasClause | ImportsNamespaceClause
```

See Also

6.3.1 Import Aliases | 6.3.2 Namespace Imports | 6.2 Compilation Options |6.4 Namespaces |
Imports Statement (Visual Basic Language Reference) | References and the Imports Statement (Visual Basic Language Concepts)

6.3.1 Import Aliases

An *import alias* defines an alias for a namespace or type.

```
Imports A = N1.N2.A

Namespace N1.N2
    Class A
    End Class
End Namespace

Namespace N3
    Class B
        Inherits A
    End Class
End Namespace
```

Here, within the source file, `A` is an alias for `N1.N2.A`, and thus class `N3.B` derives from class `N1.N2.A`. The same effect can be obtained by creating an alias `R` for `N1.N2` and then referencing `R.A`:

```
Imports R = N1.N2

Namespace N3
    Class B
        Inherits R.A
    End Class
End Namespace
```

The identifier of an import alias must be unique within the declaration space of the global namespace (not just the global namespace declaration in the source file in which the import alias is defined), even though it does not declare a name in the global namespace's declaration space.

```
Imports A = N3.A

Class A
End Class

Namespace N3
    Class A
    End Class
End Namespace
```

Here, the global namespace already contains a member `A`, so it is an error for an import alias to use that identifier. It is likewise an error for two or more import aliases in the same source file to declare aliases by the same name.

An import alias can create an alias for any namespace or type. Accessing a namespace or type through an alias yields exactly the same result as accessing the namespace or type through its declared name.

```
Imports R1 = N1
Imports R2 = N1.N2

Namespace N1.N2
    Class A
    End Class
End Namespace

Namespace N3
    Class B
        Private a As N1.N2.A
        Private b As R1.N2.A
        Private c As R2.A
    End Class
End Namespace
```

Here, the names `N1.N2.A`, `R1.N2.A`, and `R2.A` are equivalent, and all refer to the class whose fully qualified name is `N1.N2.A`.

Declarations in the source file may shadow the import alias name.

```
Imports R = N1.N2

Namespace N1.N2
    Class A
    End Class
End Namespace

Namespace N3
    Class R
    End Class

    Class B
        Inherits R.A  ' Error, R has no member A
    End Class
End Namespace
```

In the preceding example the reference to `R.A` in the declaration of `B` causes an error because `R` refers to `N3.R`, not `N1.N2`.

An import alias makes an alias available within a particular source file, but it does not contribute any new members to the underlying declaration space. In other words, an import alias is not transitive, but rather affects only the source file in which it occurs.

File1.vb:

```
Imports R = N1.N2

Namespace N1.N2
    Class A
    End Class
End Namespace
```

File2.vb:

```
Class B
    Inherits R.A ' Error, R unknown.
End Class
```

In the above example, because the scope of the import alias that introduces `R` only extends to declarations in the source file in which it is contained, `R` is unknown in the second source file.

```
ImportsAliasClause ::= Identifier = NamespaceOrTypeName
```

See Also

6.3.2 Namespace Imports | 6.3 Imports Statement | Imports Statement (Visual Basic Language Reference) | References and the Imports Statement (Visual Basic Language Concepts)

6.3.2 Namespace Imports

A *namespace import* imports all of the members of a namespace or type, allowing the identifier of each member of the namespace or type to be used without qualification. In the case of types, a namespace import only allows access to the shared members of the type without requiring qualification of the class name. In particular, it allows the members of enumerated types to be used without qualification. For example:

```
Imports Colors

Enum Colors
    Red
    Green
    Blue
End Enum

Module M1
    Sub Main()
        Dim c As Colors = Red
    End Sub
End Module
```

Unlike an import alias, a namespace import has no restrictions on the names it imports and may import namespaces and types whose identifiers are already declared within the global namespace. The names imported by a regular import are shadowed by import aliases and declarations in the source file.

In the following example, `A` refers to `N3.A` rather than `N1.N2.A` within member declarations in the `N3` namespace.

```
Imports N1.N2

Namespace N1.N2
    Class A
    End Class
End Namespace

Namespace N3
    Class A
    End Class

    Class B
        Inherits A
    End Class
End Namespace
```

When more than one imported namespace contains members by the same name (and that name is not otherwise shadowed by an import alias or declaration), a reference to that name is ambiguous and causes a compile-time error.

```
Imports N1
Imports N2

Namespace N1
    Class A
    End Class
End Namespace

Namespace N2
    Class A
    End Class
End Namespace

Namespace N3
    Class B
        Inherits A ' Error, A is ambiguous.
    End Class
End Namespace
```

In the above example, both `N1` and `N2` contain a member `A`. Because `N3` imports both, referencing `A` in `N3` causes a compile-time error. In this situation, the conflict can be resolved either through qualification of references to `A`, or by introducing an import alias that picks a particular `A`, as in the following example:

```
Imports N1
Imports N2
Imports A = N1.A

Namespace N3
    Class B
        Inherits A ' A means N1.A.
    End Class
End Namespace
```

Only namespaces, classes, structures, enumerated types, and standard modules may be imported.

```
ImportsNamespaceClause ::= NamespaceOrTypeName
```

See Also

6.3.1 Import Aliases | 6.3 Imports Statement | Imports Statement (Visual Basic Language Reference) | References and the Imports Statement (Visual Basic Language Concepts)

6.4 Namespaces

Visual Basic .NET programs are organized using namespaces. Namespaces both internally organize a program as well as organize the way program elements are exposed to other programs.

Unlike other entities, namespaces are open-ended, and may be declared multiple times within the same program and across many programs, with each declaration contributing members to the same namespace. In the following example, the two namespace declarations contribute to the same declaration space, declaring two classes with the fully qualified names `N1.N2.A` and `N1.N2.B`.

```
Namespace N1.N2
    Class A
    End Class
End Namespace

Namespace N1.N2
    Class B
    End Class
End Namespace
```

Because the two declarations contribute to the same declaration space, it would be an error if each contained a declaration of a member with the same name.

There is a global namespace that has no name and whose nested namespaces and types can always be accessed without qualification. The scope of a namespace member declared in the global namespace is the entire program text. Otherwise, the scope of a type or namespace declared in a namespace whose fully qualified name is **N** is the program text of each namespace whose corresponding namespace's fully qualified name begins with **N** or is **N** itself.

In this example, the class `B` can see the class `A` because `B`'s namespace `N1.N2.N3` is conceptually nested within the namespace `N1.N2`.

```
Namespace N1.N2
    Class A
    End Class
End Namespace

Namespace N1.N2.N3
    Class B
        Inherits A
    End Class
End Namespace
```

See Also

6.4.1 Namespace Declarations | 6.4.2 Namespace Members | 6.2 Compilation Options | 6.3 Imports Statement | Namespace Statement (Visual Basic Language Reference) | Namespaces (Visual Basic Language Concepts)

6.4.1 Namespace Declarations

A namespace declaration consists of the keyword **Namespace** followed by a qualified identifier and optional namespace member declarations. If the namespace name is qualified, the namespace declaration is treated as if it is lexically nested within namespace declarations corresponding to each name in the qualified name. For example, the following two namespaces are semantically equivalent:

```
Namespace N1.N2
    Class A
    End Class

    Class B
    End Class
End Namespace

Namespace N1
    Namespace N2
        Class A
        End Class

        Class B
        End Class
    End Namespace
End Namespace
```

When dealing with the members of a namespace, it is not important where a particular member is declared. If two programs define an entity with the same name in the same namespace, attempting to resolve the name in the namespace causes an ambiguity error.

Namespaces are by definition **Public**, so a namespace declaration cannot include any access modifiers.

```
NamespaceDeclaration ::=
   Namespace QualifiedIdentifier LineTerminator
   [ NamespaceMemberDeclaration+ ]
   End Namespace LineTerminator
```

See Also

6.4.2 Namespace Members | 6.4 Namespaces | Namespace Statement (Visual Basic Language Reference) | Namespaces (Visual Basic Language Concepts)

6.4.2 Namespace Members

Namespace members can only be namespace declarations and type declarations. Type declarations may have **Public** or **Friend** access. The default access for types is **Friend** access.

```
NamespaceMemberDeclaration ::=
   NamespaceDeclaration |
   TypeDeclaration
TypeDeclaration ::=
   ModuleDeclaration |
   NonModuleDeclaration
NonModuleDeclaration ::=
   EnumDeclaration |
   StructureDeclaration |
   InterfaceDeclaration |
   ClassDeclaration |
   DelegateDeclaration
```

See Also

6.4.1 Namespace Declarations | 6.4 Namespaces | Namespace Statement | Namespaces (Visual Basic Language Concepts)

7. Types

The two fundamental categories of types in Visual Basic .NET are *value types* and *reference types*. Primitive types (except strings), enumerations, and structures are value types. Classes, strings, standard modules, interfaces, arrays, and delegates are reference types.

With two exceptions, all types are either value types or reference types.

- The root type **Object**, which is an alias for **System.Object**, is a **MustInherit** type that is neither a reference type nor a value type. Thus, a variable of type **Object** can either contain a value type or a reference type.
- The type **System.Void**, which represents a void type in the .NET Framework, can be referenced only in **GetType** expressions.

Every type has a *default value*, which is the value that is assigned to variables of that type upon initialization.

```
TypeName ::=
   NamespaceOrTypeName |
   BuiltInTypeName |
   ArrayTypeName
BuiltInTypeName ::= Object | PrimitiveTypeName
```

See Also

7.1 Value Types and Reference Types | 7.3 Primitive Types | 7.4 Enumerations | 7.6 Structures | 7.5 Classes | 7.7 Standard Modules | 4.3.2 Interfaces and Multiple Inheritance | 7.9 Arrays | 7.10 Delegates | Data Type Summary (Visual Basic Language Reference) | Value Types and Reference Types (Visual Basic Language Concepts)

7.1 Value Types and Reference Types

Although value types and reference types can be similar in terms of declaration syntax and usage, their semantics are distinct.

Reference types are stored on the run-time heap; they may only be accessed through a reference to that storage. Because reference types are always accessed through references, their lifetime is managed by the .NET Framework. Outstanding references to a particular instance are tracked and the instance is destroyed only when no more references remain. A variable of reference type contains a reference to a value of that type, a value of a more derived type, or a null reference. A *null reference* is a reference that refers to nothing; it is not possible to do anything with a null reference except assign it. Assignment to a variable of a reference type creates a copy of the reference rather than a copy of the referenced value. For a variable of a reference type, the default value is a null reference.

Value types are stored directly on the stack, either within an array or within another type; their storage can only be accessed directly. Because value types are stored directly within variables, their lifetime is determined by the lifetime of the variable that contains them. When the location containing a value type instance is destroyed, the value type instance is also destroyed. Value types are always accessed directly; it is not possible to create a reference to a value type. Prohibiting such a reference makes it impossible to refer to a value class instance that has been destroyed. Because value types are always **NotInheritable**, a variable of a value type always contains a value of that type. Because of this, the value of a value type cannot be a null reference, nor can it reference an object of a more derived type. Assignment to a variable of a value type creates a copy of the value being assigned. For a variable of a value type, the default value is the result of initializing each variable member of the type to its default value.

The following example shows the difference between reference types and value types:

```
Imports System

Class Class1
    Public Value As Integer = 0
End Class

Module Test
    Sub Main()
        Dim val1 As Integer = 0
        Dim val2 As Integer = val1
        val2 = 123
        Dim ref1 As Class1 = New Class1()
        Dim ref2 As Class1 = ref1
        ref2.Value = 123
        Console.WriteLine("Values: " & val1 & ", " & val2)
        Console.WriteLine("Refs: " & ref1.Value & ", " & ref2.Value)
    End Sub
End Module
```

The output of the program is:

```
Values: 0, 123
Refs: 123, 123
```

The assignment to the local variable `val2` does not impact the local variable `val1` because both local variables are of a value type (the type **Integer**) and each local variable of a value type has its own storage. In contrast, the assignment `ref2.Value = 123;` affects the object that both `ref1` and `ref2` reference.

Value types and reference types are unified under the type **Object**, which is the root type of all types. Because **Object** is neither a reference nor a value type, it can contain values of either kind and has semantics consistent with the kind of type contained within it at run time.

The following example shows how **Object** can be used with both value types and reference types:

```
Imports System

Class Class1
    Public Value As Integer = 0
End Class

Structure Struct1
```

```
    Public Value As Integer
End Structure

Module Test
    Sub Main()
        Dim val1 As Object = New Struct1()
        Dim val2 As Object = val1

        val2.Value = 123

        Dim ref1 As Object = New Class1()
        Dim ref2 As Object = ref1

        ref2.Value = 123

        Console.WriteLine("Values: " & val1.Value & ", " & val2.Value)
        Console.WriteLine("Refs: " & ref1.Value & ", " & ref2.Value)
    End Sub
End Module
```

The output of the program is:

```
Values: 0, 123
Refs: 123, 123
```

The assignment to the field of the local variable `val2` does not impact the field of the local variable `val1` because both local variables are of a value type (the type **Integer**) and each local variable of a value type has its own storage. In contrast, the assignment `ref2.Value = 123` affects the object that both `ref1` and `ref2` references.

See Also

7.3 Primitive Types | 7.4 Enumerations | 7.6 Structures | 7.5 Classes | 7.7 Standard Modules | 7.8 Interfaces | 7.9 Arrays | 7.10 Delegates | Data Type Summary (Visual Basic Language Reference) | Value Types and Reference Types (Visual Basic Language Concepts)

7.2 Interface Implementation

Structure and class declarations may declare that they implement a set of interface types through one or more **Implements** clauses. All the types specified in the **Implements** clause must be interfaces, and the type must implement all members of the interfaces. For example:

```
Interface ICloneable
    Function Clone() As Object
End Interface

Interface IComparable
    Function CompareTo(ByVal other As Object) As Integer
End Interface

Structure ListEntry
    Implements ICloneable, IComparable

    Public Function Clone() As Object Implements ICloneable.Clone
    End Function

    Public Function CompareTo(ByVal other As Object) As Integer _
        Implements IComparable.CompareTo
    End Function
End Structure
```

A type that implements an interface also implicitly implements all of the interface's base interfaces. This is true even if the type does not explicitly list all base interfaces in the **Implements** clause. In this example, the `TextBox` structure implements both `IControl` and `ITextBox`.

```
Interface IControl
    Sub Paint()
End Interface

Interface ITextBox
    Inherits IControl
    Sub SetText(ByVal Text As String)
End Interface

Structure TextBox
    Implements ITextBox

    Public Sub Paint() Implements ITextBox.Paint
    End Sub

    Public Sub SetText(ByVal Text As String) Implements ITextBox.SetText
    End Sub
End Structure
```

Declaring that a type implements an interface in and of itself does not declare anything in the declaration space of the type. Thus, it is valid to implement two interfaces with a method by the same name.

```
TypeImplementsClause ::= Implements Implements LineTerminator
Implements ::=
   TypeName |
   Implements , TypeName
```

See Also

7.5.1 Class Base Specification | 7.5.2 Class Members | 7.5 Classes | 7.6.1 Structure Members | 7.6 Structures | Implements Statement (Visual Basic Language Reference) | Implements Keyword (Visual Basic Language Concepts)

7.3 Primitive Types

The *primitive types* are identified through keywords, which are aliases for predefined types in the **System** namespace. A primitive type is completely indistinguishable from the type it aliases: writing the reserved word **Byte** is exactly the same as writing **System.Byte**.

Because a primitive type aliases a regular type, every primitive type has members. For example, **Integer** has the members declared in **System.Int32**. Literals can be treated as instances of their corresponding types.

The primitive types differ from other structure types in that they permit certain additional operations:

- Primitive types permit values to be created by writing literals. For example, `123I` is a literal of type **Integer**.
- It is possible to declare constants of the primitive types.
- When the operands of an expression are all primitive type constants, it is possible for the compiler to evaluate the expression at compile time. Such an expression is known as a constant expression.

Visual Basic .NET defines the following primitive types:

- The integral value types **Byte** (1-byte unsigned integer), **Short** (2-byte signed integer), **Integer** (4-byte signed integer), and **Long** (8-byte signed integer). These types map to **System.Byte**, **System.Int16**, **System.Int32**, and **System.Int64**, respectively. The default value of an integral type is equivalent to the literal `0`.
- The floating-point value types **Single** (4-byte floating point) and **Double** (8-byte floating point). These types map to **System.Single** and **System.Double**, respectively. The default value of a floating-point type is equivalent to the literal `0`.
- The **Decimal** type (16-byte decimal value), which maps to **System.Decimal**. The default value of decimal is equivalent to the literal `0D`.
- The **Boolean** value type, which represents a truth value, typically the result of a relational or logical operation. The literal is of type **System.Boolean**. The default value of the **Boolean** type is equivalent to the literal **False**.
- The **Date** value type, which represents a date and/or a time and maps to **System.DateTime**. The default value of the **Date** type is equivalent to the literal `# 01/01/0001 12:00:00AM #`.
- The **Char** value type, which represents a single Unicode character and maps to **System.Char**. The default value of the **Char** type is equivalent to the constant expression `ChrW(0)`.
- The **String** reference type, which represents a sequence of Unicode characters and maps to **System.String**. The default value of the **String** type is a null reference.

```
PrimitiveTypeName ::= NumericTypeName | Boolean | Date | Char | String
NumericTypeName ::= IntegralTypeName | FloatingPointTypeName | Decimal
IntegralTypeName ::= Byte | Short | Integer | Long
FloatingPointTypeName ::= Single | Double
```

See Also

7.1 Value Types and Reference Types | 7.6 Structures | 7.5 Classes | 7.7 Standard Modules | 7.8 Interfaces | 7.9 Arrays | 7.10 Delegates | 11.2 Constant Expressions | Data Type Summary (Visual Basic Language Reference) | Value Types and Reference Types (Visual Basic Language Concepts)

7.4 Enumerations

Enumerations are value types that inherit from **System.Enum** and symbolically represent a set of values of one of the primitive integral types. For an enumeration type `E`, the default value is the value produced by the expression `CType(0, E)`.

The underlying type of an enumeration must be an integral type that can represent all the enumerator values defined in the enumeration. If an underlying type is specified, it must be **Byte**, **Short**, **Integer**, or **Long**. If no underlying type is explicitly specified, the default is **Integer**.

The following example declares an enumeration with an underlying type of **Long**:

```
Enum Color As Long
    Red
    Green
    Blue
End Enum
```

A developer might choose to use an underlying type of **Long**, as in the example, to enable the use of values that are in the range of **Long**, but not in the range of **Integer**, or to preserve this option for the future.

```
EnumDeclaration ::=
   [ Attributes ] [ EnumModifier+ ] Enum Identifier
      [ As IntegralTypeName ] LineTerminator
   EnumMemberDeclaration+
   End Enum LineTerminator
EnumModifier ::= AccessModifier | Shadows
```

See Also

7.4.1 Enumeration Members | 7.4.2 Enumeration Values | 7.1 Value Types and Reference Types | 7.3 Primitive Types | 7.6 Structures | 7.5 Classes | 7.7 Standard Modules | 7.8 Interfaces | 7.9 Arrays | 7.10 Delegates | Enum Statement (Visual Basic Language Reference) | Enumerations Overview(Visual Basic Language Concepts)

7.4.1 Enumeration Members

The members of an enumeration are the enumerated values declared in the enumeration and the members inherited from class **System.Enum**.

The scope of an enumeration member is the enumeration declaration body. This means that outside of an enumeration declaration, an enumeration member must always be qualified (unless the type is specifically imported into a namespace through a namespace import).

Declaration order for enumeration member declarations is significant when constant expression values are omitted. Enumeration members implicitly have **Public** access only; no access modifiers are allowed on enumeration member declarations.

```
EnumMemberDeclaration ::=
   [ Attributes ] Identifier [ = ConstantExpression ] LineTerminator
```

See Also

7.4.2 Enumeration Values | 7.4 Enumerations | Enum Statement (Visual Basic Language Reference) | Enumerations Overview(Visual Basic Language Concepts)

7.4.2 Enumeration Values

The enumerated values in an enumeration member list are declared as constants typed as the underlying enumeration type, and they can appear wherever constants are required. An enumeration member definition with **=** gives the associated member the value indicated by the constant expression. The constant expression must evaluate to an integral type that is implicitly convertible to the underlying type and must be within the range of values that can be represented by the underlying type. The following example is in error because the constant values `1.5`, `2.3`, and `3.3` are not implicitly convertible to the underlying integral type **Long** with strict semantics.

```
Option Strict On

Enum Color As Long
    Red = 1.5
    Green = 2.3
    Blue = 3.3
End Enum
```

Multiple enumeration members may share the same associated value, as shown below:

```
Enum Color
    Red
    Green
    Blue
    Max = Blue
End Enum
```

The example shows an enumeration that has two enumeration members — `Blue` and `Max` — that have the same associated value.

If the first enumerator value definition in the enumeration has no initializer, the value of the corresponding constant is `0`. An enumeration value definition without an initializer gives the enumerator the value obtained by increasing the value of the previous enumeration value by `1`. This increased value must be within the range of values that can be represented by the underlying type.

```
Imports System

Enum Color
    Red
    Green = 10
    Blue
End Enum

Module Test
    Sub Main()
        Console.WriteLine(StringFromColor(Color.Red))
        Console.WriteLine(StringFromColor(Color.Green))
        Console.WriteLine(StringFromColor(Color.Blue))
    End Sub

    Function StringFromColor(ByVal c As Color) As String
        Select Case c
            Case Color.Red
                Return String.Format("Red = " & CInt(c))

            Case Color.Green
                Return String.Format("Green = " & CInt(c))

            Case Color.Blue
                Return String.Format("Blue = " & CInt(c))

            Case Else
                Return "Invalid color"
        End Select
    End Function
End Class
```

The example above prints the enumeration values and their associated values. The output is:

```
Red = 0
Blue = 11
Green = 10
```

The reasons for the values are as follows:

- The enumeration value `Red` is automatically assigned the value `0` (since it has no initializer and is the first enumeration value member).
- The enumeration value `Green` is explicitly given the value `10`.
- The enumeration value `Blue` is automatically assigned the value one greater than the enumeration value that textually precedes it.

The constant expression may not directly or indirectly use the value of its own associated enumeration value (that is, circularity in the constant expression is not allowed). The following example is invalid because the declarations of `A` and `B` are circular.

```
Enum Circular
    A = B
    B
End Enum
```

`A` depends on `B` explicitly, and `B` depends on `A` implicitly.

See Also

7.4.1 Enumeration Members | 7.4 Enumerations | Enum Statement (Visual Basic Language Reference) | Enumerations Overview(Visual Basic Language Concepts)

7.5 Classes

A *class* is a data structure that may contain data members (constants, variables, and events), function members (methods, properties, indexers, operators, and constructors), and nested types. Classes are reference types. The following example shows a class that contains each kind of member:

```
Imports System

Class AClass

    Public Sub New()
        Console.WriteLine("Constructor")
    End Sub

    Public Sub New(ByVal value As Integer)
        MyVariable = value
        Console.WriteLine("Constructor")
    End Sub

    Public Const MyConst As Integer = 12
    Public MyVariable As Integer = 34

    Public Sub MyMethod()
        Console.WriteLine("MyClass.MyMethod")
    End Sub

    Public Property MyProperty() As Integer
        Get
            Return MyVariable
        End Get

        Set (ByVal value As Integer)
            MyVariable = value
        End Set
    End Property

    Default Public Property Item(ByVal index As Integer) As Integer
        Get
            Return 0
        End Get

        Set (ByVal value As Integer)
            Console.WriteLine("Item(" & index & ") = " & value)
        End Set
    End Property

    Public Event MyEvent()

    Friend Class MyNestedClass
    End Class
End Class
```

The following example shows uses of these members:

```
Imports System

Module Test

    ' Event usage.
    Dim WithEvents aInstance As AClass

    Sub Main()
        ' Constructor usage.
        Dim a As AClass = New AClass()
        Dim b As AClass = New AClass(123)
```

```
        ' Constant usage.
        Console.WriteLine("MyConst = " & AClass.MyConst)

        ' Variable usage.
        a.MyVariable += 1
        Console.WriteLine("a.MyVariable = " & a.MyVariable)

        ' Method usage.
        a.MyMethod()

        ' Property usage.
        a.MyProperty += 1
        Console.WriteLine("a.MyProperty = " & a.MyProperty)
        a(1) = 1

        ' Event usage.
        aInstance = a
    End Sub

    Sub MyHandler() Handles aInstance.MyEvent
        Console.WriteLine("Test.MyHandler")
    End Sub
End Module
```

There are two class-specific modifiers, **MustInherit** and **NotInheritable**. It is invalid to specify them both.

```
ClassDeclaration ::=
   [ Attributes ] [ ClassModifier+ ] Class Identifier LineTerminator
   [ ClassBase ]
   [ TypeImplementsClause+ ]
   [ ClassMemberDeclaration+ ]
   End Class LineTerminator
ClassModifier ::= AccessModifier | Shadows | MustInherit | NotInheritable
```

See Also

7.5.1 Class Base Specification | 7.2 Interface Implementation | 7.5.2 Class Members | 7.1 Value Types and Reference Types | 7.3 Primitive Types | 7.4 Enumerations | 7.6 Structures | 7.7 Standard Modules | 7.8 Interfaces | 7.9 Arrays | 7.10 Delegates | Class Statement (Visual Basic Language Reference) | Classes: Blueprints for Objects (Visual Basic Language Concepts)

7.5.1 Class Base Specification

A class declaration may include a base type specification that defines the direct base type of the class. If a class declaration has no explicit base type, the direct base type is implicitly **Object**. For example:

```
Class Base
End Class

Class Derived
    Inherits Base
End Class
```

Classes may only derive from **Object** and classes. It is invalid for a class to derive from **System.ValueType**, **System.Enum**, **System.Array**, or **System.Delegate**.

Every class has exactly one direct base class, and circularity in derivation is prohibited. It is not possible to derive from a **NotInheritable** class, and the accessibility domain of the base class must be the same as or a superset of the accessibility domain of the class itself.

```
ClassBase ::= Inherits TypeName LineTerminator
```

See Also

7.2 Interface Implementation | 7.5.2 Class Members | 7.5 Classes | Inherits Statement (Visual Basic Language Reference)

7.5.2 Class Members

The members of a class consist of the members introduced by its class member declarations and the members inherited from its direct base class.

A class member declaration may have **Public**, **Protected**, **Friend**, **Protected Friend**, or **Private** access. When a class member declaration does not include an access modifier, the declaration defaults to **Public** access, unless it is a variable declaration; in that case it defaults to **Private** access.

The scope of a class member is the class body in which the declaration occurs. If the member has **Friend** access, its scope extends to the class body of any derived class in the same program, and if the member has **Public**, **Protected**, or **Protected Friend** access, its scope extends to the class body of any derived class in any program.

```
ClassMemberDeclaration ::=
   NonModuleDeclaration |
   EventMemberDeclaration |
   VariableMemberDeclaration |
   ConstantMemberDeclaration |
   MethodMemberDeclaration |
   PropertyMemberDeclaration |
   ConstructorMemberDeclaration
```

See Also

7.5.1 Class Base Specification | 7.2 Interface Implementation | 7.5 Classes | Class Statement (Visual Basic Language Reference) | Understanding Classes (Visual Basic Language Concepts)

7.6 Structures

Structures are value types that inherit from **System.ValueType**. Structures are similar to classes in that they represent data structures that can contain data members and function members. Unlike classes, however, structures do not require heap allocation.

In the case of classes, it is possible for two variables to reference the same object, and thus possible for operations on one variable to affect the object referenced by the other variable. With structures, the variables each have their own copy of the data, so it is not possible for operations on one to affect the other, as the following example illustrates:

```
Structure Point
    Public x, y As Integer

    Public Sub New(ByVal x As Integer, ByVal y As Integer)
        Me.x = x
        Me.y = y
    End Sub
End Structure
```

Given the above declaration the following code fragment outputs the value `10`:

```
Point a = new Point(10, 10)
Point b = a

a.x = 100
System.Console.WriteLine(b.x)
```

The assignment of `a` to `b` creates a copy of the value, and `b` is thus unaffected by the assignment to `a.x`. Had `Point` instead been declared as a class, the output would be `100` because `a` and `b` would reference the same object.

```
StructureDeclaration ::=
   [ Attributes ] [ StructureModifier+ ] Structure Identifier
      LineTerminator
   [ TypeImplementsClause+ ]
   [ StructMemberDeclaration+ ]
   End Structure LineTerminator
StructureModifier ::= AccessModifier | Shadows
```

See Also

7.2 Interface Implementation | 7.6.1 Structure Members | 7.1 Value Types and Reference Types | 7.3 Primitive Types | 7.4 Enumerations | 7.5 Classes | 7.7 Standard Modules | 7.8 Interfaces | 7.9 Arrays | 7.10 Delegates | Structure Statement (Visual Basic Language Reference) | Structures: Your Own Data Types (Visual Basic Language Concepts)

7.6.1 Structure Members

The members of a structure are the members introduced by its structure member declarations and the members inherited from **System.ValueType**. Structures must contain at least one instance variable.

Every structure implicitly has a **Public** parameterless instance constructor that produces the default value of the structure. As a result, it is not possible for a structure type declaration to declare a parameterless instance constructor. A structure type is, however, permitted to declare *parameterized* instance constructors, as in the following example:

```
Structure Point
    Private x, y As Integer

    Public Sub New(ByVal x As Integer, ByVal y As Integer)
        Me.x = x
        Me.y = y
    End Sub
End Structure
```

Given the above declaration, the following statements both create a `Point` with `x` and `y` initialized to zero.

```
Dim p1 As Point = New Point()
Dim p2 As Point = New Point(0, 0)
```

Normally, a structure member declaration may only have **Public**, **Friend**, or **Private** access, but when overriding members inherited from **Object**, **Protected** and **Protected Friend** access may also be used. When a structure member declaration does not include an access modifier, the declaration defaults to **Public** access. The scope of a member declared by a structure is the structure body in which the declaration occurs.

```
StructMemberDeclaration ::=
   NonModuleDeclaration |
   VariableMemberDeclaration |
   ConstantMemberDeclaration |
   EventMemberDeclaration |
   MethodMemberDeclaration |
   PropertyMemberDeclaration |
   ConstructorMemberDeclaration
```

See Also

7.2 Interface Implementation | 7.6 Structures | Structure Statement (Visual Basic Language Reference) | Structures: Your Own Data Types (Visual Basic Language Concepts)

7.7 Standard Modules

A *standard module* is a type whose members are implicitly **Shared** and scoped to the declaration space of the standard module's containing namespace, rather than just to the standard module declaration itself. Standard modules may never be instantiated. It is an error to declare a variable of a standard module type.

A member of a standard module has two fully qualified names, one without the standard module name and one with the standard module name. More than one standard module in a namespace may define a member with a particular name; unqualified references to the name outside of either module are ambiguous. For example:

```
Namespace N1
    Module M1
        Sub S1()
        End Sub

        Sub S2()
        End Sub
    End Module

    Module M2
        Sub S2()
        End Sub
    End Module

    Module M3
        Sub Main()
            S1() ' Valid: Calls N1.M1.S1.
            N1.S1() ' Valid: Calls N1.M1.S1.
            S2() ' Not valid: ambiguous.
            N1.S2() ' Not valid: ambiguous.
            N1.M2.S2() ' Valid: Calls N1.M2.S2.
        End Sub
    End Module
End Namespace
```

A module may only be declared in a namespace and may not be nested in another type. Standard modules may not implement interfaces, they implicitly derive from **Object**, and they have only **Shared** constructors.

```
ModuleDeclaration ::=
   [ Attributes ] [ AccessModifier+ ] Module Identifier LineTerminator
   [ ModuleMemberDeclaration+ ]
   End Module LineTerminator
```

See Also

7.7.1 Standard Module Members |7.1 Value Types and Reference Types | 7.3 Primitive Types | 7.4 Enumerations | 7.6 Structures | 7.5 Classes | 7.8 Interfaces | 7.9 Arrays | 7.10 Delegates | Module Statement (Visual Basic Language Reference) | Classes vs. Standard Modules (Visual Basic Language Concepts)

7.7.1 Standard Module Members

The members of a standard module are the members introduced by its member declarations and the members inherited from **Object**. Standard modules may have any type of member except instance constructors. All standard module type members are implicitly **Shared**.

Normally, a standard module member declaration may only have **Public**, **Friend**, or **Private** access, but when overriding members inherited from **Object**, the **Protected** and **Protected Friend** access modifiers may be specified. When a standard module member declaration does not include an access modifier, the declaration defaults to **Public** access, unless it is a variable, which defaults to **Private** access.

As previously noted, the scope of a standard module member is the declaration containing the standard module declaration. Members inherited from **Object** are not included in this special scoping; those members have no scope and must always be qualified with the name of the module. If the member has **Friend** access, its scope extends only to namespace members declared in the same program.

```
ModuleMemberDeclaration ::=
   NonModuleDeclaration |
   VariableMemberDeclaration |
   ConstantMemberDeclaration |
   EventMemberDeclaration |
   MethodMemberDeclaration |
   PropertyMemberDeclaration |
   ConstructorMemberDeclaration
```

See Also

7.7 Standard Modules | Module Statement (Visual Basic Language Reference) |
Shared Members (Visual Basic Language Concepts)

7.8 Interfaces

Interfaces are reference types that other types implement to guarantee that they support certain methods. An interface is never directly created and has no actual representation — other types must be converted to an interface type. An interface defines a contract. A class or structure that implements an interface must adhere to its contract.

The following example shows an interface that contains a default property `Item`, an event `E`, a method `F`, and a property `P`:

```
Interface IExample
    Default Property Item(ByVal index As Integer) As String
    Event E()
    Sub F(ByVal value As Integer)
    Property P() As String
End Interface
```

Interfaces may employ multiple inheritance. In the following example, the interface `IComboBox` inherits from both `ITextBox` and `IListBox`:

```
Interface IControl
    Sub Paint()
End Interface

Interface ITextBox
    Inherits IControl
    Sub SetText(ByVal Text As String)
End Interface

Interface IListBox
    Inherits IControl
    Sub SetItems(ByVal items() As String)
End Interface

Interface IComboBox
    Inherits ITextBox, IListBox
End Interface
```

Classes and structures can implement multiple interfaces. In the following example, the class `EditBox` derives from the class `Control` and implements both `IControl` and `IDataBound`:

```
Interface IDataBound
    Sub Bind(ByVal b As Binder)
End Interface

Public Class EditBox
    Inherits Control
    Implements IControl, IDataBound

    Public Sub Paint() Implements IControl.Paint
        ' ...
    End Sub

    Public Sub Bind(ByVal b As Binder) Implements IDataBound.Bind
    End Sub
End Class
```

```
InterfaceDeclaration ::=
   [ Attributes ] [ InterfaceModifier+ ] Interface Identifier
      LineTerminator
   [ InterfaceBase+ ]
   [ InterfaceMemberDeclaration+ ]
   End Interface LineTerminator
InterfaceModifier ::= AccessModifier | Shadows
```

See Also

7.8.1 Interface Inheritance | 7.8.2 Interface Members | | 7.3 Primitive Types | 7.4 Enumerations | 7.6 Structures | 7.5 Classes | 7.7 Standard Modules | 7.9 Arrays | 7.10 Delegates | Interface Statement (Visual Basic Language Reference) | Interfaces Overview (Visual Basic Language Concepts)

7.8.1 Interface Inheritance

The base interfaces of an interface are the explicit base interfaces and their base interfaces. In other words, the set of base interfaces is the complete transitive closure of the explicit base interfaces, their explicit base interfaces, and so on. If an interface declaration has no explicit interface base, then there is no base interface for the type. In the following example, the base interfaces of `IComboBox` are `IControl`, `ITextBox`, and `IListBox`.

```
Interface IControl
    Sub Paint()
End Interface

Interface ITextBox
    Inherits IControl
    Sub SetText(ByVal Text As String)
End Interface

Interface IListBox
    Inherits IControl
    Sub SetItems(ByVal items() As String)
End Interface

Interface IComboBox
    Inherits ITextBox, IListBox
End Interface
```

An interface inherits all members of its base interfaces. In other words, the `IComboBox` interface above inherits members `SetText` and `SetItems` as well as `Paint`.

A class or structure that implements an interface also implicitly implements all of the interface's base interfaces.

If an interface appears more than once in the transitive closure of the base interfaces, it only contributes its members to the derived interface once. A type implementing the derived interface only has to implement the methods of the multiply defined base interface once. In the following example, `Paint` only needs to be implemented once, even though the class implements `IComboBox` and `IControl`.

```
Class ComboBox
    Implements IControl, IComboBox

    Sub SetText(ByVal Text As String) Implements IComboBox.SetText
    End Sub

    Sub SetItems(ByVal items() As String) Implements IComboBox.SetItems
    End Sub

    Sub Print() Implements IComboBox.Paint
    End Sub
End Class
```

An **Inherits** clause has no effect on other **Inherits** clauses. In the following example, `IDerived` must qualify the name of `INested` with `IBase`.

```
Interface IBase
    Interface INested
        Sub Nested()
    End Interface

    Sub Base()
End Interface

Interface IDerived
    Inherits IBase, INested   ' Error: Must specify IBase.INested.
End Interface
```

The accessibility domain of a base interface must be the same as or a superset of the accessibility domain of the interface itself.

```
InterfaceBase ::= Inherits InterfaceBases LineTerminator
InterfaceBases ::=
   TypeName |
   InterfaceBases , TypeName
```

See Also

7.8.2 Interface Members | 7.8 Interfaces | Inherits Statement (Visual Basic Language Reference)

7.8.2 Interface Members

The members of an interface consist of the members introduced by its member declarations and the members inherited from its base interfaces.

Only nested types, methods, properties, and events may be members of an interface. Methods and properties may not have a body. Interface members are implicitly **Public** and may not specify an access modifier. Interface members have no scope, and they must always be qualified.

```
InterfaceMemberDeclaration ::=
   NonModuleDeclaration |
   EventMemberDeclaration |
   MethodMemberDeclaration |
   PropertyMemberDeclaration
```

See Also

7.8.1 Interface Inheritance | 7.8 Interfaces | Interfaces in Visual Basic .NET (Visual Basic Language Concepts)

7.9 Arrays

An *array* is a reference type that contains variables accessed through *indices* corresponding in a one-to-one fashion with the order of the variables in the array. The variables contained in an array, also called the *elements* of the array, must all be of the same type, and this type is called the *element type* of the array. The elements of an array come into existence when an array instance is created, and cease to exist when the array instance is destroyed. Each element of an array is initialized to the default value of its type. The type **System.Array** is the base type of all array types and may not be instantiated. Every array type inherits the members declared by the **System.Array** type and is convertible to it (and **Object**).

An array has a *rank* that determines the number of indices associated with each array element. The rank of an array determines the number of *dimensions* of the array. For example, an array with a rank of one is called a single-dimensional array, and an array with a rank greater than one is called a multidimensional array.

The following example creates a single-dimensional array of integer values, initializes the array elements, and then prints each of them out:

```
Imports System

Module Test
    Sub Main()
        Dim arr(5) As Integer
        Dim i As Integer

        For i = 0 To arr.Length - 1
            arr(i) = i * i
        Next i

        For i = 0 To arr.Length - 1
            Console.WriteLine("arr(" & i & ") = " & arr(i))
        Next i
    End Sub
End Module
```

The program outputs the following:

```
arr(0) = 0
arr(1) = 1
arr(2) = 4
arr(3) = 9
arr(4) = 16
arr(5) = 25
```

Each dimension of an array has an associated length. Dimension lengths are not part of the type of the array, but rather are established when an instance of the array type is created at run time. The length of a dimension determines the valid range of indices for that dimension: for a dimension of length `N`, indices can range from zero to `N - 1`. If a dimension is of length zero, there are no valid indices for that dimension. The total number of elements in an array is the product of the lengths of each dimension in the array. If any of the dimensions of an array has a length of zero, the array is said to be empty. The element type of an array can be any type.

Array types are specified by adding a modifier to an existing type name. The modifier consists of a left parenthesis, a set of zero or more commas, and a right parenthesis. The type modified is the element type of the array, and the number of dimensions is the number of commas plus one. If more than one modifier is specified, then the element type of the array is an array. The modifiers are read left to right, with the leftmost modifier being the outermost array. In the example

```
Module Test
    Dim arr As Integer(,)(,,)()
End Module
```

the element type of `arr` is a two-dimensional array of three-dimensional arrays of one-dimensional arrays of **Integer**.

A variable may also be declared to be of an array type by putting an array type modifier or an array initialization modifier on the variable name. In that case, the array element type is the type given in the declaration, and the array dimensions are determined by the variable name modifier. For clarity, it is not valid to have an array type modifier on both a variable name and a type name

in the same declaration.

The following example shows a variety of local variable declarations that use array types with **Integer** as the element type:

```
Module Test
    Sub Main()
        Dim a1() As Integer    ' Declares 1-dimensional array of integers.
        Dim a2(,) As Integer   ' Declares 2-dimensional array of integers.
        Dim a3(,,) As Integer  ' Declares 3-dimensional array of integers.

        Dim a4 As Integer()    ' Declares 1-dimensional array of integers.
        Dim a5 As Integer(,)   ' Declares 2-dimensional array of integers.
        Dim a6 As Integer(,,)  ' Declares 3-dimensional array of integers.

        ' Declare 1-dimensional array of 2-dimensional arrays of integers
        Dim a7()(,) As Integer
        ' Declare 2-dimensional array of 1-dimensional arrays of integers.
        Dim a8(,)() As Integer

        Dim a9() As Integer() ' Not allowed.
    End Sub
End Module
```

An array type name modifier extends to all sets of parentheses that follow it. This means that in the situations where a set of arguments enclosed in parenthesis is allowed after a type name, it is not possible to specify the arguments for an array type name. For example:

```
Module Test
    Sub Main()
        ' This calls the Integer constructor.
        Dim x As New Integer(3)

        ' This declares a variable of Integer().
        Dim y As Integer()

        ' This gives an error.
        ' Array sizes can not be specified in a type name.
        Dim z As Integer()(3)
    End Sub
End Module
```

In the last case, `(3)` is interpreted as part of the type name rather than as a set of constructor arguments.

```
ArrayTypeName ::= TypeName ArrayTypeModifiers
ArrayTypeModifiers ::= ArrayTypeModifier+
ArrayTypeModifier ::= ( [ RankList ] )
RankList ::=
   , |
   RankList ,
ArrayNameModifier ::=
   ArrayTypeModifiers |
   ArraySizeInitializationModifier
```

See Also

7.1 Value Types and Reference Types | 7.3 Primitive Types | 7.4 Enumerations | 7.6 Structures | 7.5 Classes | 7.7 Standard Modules | 7.8 Interfaces | 7.10 Delegates | Arrays Summary (Visual Basic Language Reference) | Arrays (Visual Basic Language Concepts)

7.10 Delegates

A *delegate* is a reference type that refers to a **Shared** method of a type or to an instance method of an object. The closest equivalent of a delegate in other languages is a function pointer, but whereas a function pointer can only reference **Shared** functions, a delegate can reference both **Shared** and instance methods. In the latter case, the delegate stores not only a reference to the method's entry point, but also a reference to the object instance with which to invoke the method.

The method declaration may not have attributes, modifiers, a **Handles** clause, an **Implements** clause, a method body, or an **End** construct. The parameter list of the method declaration may not contain **Optional** or **ParamArray** parameters. The accessibility domain of the return type and parameter types must be the same as or a superset of the accessibility domain of the delegate itself.

The members of a delegate are the members inherited from class **System.Delegate**. A delegate also contains a set of constructors and methods defined by the system. As these are specified and implemented by the .NET Framework, they are not listed here.

There are three steps in defining and using delegates: declaration, instantiation, and invocation.

Delegates are declared using delegate declaration syntax. The following example declares a delegate named `SimpleDelegate` that takes no arguments:

```
Delegate Sub SimpleDelegate()
```

The next example creates a `SimpleDelegate` instance and then immediately calls it:

```
Imports System

Module Test
    Sub F()
        System.Console.WriteLine("Test.F")
    End Sub

    Sub Main()
        Dim d As SimpleDelegate = AddressOf F
        d()
    End Sub
End Module
```

There is not much point in instantiating a delegate for a method and then immediately calling via the delegate, as it would be simpler to call the method directly. Delegates show their usefulness when their anonymity is used. The next example shows a `MultiCall` method that repeatedly calls a `SimpleDelegate` instance:

```
Sub MultiCall(ByVal d As SimpleDelegate, ByVal count As Integer)
    Dim i As Integer

    For i = 0 To count - 1
        d()
    Next i
End Sub
```

It is unimportant to the `MultiCall` method what the target method for the `SimpleDelegate` is, what accessibility this method has, or whether the method is **Shared** or nonshared. All that matters is that the signature of the target method is compatible with `SimpleDelegate`.

```
DelegateDeclaration ::=
   [ Attributes ] [ DelegateModifier+ ] Delegate MethodDeclaration
DelegateModifier ::= AccessModifier | Shadows
```

See Also

7.1 Value Types and Reference Types | 7.3 Primitive Types | 7.4 Enumerations | 7.6 Structures | 7.5 Classes | 7.7 Standard Modules

| 7.8 Interfaces | 7.9 Arrays | Delegate Statement (Visual Basic Language Reference) |
Events and Delegates (Visual Basic Language Concepts)

8. Conversions

Conversion is the process of changing a value from one type to another. Conversions may either be widening or narrowing. A *widening conversion* is a conversion from one type to another type that is guaranteed to be able to contain it. Widening conversions never fail. A *narrowing conversion* may fall into one of two categories. One category is a conversion from one type to another type that is not guaranteed to be able to contain it. The other category is a conversion between types that are sufficiently unrelated as to have no obvious conversion. Narrowing conversions, which entail loss of information, may fail.

See Also

8.1 Implicit and Explicit Conversions | 8.5 Array Covariance | 8.8 Widening Conversions | 8.9 Narrowing Conversions | Conversion Summary (Visual Basic Language Reference) | Type Conversions (Visual Basic Language Concepts)

8.1 Implicit and Explicit Conversions

Conversions can be either *implicit* or *explicit*. Implicit conversions occur without any special syntax. The following is an example of implicit conversion of an **Integer** value to a **Long** value:

```
Module Test
    Sub Main()
        Dim intValue As Integer = 123
        Dim longValue As Long = intValue

        Console.WriteLine(intValue & " = " & longValue)
    End Sub
End Module
```

Explicit conversions, on the other hand, require cast operators. Attempting to do an explicit conversion on a value without a cast operator causes a compile-time error. The following example uses an explicit conversion to convert a **Long** value to an **Integer** value.

```
Module Test
    Sub Main()
        Dim longValue As Long = 134
        Dim intValue As Integer = CInt(longValue)

        Console.WriteLine(longValue & " = " & intValue)
    End Sub
End Module
```

The set of implicit conversions depends on the compilation environment and the **Option Strict** statement. If strict semantics are being used, only widening conversions may occur implicitly. If permissive semantics are being used, all widening and narrowing conversions may occur implicitly.

See Also

8.5 Array Covariance | 8.8 Widening Conversions | 8.9 Narrowing Conversions | 8. Conversions |
Conversion Summary (Visual Basic Language Reference) | Implicit and Explicit Conversions (Visual Basic Language Concepts)

8.2 Boolean Conversions

Although **Boolean** is not a numeric type, it does have conversions to and from the numeric types as if it were an enumerated type. The literal **True** converts to the literal `255` for **Byte** and to the expression `-1` for **Short**, **Integer**, **Long**, **Decimal**, **Single**, and **Double**. The literal **False** converts to the literal `0`. A zero numeric value converts to the literal **False**. All other numeric values convert to the literal **True**.

See Also

8.5 Array Covariance | 8.8 Widening Conversions | 8.9 Narrowing Conversions | 8. Conversions |
Conversion Summary (Visual Basic Language Reference) | Implicit and Explicit Conversions (Visual Basic Language Concepts)

8.3 Numeric Conversions

Numeric conversions are conversions between the types **Byte**, **Short**, **Integer**, **Long**, **Decimal**, **Single** and **Double**, and enumerated types. Enumerated types are treated as if they were their underlying types for the purpose of conversions. When converting from a numeric type to an enumerated type, the numeric type is never required to conform to the set of values defined in the enumerated type.

Numeric conversions are processed at run-time as follows:

- For a conversion from a numeric type to a wider numeric type, the value is simply converted to the wider type. Conversions from **Integer**, **Long**, or **Decimal** to **Single** or **Double** are rounded to the nearest **Single** or **Double** value. While this conversion may cause a loss of precision, it will never cause a loss of magnitude.
- For a conversion from an integral type to a smaller integral type, or from **Single**, **Double**, or **Decimal** to an integral type, the result depends on whether integer overflow checking is on:

  If integer overflow is being checked:

  - If the source is an integral type, the conversion succeeds if the source argument is within the range of the destination type. The conversion throws a **System.OverflowException** exception if the source argument is outside the range of the destination type.
  - If the source is **Single**, **Double**, or **Decimal**, the source value is rounded up or down to the nearest integral value, and this integral value becomes the result of the conversion. If the source value is equally close to two integral values, the value is rounded to the value that has an even number in the least significant digit position. If the resulting integral value is outside the range of the destination type, a **System.OverflowException** exception is thrown.

  If integer overflow is not being checked:

  - If the source is an integral type, the conversion always succeeds and simply consists of discarding the most significant bits of the source value.
  - If the source is **Single**, **Double**, or **Decimal**, the conversion always succeeds and simply consists of rounding the source value towards the nearest integral value. If the source value is equally close to two integral values, the value is always rounded to the value that has an even number in the least significant digit position.
  - For a conversion from **Double** to **Single**, the **Double** value is rounded to the nearest **Single** value. If the **Double** value is too small to represent as a **Single**, the result becomes positive zero or negative zero. If the **Double** value is too large to represent as a **Single**, the result becomes positive infinity or negative infinity. If the **Double** value is NaN, the result is also NaN.
- For a conversion from **Single** or **Double** to **Decimal**, the source value is converted to **Decimal** representation and rounded to the nearest number after the 28th decimal place if required. If the source value is too small to represent as a **Decimal**, the result becomes zero. If the source value is NaN, infinity, or too large to represent as a **Decimal**, a **System.OverflowException** exception is thrown.
- For a conversion from **Double** to **Single**, the **Double** value is rounded to the nearest **Single** value. If the **Double** value is too small to represent as a **Single**, the result becomes positive zero or negative zero. If the **Double** value is too large to represent as a **Single**, the result becomes positive infinity or negative infinity. If the **Double** value is NaN, the result is also NaN.

See Also

8.5 Array Covariance | 8.8 Widening Conversions | 8.9 Narrowing Conversions | 8. Conversions |
Conversion Summary (Visual Basic Language Reference) | Implicit and Explicit Conversions (Visual Basic Language Concepts)

8.4 Reference Conversions

A value typed as a reference type may always be converted to a base type. Conversions from a base type to a derived type only succeed at run time if the value being converted is a null reference or a reference type that is either the derived type itself or a more derived type.

Within the .NET Framework, it is possible to know at compile time whether a class type implements a particular interface type. Consequently, a class type can always be cast to an interface type that it implements. Similarly, conversions from an interface type to a class type that implements it only succeed at run time if the value being converted is a null reference or a reference type that is either the class type itself or a type derived from the class type.

However, classes that represent COM classes may have interface implementations that are not known until run time. Consequently, a class type may also be converted to an interface type that it does not implement, an interface type may be converted to a class type that does not implement it, and an interface type may be converted to another interface type with which it has no inheritance relationship. In all these cases, a check is performed at run time by the .NET Framework to determine if the class type involved implements the required interface types.

If a reference conversion fails at run time, a **System.InvalidCastException** exception is thrown.

See Also

8.5 Array Covariance | 8.8 Widening Conversions | 8.9 Narrowing Conversions | 8. Conversions |
Conversion Summary (Visual Basic Language Reference) | Implicit and Explicit Conversions (Visual Basic Language Concepts)

8.5 Array Covariance

For any two reference types `A` and `B`, if `A` is a derived type of `B`, a conversion exists from an array of type `A` to a compatible array of type `B`. A *compatible array* is an array of the same rank and type. This relationship is known as *array covariance*. Array covariance in particular means that an element of an array whose element type is `B` may actually be an element of an array whose element type is `A`, provided that both `A` and `B` are reference types and that `B` is a base type of `A`. In the following example, the second invocation of `F` causes a **System.ArrayTypeMismatchException** exception to be thrown because the actual element type of `b` is **String**, not **Object**:

```
Module Test
    Sub F(ByRef x As Object)
    End Sub

    Sub Main()
        Dim a(10) As Object
        Dim b() As Object = New String(10) {}
        F(a(0)) ' OK.
        F(b(1)) ' Not allowed: System.ArrayTypeMismatchException.
   End Sub
End Module
```

Because of array covariance, assignments to elements of reference type arrays include a run-time check that ensures that the value being assigned to the array element is actually of a permitted type.

```
Module Test
    Sub Fill(array() As Object, index As Integer, count As Integer, _
            value As Object)
        Dim i As Integer

        For i = index To (index + count) - 1
            array(i) = value
        Next i
    End Sub

    Sub Main()
        Dim strings(100) As String

        Fill(strings, 0, 101, "Undefined")
        Fill(strings, 0, 10, Nothing)
        Fill(strings, 91, 10, 0)
    End Sub
End Module
```

In this example, the assignment to `array(i)` in method `Fill` implicitly includes a run-time check that ensures that the object referenced by the variable `value` is either **Nothing** or an instance of a type that is compatible with the actual element type of array `array`. In method `Main`, the first two invocations of method `Fill` succeed, but the third invocation causes a **System.ArrayTypeMismatchException** exception to be thrown upon executing the first assignment to `array(i)`. The exception occurs because an **Integer** cannot be stored in a **String** array.

Array covariance specifically does not extend to arrays of value types.

See Also

8.5 Array Covariance | 8.8 Widening Conversions | 8.9 Narrowing Conversions | 8. Conversions |
Conversion Summary (Visual Basic Language Reference) | Array Conversions (Visual Basic Language Concepts)

8.6 Value Type Conversions

Although interfaces are reference types, value types may be converted to and from them. When converting a value type to an interface, the value type is copied to the .NET Framework heap and a reference is returned. This reference is then converted to the interface type.

When converting an interface type to a value type, the value must be a null reference or an instance of the value type. Otherwise a **System.InvalidCastException** exception is thrown. The conversion copies the value on the .NET Framework heap to the stack and converts it to the value type. A null reference is treated as if it were the literal **Nothing**.

See Also

8.5 Array Covariance | 8.8 Widening Conversions | 8.9 Narrowing Conversions | 8. Conversions |
Conversion Summary (Visual Basic Language Reference) | Array Conversions (Visual Basic Language Concepts)

8.7 String Conversions

Converting **Char** into **String** results in a string whose first character is the character value. Converting **String** into **Char** results in a character whose value is the first character of the string. Converting an array of **Char** into **String** results in a string whose characters are the elements of the array. Converting **String** into an array of **Char** results in an array of characters whose elements are the characters of the string.

The exact conversions between **String** and **Boolean**, **Byte**, **Short**, **Integer**, **Long**, **Decimal**, **Single**, **Double**, **Date**, and vice versa, are beyond the scope of this specification and are implementation dependent with the exception of one detail. String conversions always consider the current culture of the run-time environment. As such, they must be performed at run time.

See Also

8.5 Array Covariance | 8.8 Widening Conversions | 8.9 Narrowing Conversions | 8. Conversions |
Conversion Summary (Visual Basic Language Reference) | Array Conversions (Visual Basic Language Concepts)

8.8 Widening Conversions

Widening conversions never overflow but may entail a loss of precision. The following conversions are widening conversions:

- Conversions from any derived type to one of its base types.
- Numeric type conversions in the following direction: **Byte**, **Short**, **Integer**, **Long**, **Decimal**, **Single**, **Double**. Conversions from **Integer** to **Single**, **Long** to **Single** or **Double**, or **Decimal** to **Single** or **Double** may cause a loss of precision, but will never cause a loss of magnitude. The other widening numeric conversions never lose any information.
- Conversions from the literal **Nothing** to any type.
- Conversions from the literal `0` to any enumerated type.
- Conversions from any enumerated type to its underlying type, or to any type that its underlying type has a widening conversion to.
- Conversions from any reference or value type to an interface type that the value or reference type implements.
- Conversions from an array type `S` with an element type `SE` to a covariant-array type `T` with an element type `TE`, provided all of the following are true:
  - `S` and `T` differ only in element type.
  - Both `SE` and `TE` are reference types.
  - A widening reference conversion exists from `SE` to `TE`.
- Conversions from **Char** to **String**.
- Conversions from **Char()** to **String**.
- Conversions from a constant expression of type **Long**, **Integer**, **Short**, or **Byte** to a narrower type, provided the value of the constant expression is within the range of the destination type.

See Also

8.5 Array Covariance | 8.9 Narrowing Conversions | 8. Conversions | Conversion Summary (Visual Basic Language Reference) | Widening and Narrowing Conversions (Visual Basic Language Concepts)

8.9 Narrowing Conversions

Narrowing conversions are conversions that cannot be proved to always succeed, conversions that are known to possibly lose information, and conversions across domains of types sufficiently different to merit narrowing notation. The following conversions are classified as narrowing conversions:

- Conversions from any type to a more derived type.
- Numeric type conversions in the following direction: **Double**, **Single**, **Decimal**, **Long**, **Integer**, **Short**, **Byte**.
- Conversions from **Boolean** to any numeric type.
- Conversions from any numeric type to **Boolean**.
- Conversions from any numeric type to any enumerated type.
- Conversions from any enumerated type to any type its underlying type has a narrowing conversion to.
- Conversions from any enumerated type to any other enumerated type.
- Conversions from any class type to any interface type, provided the class type does not implement the interface type.
- Conversions from any interface type to any class type.
- Conversions from any interface type to any value type that implements the interface type.
- Conversions from any interface type to any other interface type, provided there is no inheritance relationship between the two types.
- Conversions from an array type `S` with an element type `SE`, to a covariant-array type `T` with an element type `TE`, provided that all of the following are true:
    - `S` and `T` differ only in element type.
    - Both `SE` and `TE` are reference types.
    - A narrowing reference conversion exists from `SE` to `TE`.
- Conversions from **String** to **Char**.
- Conversions from **String** to **Char()**.
- Conversions from **String** to **Boolean** and from **Boolean** to **String**.
- Conversions between **String** and **Byte**, **Short**, **Integer**, **Long**, **Decimal**, **Single**, or **Double**.
- Conversions from **String** to **Date** and from **Date** to **String**.

See Also

8.3 Numeric Conversions | 8.4 Reference Conversions | 8.6 Value Type Conversions | 8.5 Array Covariance | 8.8 Widening Conversions | 8. Conversions | Conversion Summary (Visual Basic Language Reference) | Widening and Narrowing Conversions (Visual Basic Language Concepts)

9. Type Members

Type members define storage locations and executable code. They can be methods, constructors, events, constants, variables, and properties.

See Also

9.2 Methods | 9.3 Constructors | 9.4 Events | 9.5 Constants | 9.6 Instance and Shared Variables | 9.7 Properties | Objects, Properties, Methods, and Events (Visual Basic Language Concepts)

9.1 Interface Method Implementation

Methods, events, and properties can implement interface members. To implement an interface member, a member declaration specifies the **Implements** keyword and lists one or more interface members. Methods and properties that implement interface members are implicitly **NotOverridable** unless declared to be **MustOverride**, **Overridable**, or overriding another member. It is an error for a member implementing an interface member to be **Shared**. A member's accessibility has no effect on its ability to implement interface members.

For an interface implementation to be valid, the implements list of the containing type must name an interface that contains a compatible member. A compatible member is one whose signature matches the signature of the implementing member. If an event declared using a delegate type is implementing an interface event, then a compatible event is one whose underlying delegate type is the same type. Otherwise, the event uses the delegate type from the interface event it is implementing. If such an event implements multiple interface events, all the interface events must have the same underlying delegate type. For example:

```
Interface ClickEvents
    Event LeftClick(ByVal x As Integer, ByVal y As Integer)
    Event RightClick(ByVal x As Integer, ByVal y As Integer)
End Interface

Class Button
    Implements ClickEvents

    ' OK. Signatures match, delegate type = ClickEvents.LeftClickHandler.
    Event LeftClick(ByVal x As Integer, ByVal y As Integer) _
        Implements ClickEvents.LeftClick

    ' OK. Signatures match, delegate type = ClickEvents.RightClickHandler.
    Event RightClick(ByVal x As Integer, ByVal y As Integer) _
        Implements ClickEvents.RightClick
End Class

Class Label
    Implements ClickEvents

    ' Error. Signatures match, but can't be both delegate types.
    Event Click(ByVal x As Integer, ByVal y As Integer) _
        Implements ClickEvents.LeftClick, ClickEvents.RightClick
End Class
```

An interface member in the implements list is specified using a type name, a period, and an identifier. The type name must be an interface in the implements list or a base interface of an interface in the implements list, and the identifier must be a member of the specified interface. A single member can implement more than one matching interface member.

```
Interface ILeft
    Sub F()
End Interface

Interface IRight
    Sub F()
End Interface

Class Test
    Implements ILeft, IRight

    Sub F() Implements ILeft.F, IRight.F
    End Sub
End Class
```

If the interface member being implemented is unavailable in all explicitly implemented interfaces because of multiple interface inheritance, the implementing member must explicitly reference a base interface on which the member is available. For example, if `I1` and `I2` contain a member `M`, and `I3` inherits from `I1` and `I2`, a type implementing `I3` will implement `I1.M` and `I2.M`. If an interface shadows multiply inherited members, an implementing type will have to implement the inherited members and the member(s) shadowing them.

```
Interface ILeft
    Sub F()
End Interface

Interface IRight
    Sub F()
End Interface

Interface ILeftRight
    Inherits ILeft, IRight

    Shadows Sub F()
End Interface

Class Test
    Implements ILeftRight

    Sub LeftF() Implements ILeft.F
    End Sub

    Sub RightF() Implements IRight.F
    End Sub

    Sub LeftRightF() Implements ILeftRight.F
    End Sub
End Sub
```

```
ImplementsClause ::= [ Implements ImplementsList ]
ImplementsList ::=
   InterfaceMemberSpecifier |
   ImplementsList , InterfaceMemberSpecifier
InterfaceMemberSpecifier ::= TypeName . Identifier
```

See Also

9.2.1 Regular Method Declarations | 9.2.2 External Method Declarations | 9.2.3 Overridable Methods |9.2.4 Shared Methods | 9.2.5 Method Parameters | 9.2.6 Event Handling | 9.2 Methods | Implements Statement (Visual Basic Language Reference) | Implements Keyword (Visual Basic Language Concepts)

9.2 Methods

Methods contain the executable statements of a program. Methods, which have an optional list of parameters and an optional return value, are either shared or nonshared. Shared methods are accessed through the class or instances of the class. Nonshared methods, also called instance methods, are accessed through instances of the class. The following example shows a class `Stack` that has several shared methods (`Clone` and `Flip`), and several instance methods (`Push`, `Pop`, and `ToString`):

```
Imports System

Public Class Stack
    Public Shared Function Clone(ByVal s As Stack) As Stack
    End Function

    Public Shared Function Flip(ByVal s As Stack) As Stack
    End Function

    Public Function Pop() As Object
    End Function

    Public Sub Push(ByVal o As Object)
    End Sub

    Public Overrides Function ToString() As String
    End Function
End Class

Module Test
    Sub Main()
        Dim s As Stack = New Stack()
        Dim i As Integer

        While i < 10
            s.Push(i)
        End While

        Dim flipped As Stack = Stack.Flip(s)
        Dim cloned As Stack = Stack.Clone(s)

        Console.WriteLine("Original stack: " & s.ToString())
        Console.WriteLine("Flipped stack: " & flipped.ToString())
        Console.WriteLine("Cloned stack: " & cloned.ToString())
    End Sub
End Module
```

Methods can be overloaded, which means that multiple methods may have the same name so long as they have unique signatures. The signature of a method consists of the name of the method and the number and types of its parameters. The signature of a method specifically does not include the return type or parameter modifiers. The following example shows a class with a number of `F` methods:

```
Imports System

Module Test
    Sub F()
        Console.WriteLine("F()")
    End Sub

    Sub F(ByVal o As Object)
        Console.WriteLine("F(Object)")
    End Sub

    Sub F(ByVal value As Integer)
        Console.WriteLine("F(Integer)")
    End Sub

    Sub F(ByVal a As Integer, ByVal b As Integer)
```

```
        Console.WriteLine("F(Integer, Integer)")
    End Sub

    Sub F(ByVal values() As Integer)
        Console.WriteLine("F(Integer())")
    End Sub

    Sub Main()
        F()
        F(1)
        F(CType(1, Object))
        F(1, 2)
        F(New Integer() { 1, 2, 3 })
    End Sub
End Module
```

The output of the program is:

```
F()
F(Integer)
F(Object)
F(Integer, Integer)
F(Integer())
```

```
MethodMemberDeclaration ::= MethodDeclaration | ExternalMethodDeclaration
```

See Also

9.2.1 Regular Method Declarations | 9.2.2 External Method Declarations | 9.2.3 Overridable Methods | 9.2.4 Shared Methods | 9.2.5 Method Parameters | 9.1 Interface Method Implementation | 9.2.6 Event Handling | 9.3 Constructors | 9.4 Events | 9.5 Constants | 9.6 Instance and Shared Variables | 9.7 Properties | Methods (Visual Basic Language Reference) | Performing Actions with Methods (Visual Basic Language Concepts)

9.2.1 Regular Method Declarations

There are two types of methods: *subroutines*, which do not return values, and *functions*, which do. The body and **End** construct of a method may only be omitted if the method is defined in an interface or has the **MustOverride** modifier. If no return type is specified on a function and strict semantics are being used, a compile-time error occurs; otherwise the type is implicitly **Object** or the type of the method's type character. The accessibility domain of the return type and parameter types of a method must be the same as or a superset of the accessibility domain of the method itself.

```
MethodDeclaration ::= SubDeclaration | FunctionDeclaration
SubDeclaration ::=
   [ Attributes ] [ ProcedureModifier+ ] Sub Identifier
      [ ( [ ParameterList ] ) ] [ HandlesOrImplements ] LineTerminator
   [ Block ]
   [ End Sub LineTerminator ]
FunctionDeclaration ::=
   [ Attributes ] [ ProcedureModifier+ ] Function Identifier
      [ ( [ ParameterList ] ) ] [ As [ Attributes ] TypeName ]
      [ HandlesOrImplements ] LineTerminator
   [ Block ]
   [ End Function LineTerminator ]
ProcedureModifier ::=
   AccessModifier |
   Shadows |
   Shared |
   Overridable |
   NotOverridable |
   MustOverride |
   Overrides |
   Overloads
HandlesOrImplements ::= HandlesClause | ImplementsClause
```

See Also

9.2.2 External Method Declarations | 9.2.3 Overridable Methods |9.2.4 Shared Methods | 9.2.5 Method Parameters | 9.1 Interface Method Implementation | 9.2.6 Event Handling | 9.2 Methods | 4.6 Accessibility | 4.3.3 Shadowing | Sub Statement (Visual Basic Language Reference) | Sub Procedures (Visual Basic Language Concepts)

9.2.2 External Method Declarations

An external method declaration introduces a new method whose implementation is provided external to the program. Because an external method declaration provides no actual implementation, it has no method body or **End** construct. External methods are implicitly shared and may not handle events or implement interface members. If no return type is specified on a function and strict semantics are being used, a compile-time error occurs. Otherwise the type is implicitly **Object** or the type of the method's type character. The accessibility domain of the return type and parameter types of an external method must be the same as or a superset of the accessibility domain of the external method itself.

The library clause of an external method declaration specifies the name of the external file that implements the method. The optional alias clause is a string that specifies the numeric ordinal (prefixed by a `@` character) or name of the method in the external file. A single-character set modifier may also be specified, which governs the character set used to marshal strings during a call to the external method. The **Unicode** modifier marshals all strings to Unicode values, the **Ansi** modifier marshals all strings to ANSI values, and the **Auto** modifier marshals the strings according to .NET Framework rules based on the name of the method, or the alias name if specified. If no modifier is specified, the default is **Ansi**.

If **Ansi** or **Unicode** is specified, then the method name is looked up in the external file with no modification. If **Auto** is specified, then method name lookup depends on the platform. If the platform is considered to be ANSI (for example, Windows 95, Windows 98, Windows ME), then the method name is looked up with no modification. If the lookup fails, an `A` is appended and the lookup tried again. If the platform is considered to be Unicode (for example, Windows NT, Windows 2000, Windows XP), then a `W` is appended and the name is looked up. If the lookup fails, the lookup is tried again without the `W`. For example:

```
Module Test
    ' All platforms bind to "ExternSub".
    Declare Ansi Sub ExternSub Lib "ExternDLL" ()

    ' All platforms bind to "ExternSub".
    Declare Unicode Sub ExternSub Lib "ExternDLL" ()

    ' ANSI platforms: bind to "ExternSub" then "ExternSubA".
    ' Unicode platforms: bind to "ExternSubW" then "ExternSub".
    Declare Auto Sub ExternSub Lib "ExternDLL" ()
End Module
```

Data types being passed to external methods are marshaled according to the .NET Framework data marshalling conventions with one exception. String variables that are passed by value (that is, `ByVal x As String`) are marshaled to the OLE Automation BSTR type, and changes made to the BSTR in the external method are reflected back in the string argument. This is because the type **String** in external methods is mutable, and this special marshalling mimics that behavior. String parameters that are passed by reference (i.e. `ByRef x As String`) are marshaled as a pointer to the OLE Automation BSTR type. It is possible to override these special behaviors by specifying the **System.Runtime.InteropServices.MarshalAsAttribute** attribute on the parameter.

The example demonstrates use of external methods:

```
Class Path
    Declare Function CreateDirectory Lib "kernel32" ( _
        ByVal Name As String, ByVal sa As SecurityAttributes) As Boolean
    Declare Function RemoveDirectory Lib "kernel32" ( _
        ByVal Name As String) As Boolean
    Declare Function GetCurrentDirectory Lib "kernel32" ( _
        ByVal BufSize As Integer, ByVal Buf As String) As Integer
    Declare Function SetCurrentDirectory Lib "kernel32" ( _
        ByVal Name As String) As Boolean
End Class
```

```
ExternalMethodDeclaration ::=
   ExternalSubDeclaration |
   ExternalFunctionDeclaration
ExternalSubDeclaration ::=
   [ Attributes ] [ ExternalMethodModifier+ ] Declare
      [ CharsetModifier ] Sub Identifier LibraryClause
      [ AliasClause ] [ ( [ ParameterList ] ) ] LineTerminator
ExternalFunctionDeclaration ::=
   [ Attributes ] [ ExternalMethodModifier+ ] Declare
```

```
        [ CharsetModifier ] Function Identifier LibraryClause
        [ AliasClause ] [ ( [ ParameterList ] ) ]
        [ As [ Attributes ] TypeName ] LineTerminator
  ExternalMethodModifier ::= AccessModifier | Shadows | Overloads
  CharsetModifier ::= Ansi | Unicode | Auto
  LibraryClause ::= Lib StringLiteral
  AliasClause ::= Alias StringLiteral
```

See Also

9.2.1 Regular Method Declarations | 9.2.4 Shared Methods | 9.2.5 Method Parameters | 9.1 Interface Method Implementation | 9.2.6 Event Handling | 9.2 Methods | 4.3.3 Shadowing | Declare Statement (Visual Basic Language Reference) | Adding Methods to a Class (Visual Basic Language Concepts)

9.2.3 Overridable Methods

The **Overridable** modifier indicates that a method is overridable. The **Overrides** modifier indicates that a method overrides a base-type overridable method that has the same signature. The **NotOverridable** modifier indicates that an overridable method cannot be further overridden. The **MustOverride** modifier indicates that a method must be overridden in derived classes.

Certain combinations of these modifiers are not valid:

- **Overridable** and **NotOverridable** are mutually exclusive and cannot be combined.
- **MustOverride** implies **Overridable** (and so cannot specify it) and cannot be combined with **NotOverridable**. **MustOverride** methods cannot override other methods, and so cannot be combined with **Overrides**.
- **NotOverridable** cannot be combined with **Overridable** or **MustOverride** and must be combined with **Overrides**.
- **Overrides** implies **Overridable** (and so cannot specify it) and cannot be combined with **MustOverride**.

There are also additional restrictions on overridable methods:

- A **MustOverride** method may not include a method body or an **End** construct, may not override another method, and may only appear in **MustInherit** classes.
- If a method specifies **Overrides** and there is no matching base method to override, a compile-time error occurs. An overriding method may not specify **Shadows**.
- A method may not override another method if the overriding method's accessibility domain is not equal to the accessibility domain of the method being overridden. The one exception is that a method overriding a **Protected Friend** method in another assembly must specify **Protected** (not **Protected Friend**).
- **Private** methods may not be **Overridable**, **NotOverridable**, or **MustOverride**, nor may they override other methods.
- Methods in **NotInheritable** classes may not be declared **Overridable** or **MustOverride**.

The following example illustrates the differences between overridable and nonoverridable methods:

```
Imports System

Class Base
    Public Sub F()
        Console.WriteLine("Base.F")
    End Sub

    Public Overridable Sub G()
        Console.WriteLine("Base.G")
    End Sub
End Class

Class Derived
    Inherits Base

    Public Shadows Sub F()
        Console.WriteLine("Derived.F")
    End Sub

    Public Overrides Sub G()
        Console.WriteLine("Derived.G")
    End Sub
End Class

Module Test
    Sub Main()
        Dim d As Derived = New Derived()
        Dim b As Base = d

        b.F()
        d.F()
        b.G()
        d.G()
    End Sub
End Module
```

In the example, class `Base` introduces a method `F` and an **Overridable** method `G`. The class `Derived` introduces a new method `F`, thus shadowing the inherited `F`, and also overrides the inherited method `G`. The example produces the following output:

```
Base.F
Derived.F
Derived.G
Derived.G
```

Notice that the statement `b.G()` invokes `Derived.G`, not `Base.G`. This is because the run-time type of the instance (which is `Derived`) rather than the compile-time type of the instance (which is `Base`) determines the actual method implementation to invoke.

See Also

9.2.1 Regular Method Declarations | 9.2.2 External Method Declarations | 9.2.3 Overridable Methods | 9.2.4 Shared Methods | 9.2.5 Method Parameters | 9.1 Interface Method Implementation | 9.2.6 Event Handling | 9.2 Methods | 4.3.3 Shadowing | Overridable (Visual Basic Language Reference) | Overrides (Visual Basic Language Reference) | Overriding Properties and Methods (Visual Basic Language Concepts)

9.2.4 Shared Methods

The **Shared** modifier indicates a method is a *shared method*. A shared method does not operate on a specific instance of a type and may be invoked directly from a type rather than through a particular instance of a type. It is valid, however, to use an instance to qualify a shared method. It is invalid to refer to **Me**, **MyClass**, or **MyBase** in a shared method. Shared methods may not be **Overridable**, **NotOverridable**, or **MustOverride**, and they may not override methods. Methods defined in standard modules and interfaces may not specify **Shared**, because they are implicitly **Shared** already.

A method declared in a structure or class without a **Shared** modifier is an *instance method*. An instance method operates on a given instance of a type. Instance methods can only be invoked through an instance of a type and may refer to the instance through the **Me** expression.

The following example illustrates the rules for accessing shared and instance members:

```
Class Test
    Private x As Integer
    Private Shared y As Integer

    Sub F()
        x = 1 ' Ok, same as Me.x = 1.
        y = 1 ' Ok, same as Test.y = 1.
    End Sub

    Shared Sub G()
        x = 1 ' Error, cannot access Me.x.
        y = 1 ' Ok, same as Test.y = 1.
    End Sub

    Shared Sub Main()
        Dim t As Test = New Test()

        t.x = 1 ' Ok.
        t.y = 1 ' Ok.
        Test.x = 1 ' Error, cannot access instance member through type.
        Test.y = 1 ' Ok.
    End Sub
End Class
```

Method `F` shows that in an instance function member, an identifier can be used to access both instance members and shared members. Method `G` shows that in a shared function member, it is an error to access an instance member through an identifier. Method `Main` shows that in a member access expression, instance members must be accessed through instances, but shared members can be accessed through types or instances.

See Also

9.2.1 Regular Method Declarations | 9.2.2 External Method Declarations | 9.2.3 Overridable Methods |9.2.5 Method Parameters | 9.1 Interface Method Implementation | 9.2.6 Event Handling | 9.2 Methods | 11.4.3 Instance Expressions | Shared (Visual Basic Language Reference) | Sub Procedures (Visual Basic Language Concepts)

9.2.5 Method Parameters

A *parameter* is a variable that can be used to pass information into and out of a method. Parameters of a method are declared by the method's parameter list, which consists of one or more parameters separated by commas. If no type is specified for a parameter and strict semantics are used, a compile-time error occurs. Otherwise the default type is **Object** or the type of the parameter's type character. Even under permissive semantics, if one parameter includes an **As** clause, all parameters must specify types.

Parameters are specified as value, reference, optional, or paramarray parameters by the modifiers **ByVal**, **ByRef**, **Optional**, and **ParamArray**, respectively. A parameter that does not specify **ByRef** or **ByVal** defaults to **ByVal**.

Parameter names are scoped to the entire body of the method and are always publicly accessible. A method invocation creates a copy, specific to that invocation, of the parameters, and the argument list of the invocation assigns values or variable references to the newly created parameters.

```
ParameterList ::=
   Parameter |
   ParameterList , Parameter
Parameter ::=
   [ Attributes ] ParameterModifier+ Identifier [ As TypeName ]
      [ = ConstantExpression ]
ParameterModifier ::= ByVal | ByRef | Optional | ParamArray
```

See Also

9.2.5.1 Value Parameters | 9.2.5.2 Reference Parameters | 9.2.5.3 Optional Parameters | 9.2.5.4 ParamArray Parameters |
9.2.1 Regular Method Declarations | 9.2.2 External Method Declarations | 9.2.3 Overridable Methods | 9.2.4 Shared Methods |
9.1 Interface Method Implementation | 9.2.6 Event Handling | 9.2 Methods |
Argument Passing ByVal and ByRef (Visual Basic Language Concepts)

9.2.5.1 Value Parameters

A *value parameter* is declared with an explicit **ByVal** modifier. If the **ByVal** modifier is used, the **ByRef** modifier may not be specified. A value parameter comes into existence with the invocation of the member the parameter belongs to, and is initialized with the value of the argument given in the invocation. A value parameter ceases to exist upon return of the member.

A method is permitted to assign new values to a value parameter. Such assignments only affect the local storage location represented by the value parameter; they have no effect on the actual argument given in the method invocation.

A value parameter is used when the value of an argument is passed into a method, and modifications of the parameter do not impact the original argument. A value parameter refers to its own variable, one that is distinct from the variable of the corresponding argument. This variable is initialized by copying the value of the corresponding argument. The following example shows a method `F` that has a value parameter named `p`:

```
Imports System

Module Test
    Sub F(ByVal p As Integer)
        Console.WriteLine("p = " & p)
        p += 1
    End Sub

    Sub Main()
        Dim a As Integer = 1

        Console.WriteLine("pre: a = " & a)
        F(a)
        Console.WriteLine("post: a = " & a)
    End Sub
End Module
```

The example produces the following output, even though the value parameter `p` is modified:

```
pre: a = 1
p = 1
post: a = 1
```

See Also

9.2.5.2 Reference Parameters | 9.2.5.3 Optional Parameters | 9.2.5.4 ParamArray Parameters | 9.2.2 External Method Declarations | 9.2.5 Method Parameters | ByVal (Visual Basic Language Reference) | Argument Passing ByVal (Visual Basic Language Concepts)

9.2.5.2 Reference Parameters

A reference parameter is a parameter declared with a **ByRef** modifier. If the **ByRef** modifier is specified, the **ByVal** modifier may not be used. A reference parameter does not create a new storage location. Instead, a reference parameter represents the variable given as the argument in the method or constructor invocation. Conceptually, the value of a reference parameter is always the same as the underlying variable.

A reference parameter is used when the parameter acts as an alias for a caller-provided argument. A reference parameter does not itself define a variable, but rather refers to the variable of the corresponding argument. Modifications of a reference parameter directly and immediately impact the corresponding argument. The following example shows a method `Swap` that has two reference parameters:

```
Imports System

Module Test
    Sub Swap(ByRef a As Integer, ByRef b As Integer)
        Dim t As Integer = a
        a = b
        b = t
    End Sub

    Sub Main()
        Dim x As Integer = 1
        Dim y As Integer = 2

        Console.WriteLine("pre: x = " & x & ", y = " & y)
        Swap(x, y)
        Console.WriteLine("post: x = " & x & ", y = " & y)
    End Sub
End Module
```

The output of the program is:

```
pre: x = 1, y = 2
post: x = 2, y = 1
```

For the invocation of method `Swap` in class `Main`, `a` represents `x`, and `b` represents `y`. Thus, the invocation has the effect of swapping the values of `x` and `y`.

In a method that takes reference parameters, it is possible for multiple names to represent the same storage location:

```
Module Test
    Private s As String

    Sub F(ByRef a As String, ByRef b As String)
        s = "One"
        a = "Two"
        b = "Three"
    End Sub

    Sub G()
        F(s, s)
    End Sub
End Module
```

In the example the invocation of method `F` in `G` passes a reference to `s` for both `a` and `b`. Thus, for that invocation, the names `s`, `a`, and `b` all refer to the same storage location, and the three assignments all modify the instance variable `s`.

If the type of the variable being passed to a reference parameter is not compatible with the reference parameter's type, or if a non-variable is passed as an argument to a reference parameter, a temporary variable may be allocated and passed to the reference parameter. The value being passed in will be copied into this temporary variable before the method is invoked and will be copied back to the original variable (if there is one) when the method returns. Thus, a reference parameter may not necessarily contain a reference to the exact storage of the variable being passed in, and any changes to the reference parameter may not be

reflected in the variable until the method exits. For example:

```
Class Base
End Class

Class Derived
    Inherits Base
End Class

Module Test
    Sub F(ByRef b As Base)
        b = New Base()
    End Sub

    Property G() As Base
        Get
        End Get
        Set
        End Set
    End Property

    Sub Main()
        Dim d As Derived

        F(G)   ' OK.
        F(d)   ' Throws TypeMismatchException after F returns.
    End Sub
End Module
```

In the case of the first invocation of `F`, a temporary variable, is created and the value of the property `G` is assigned to it and passed into `F`. Upon return from `F`, the value in the temporary variable is assigned back to the property of `G`. In the second case, another temporary variable is created and the value of `d` is assigned to it and passed into `F`. When returning from `F`, the value in the temporary variable is cast back to the type of the variable, `Derived`, and assigned to `d`. Since the value being passed back cannot be cast to `Derived`, an exception is thrown at run time.

See Also

9.2.5.1 Value Parameters | 9.2.5.3 Optional Parameters | 9.2.5.4 ParamArray Parameters | 9.2.2 External Method Declarations | 9.2.5 Method Parameters | ByRef (Visual Basic Language Reference) | Argument Passing ByRef (Visual Basic Language Concepts)

9.2.5.3 Optional Parameters

An optional parameter is declared with the **Optional** modifier. Parameters that follow an optional parameter in the formal parameter list must be optional as well; failure to specify the **Optional** modifier on the following parameters will trigger a compile-time error. An optional parameter must specify a constant expression to be used as a replacement value if no argument is specified. The expression must be implicitly convertible to the type of the parameter, and the result of the conversion must be a constant expression. Consequently, parameters typed as structures cannot be optional parameters. Optional parameters are the only situation in which an initializer on a parameter is valid. The initialization is always done as a part of the invocation expression, not within the method body itself.

```
Imports System

Module Test
    Sub F(ByVal x As Integer, Optional y As Integer = 20)
        Console.WriteLine("x = " & x & ", y = " & y)
    End Sub

    Sub Main()
        F(10)
        F(30,40)
    End Sub
End Module
```

The output of the program is:

```
x = 10, y = 20
x = 30, y = 40
```

Optional parameters may not be specified in delegate or event declarations.

See Also

9.2.5.1 Value Parameters | 9.2.5.2 Reference Parameters | 9.2.5.4 ParamArray Parameters | 9.2.2 External Method Declarations | 9.2.5 Method Parameters | Optional (Visual Basic Language Reference) | Optional Arguments (Visual Basic Language Concepts)

9.2.5.4 ParamArray Parameters

ParamArray parameters are declared with the **ParamArray** modifier. If the **ParamArray** modifier is present, the **ByVal** modifier must be specified, and no other parameter may use the **ParamArray** modifier. The **ParamArray** parameter's type must be a one-dimensional array, and it must be the last parameter in the parameter list.

A **ParamArray** parameter represents an indeterminate number of parameters of the type of the **ParamArray**. Within the method itself, a **ParamArray** parameter is treated as its declared type and has no special semantics. A **ParamArray** parameter is implicitly optional, with a default value of an empty one-dimensional array of the type of the **ParamArray**.

A **ParamArray** permits arguments to be specified in one of two ways in a method invocation:

- The argument given for a **ParamArray** can be a single expression of a type that widens to the **ParamArray** type. In this case, the **ParamArray** acts precisely like a value parameter.
- Alternatively, the invocation can specify zero or more arguments for the **ParamArray**, where each argument is an expression of a type that is implicitly convertible to the element type of the **ParamArray**. In this case, the invocation creates an instance of the **ParamArray** type with a length corresponding to the number of arguments, initializes the elements of the array instance with the given argument values, and uses the newly created array instance as the actual argument.

Except for allowing a variable number of arguments in an invocation, a **ParamArray** is precisely equivalent to a value parameter of the same type, as the following example illustrates.

```
Imports System

Module Test
    Sub F(ByVal ParamArray args As Integer)
        Dim i As Integer

        Console.Write("Array contains " & args.Length & " elements:")
        For Each i In args
            Console.Write(" " & i)
        Next i
        Console.WriteLine()
    End Sub

    Sub Main()
        Dim a As Integer() = { 1, 2, 3 }

        F(a)
        F(10, 20, 30, 40)
        F()
    End Sub
End Module
```

The example produces the output

```
Array contains 3 elements: 1 2 3
Array contains 4 elements: 10 20 30 40
Array contains 0 elements:
```

The first invocation of `F` simply passes the array `a` as a value parameter. The second invocation of `F` automatically creates a four-element array with the given element values and passes that array instance as a value parameter. Likewise, the third invocation of `F` creates a zero-element array and passes that instance as a value parameter. The second and third invocations are precisely equivalent to writing:

```
F(New Integer() {10, 20, 30, 40})
F(New Integer() {})
```

ParamArray parameters may not be specified in delegate or event declarations.

See Also

9.2.5.1 Value Parameters | 9.2.5.2 Reference Parameters | 9.2.5.3 Optional Parameters | 9.2.2 External Method Declarations | 9.2.5 Method Parameters | ParamArray (Visual Basic Language Reference) | Parameter Arrays (Visual Basic Language Concepts)

9.2.6 Event Handling

Methods can declaratively handle events raised by objects in instance or shared variables. To handle events, a method declaration specifies the **Handles** keyword and lists one or more events. An event in the **Handles** list is specified by two identifiers separated by a period:

- The first identifier must be an instance or shared variable in the containing type that specifies the **WithEvents** modifier or the **MyBase** keyword; otherwise, a compile-time error occurs. This variable contains the object that will raise the events handled by this method.
- The second identifier must specify a member of the type of the first identifier. The member must be an event, and may be shared. If a shared variable is specified for the first identifier, then the event must be shared, or an error results.

For a handler to be valid, the handled event's parameter types must exactly match those of the event handler. A single member can handle multiple matching events, and multiple methods may handle a single event. A method's accessibility has no effect on its ability to handle events. The following example shows how a method can handle events:

```
Imports System

Class Raiser
    Public Event Constructed()

    Public Sub New()
        RaiseEvent Constructed
    End Sub
End Class

Module Test
    Private WithEvents x As Raiser

    Public Sub Constructed() Handles x.Constructed
        Console.WriteLine("Constructed")
    Public End Sub

    Public Sub Main()
        x = New Raiser()
        x = New Raiser()
    End Sub
End Module
```

This will print out:

```
Constructed
Constructed
```

A type inherits all event handlers provided by its base type. A derived type cannot in any way alter the event mappings it inherits from its base types, but may add additional handlers to the event.

```
HandlesClause ::= [ Handles EventHandlesList ]
EventHandlesList ::=
   EventMemberSpecifier |
   EventHandlesList , EventMemberSpecifier
EventMemberSpecifier ::=
   Identifier . Identifier
   MyBase . Identifier
```

See Also

9.2.1 Regular Method Declarations | 9.2.2 External Method Declarations | 9.2.3 Overridable Methods |9.2.4 Shared Methods | 9.2.5 Method Parameters | 9.1 Interface Method Implementation | 9.2 Methods | 2.2 Identifiers | 11.4.3 Instance Expressions | Handles (Visual Basic Language Reference) | WithEvents and the Handles Clause (Visual Basic Language Concepts)

9.3 Constructors

Constructors are special methods that allow control over initialization. They are run after the program begins or when an instance of a type is created. Unlike other members, constructors are not inherited and do not introduce a name into a type's declaration space. Constructors may only be invoked by object-creation expressions or by the .NET Framework; they may never be directly invoked.

```
ConstructorMemberDeclaration ::=
   [ Attributes ] [ ConstructorModifier+ ] Sub New
      [ ( [ ParameterList ] ) ] LineTerminator
   [ Block ]
   End Sub LineTerminator
ConstructorModifier ::= AccessModifier | Shared
```

See Also

9.3.1 Instance Constructors | 9.3.2 Shared Constructors | 9.2 Methods | 9.4 Events | 9.5 Constants |
9.6 Instance and Shared Variables | 9.7 Properties | Using Constructors and Destructors (Visual Basic Language Concepts)

9.3.1 Instance Constructors

Instance constructors initialize instances of a type and are run by the .NET Framework when an instance is created. The parameter list of a constructor is subject to the same rules as the parameter list of a method. Instance constructors may be overloaded.

All constructors in reference types must invoke another constructor. If the invocation is explicit, it must be the first statement in the constructor method body. The statement can either invoke another of the type's instance constructors — for example, `Me.New(...)` or `MyClass.New(...)` — or if it is not a structure it can invoke an instance constructor of the type's base type — for example, `MyBase.New(...)`. It is invalid for a constructor to invoke itself. If a constructor omits a call to another constructor, `MyBase.New()` is implicit. If there is no parameterless base type constructor, a compile-time error occurs. Because **Me** is not considered to be constructed until after the call to a base class constructor, the parameters to a constructor invocation statement cannot reference **Me**, **MyClass**, or **MyBase** implicitly or explicitly.

When a constructor's first statement is of the form `MyBase.New(...)`, the constructor implicitly performs the initializations specified by the variable initializers of the instance variables declared in the type. This corresponds to a sequence of assignments that are executed immediately after invoking the direct base type constructor. Such ordering ensures that all base instance variables are initialized by their variable initializers before any statements that have access to the instance are executed. For example:

```
Imports System

Class A
    Protected x As Integer = 1
End Class

Class B
    Inherits A

    Private y As Integer = x

    Public Sub New()
        Console.WriteLine("x = " & x & ", y = " & y)
    End Sub
End Class
```

When `New B()` is used to create an instance of `B`, the following output is produced:

```
x = 1, y = 1
```

The value of `y` is `1` because the variable initializer is executed after the base class constructor is invoked. Variable initializers are executed in the textual order they appear in the type declaration.

When a type declares only **Private** constructors, it is not possible in general for other types to derive from the type or create instances of the type; the only exception is types nested within the type. **Private** constructors are commonly used in types that contain only **Shared** members.

If a type contains no instance constructor declarations, a default constructor is automatically provided. The default constructor simply invokes the parameterless constructor of the direct base type. If the direct base type does not have an accessible parameterless constructor, a compile-time error occurs. The declared access type for the default constructor is always **Public**.

In the following example a default constructor is provided because the class contains no constructor declarations:

```
Class Message
    Private sender As Object
    Private text As String
End Class
```

Thus, the example is precisely equivalent to the following:

```
Class Message
    Private sender As Object
    Private text As String
```

```
    Public Sub New()
    End Sub
End Class
```

See Also

9.3.2 Shared Constructors | 9.3 Constructors | Using Constructors and Destructors (Visual Basic Language Concepts)

9.3.2 Shared Constructors

Shared constructors initialize a type's shared variables; they are run after the program begins executing, but before any references to a member of the type. A shared constructor specifies the **Shared** modifier, unless it is in a standard module in which case the **Shared** modifier is implied.

Unlike instance constructors, shared constructors have implicit public access, have no parameters, and may not call other constructors. Before the first statement in a shared constructor, the shared constructor implicitly performs the initializations specified by the variable initializers of the shared variables declared in the type. This corresponds to a sequence of assignments that are executed immediately upon entry to the constructor. The variable initializers are executed in the textual order they appear in the type declaration.

The following example shows an `Employee` class with a shared constructor that initializes a shared variable:

```
Imports System.Data

Class Employee
    Private Shared ds As DataSet

    Shared Sub New()
        ds = New DataSet()
    End Sub

    Public Name As String
    Public Salary As Decimal
End Class
```

Exactly when shared constructors are run is mostly implementation dependent, though several guarantees are provided if a shared constructor is explicitly defined:

- Shared constructors are run before any instance of a class type is created.
- Shared constructors are run before any instance members of a structure type are accessed, or before any constructor of a structure type is explicitly called. Calling the implicit parameterless constructor created for structures will not cause the shared constructor to run.
- Shared constructors are run before any of the type's shared members are referenced.
- Shared constructors are run before any types that derive from the type are loaded.
- A shared constructor will not be run more than once during a single execution of a program.

The above guarantees do not apply in the situation where a shared constructor is implicitly created for shared initializers. The output from the following example is uncertain, because the exact ordering of loading and therefore of shared constructor execution is not defined:

```
Imports System

Module Test
    Sub Main()
        A.F()
        B.F()
    End Sub
End Module

Class A
    Shared Sub New()
        Console.WriteLine("Init A")
    End Sub

    Public Shared Sub F()
        Console.WriteLine("A.F")
    End Sub
End Class

Class B
    Shared Sub New()
```

```
        Console.WriteLine("Init B")
    End Sub

    Public Shared Sub F()
        Console.WriteLine("B.F")
    End Sub
End Class
```

The output could be either of the following:

```
Init A
A.F
Init B
B.F
```

or

```
Init B
Init A
A.F
B.F
```

By contrast, the following example produces predictable output, because the **Shared** constructor for the class `A` must execute before the **Shared** constructor for the class `B`, which derives from it:

```
Imports System

Module Test
    Sub Main()
        Console.WriteLine("1")
        B.G()
        Console.WriteLine("2")
    End Sub
End Module

Class A
    Shared Sub New()
        Console.WriteLine("Init A")
    End Sub
End Class

Class B
    Inherits A

    Shared Sub New()
        Console.WriteLine("Init B")
    End Sub

    Public Shared Sub G()
        Console.WriteLine("B.G")
    End Sub
End Class
```

The output is:

```
Init A
Init B
B.G
```

It is also possible to construct circular dependencies that allow **Shared** variables with variable initializers to be observed in their default value state, as in the following example:

```
Class A
```

```
    Public Shared X As Integer = B.Y + 1
End Class

Class B
    Public Shared Y As Integer = A.X + 1

    Shared Sub Main()
        Console.WriteLine("X = " & A.X & ", Y = " & B.Y)
    End Sub
End Class
```

This produces the output:

```
X = 1, Y = 2
```

To execute the `Main` method, the system first loads class `B`. The **Shared** constructor of class `B` proceeds to compute the initial value of `Y`, which recursively causes class `A` to be loaded because the value of `A.X` is referenced. The **Shared** constructor of class `A` in turn proceeds to compute the initial value of `X`, and in doing so fetches the *default* value of `Y`, which is zero. `A.X` is thus initialized to `1`. The process of loading `A` then completes, returning to the calculation of the initial value of `Y`, the result of which becomes `2`.

Had the `Main` method instead been located in class `A`, the example would have produced the following output:

```
X = 2, Y = 1
```

Avoid circular references in **Shared** variable initializers since it is generally impossible to determine the order in which classes containing such references are loaded.

See Also

9.3.1 Instance Constructors | 9.3 Constructors | Using Constructors and Destructors (Visual Basic Language Concepts)

9.4 Events

Events are used to notify code of a particular occurrence. An event declaration consists of an identifier, either a delegate type or a parameter list, and an optional **Implements** clause. If a delegate type is specified, the delegate type may not have a return type. If a parameter list is specified, it may not contain **Optional** or **ParamArray** parameters. The accessibility domain of the parameter types and/or delegate type must be the same as, or a superset of, the accessibility domain of the event itself. Events may be shared by specifying the **Shared** modifier.

In addition to the member name added to the type's declaration space, an event declaration implicitly declares several other members. Given an event named `X`, the following members are added to the declaration space:

- If the form of the declaration is a method declaration, a nested delegate class named `XEventHandler` is introduced. The nested delegate class matches the method declaration and has the same accessibility as the event. The attributes in the parameter list apply to the parameters of the delegate class.
- A **Private** instance variable typed as the delegate, named `XEvent`.
- A method named `add_X`, which takes the delegate type and has the same access type as the event.
- A method named `remove_X`, which takes the delegate type and has the same access type as the event.

If a type attempts to declare a name that matches one of the above names, a compile-time error will result. It is not possible to override or overload any of the introduced members, although it is possible to shadow them in derived types. For example, the class declaration

```
Class Raiser
    Public Event Constructed(ByVal i As Integer)
End Class
```

is equivalent to the following declaration

```
Class Raiser
    Public Delegate Sub ConstructedEventHandler(ByVal i As Integer)

    Protected ConstructedEvent As ConstructedEventHandler

    Public Sub add_Constructed(ByVal d As ConstructedEventHandler)
        ConstructedEvent = _
            CType( _
                ConstructedEvent.Combine(ConstructedEvent, d), _
                    Raiser.ConstructedEventHandler)
    End Sub

    Public Sub remove_Constructed(ByVal d As ConstructedEventHandler)
        ConstructedEvent = _
            CType( _
                ConstructedEvent.Remove(ConstructedEvent, d), _
                    Raiser.ConstructedEventHandler)
    End Sub
End Class
```

Declaring an event without specifying a delegate type is the simplest and most compact syntax, but has the disadvantage of declaring a new delegate type for each event. For example, in the following example, three hidden delegate types are created, even though all three events have the same parameter list:

```
Public Class Button
    Public Event Click(sender As Object, e As System.EventArgs)
    Public Event DoubleClick(sender As Object, e As System.EventArgs)
    Public Event RightClick(sender As Object, e As System.EventArgs)
End Class
```

In the following example, the events simply use the same delegate, `EventHandler`:

```
Delegate Sub EventHandler(sender As Object, e As System.EventArgs)
```

```
Public Class Button
    Public Event Click As EventHandler
    Public Event DoubleClick As EventHandler
    Public Event RightClick As EventHandler
End Class
```

Events can be handled in one of two ways: statically or dynamically. Statically handling events is simpler and only requires a **WithEvents** variable and a **Handles** clause. In the following example, class `Form1` statically handles the event `Click` of object `Button`:

```
Imports System

Public Class Form1
    Public WithEvents Button1 As Button = New Button()
    Public Sub Button1_Click(sender As Object, e As System.EventArgs) _
           Handles Button1.Click
        Console.WriteLine("Button1 was clicked!")
    End Sub
End Class
```

Dynamically handling events is more complex because the event must be explicitly connected and disconnected to in code. The statement `AddHandler` adds a handler for an event, and the statement `RemoveHandler` removes a handler for an event. The next example shows a class `Form1` that adds `Button1_Click` as an event handler for `Button1`'s `Click` event:

```
Imports System

Public Class Form1
    Public Sub New()
        ' Add Button1_Click as an event handler for Button1's Click event.
        AddHandler Button1.Click , AddressOf Button1_Click
    End Sub

    Private Button1 As Button = New Button()

    Sub Button1_Click(sender As Object, e As EventArgs)
        Console.WriteLine("Button1 was clicked!")
    End Sub

    Public Sub Disconnect()
        RemoveHandler Button1.Click, AddressOf Button1_Click
    End Sub
End Class
```

In method `Disconnect`, the event handler is removed.

```
EventMemberDeclaration ::=
   [ Attributes ] [ EventModifiers+ ] Event Identifier ParametersOrType
      [ ImplementsClause ] LineTerminator
ParametersOrType ::=
   [ ( [ ParameterList ] ) ] |
   As TypeName
EventModifiers ::= AccessModifier | Shadows | Shared
```

See Also

9.1 Interface Method Implementation | 9.2 Methods | 9.3 Constructors | 9.5 Constants | 9.6 Instance and Shared Variables | 9.7 Properties | 9.2.5 Method Parameters | 9.2.6 Event Handling | 4.6 Accessibility | 4.3.3 Shadowing | 9.2.4 Shared Methods | Event Statement (Visual Basic Language Reference) | Events and Event Handlers (Visual Basic Language Concepts)

9.5 Constants

A *constant* is a constant value that is a member of a type. Constants are implicitly shared. If the declaration contains an **As** clause, the clause specifies the type of the member introduced by the declaration. If the type is omitted and strict semantics are being used, a compile-time error occurs; otherwise the type of the constant is implicitly **Object**. The type of a constant may only be a primitive type or **Object**. If a constant is typed as **Object** and there is no type character, the real type of the constant will be the type of the constant expression. Otherwise, the type of the constant is the type of the constant's type character.

The following example shows a class named `Constants` that has two public constants:

```
Class Constants
    Public A As Integer = 1
    Public B As Integer = A + 1
End Class
```

Constants can be accessed through the class, as in the following example, which prints out the values of `Constants.A` and `Constants.B`.

```
Module Test
    Sub Main()
        Console.WriteLine(Constants.A & ", " & Constants.B)
    End Sub
End Module
```

A constant declaration that declares multiple constants is equivalent to multiple declarations of single constants. The following example declares three constants in one declaration statement.

```
Class A
    Protected Const x As Integer = 1, y As Long = 2, z As Short = 3
End Class
```

This declaration is equivalent to the following:

```
Class A
    Protected Const x As Integer = 1
    Protected Const y As Long = 2
    Protected Const z As Short = 3
End Class
```

The accessibility domain of the type of the constant must be the same as or a superset of the accessibility domain of the constant itself. The constant expression must yield a value of the constant's type or of a type that is implicitly convertible to the constant's type. The constant expression may not be circular; that is, a constant may not be defined in terms of itself.

The compiler automatically evaluates the constant declarations in the appropriate order. In the following example, the compiler first evaluates `Y`, then `Z`, and finally `X`, producing the values 10, 11, and 12, respectively.

```
Class A
    Public Const X As Integer = B.Z + 1
    Public Const Y As Integer = 10
End Class

Class B
    Public Const Z As Integer = A.Y + 1
End Class
```

When a symbolic name for a constant value is desired, but the type of the value is not permitted in a constant declaration or when the value cannot be computed at compile time by a constant expression, a read-only variable may be used instead.

```
ConstantMemberDeclaration ::=
   [ Attributes ] [ ConstantModifier+ ] Const
      ConstantDeclarators LineTerminator
```

```
ConstantModifier ::= AccessModifier | Shadows
ConstantDeclarators ::=
   ConstantDeclarator |
   ConstantDeclarators , ConstantDeclarator
ConstantDeclarator ::= Identifier [ As TypeName ] =
   ConstantExpression LineTerminator
```

See Also

9.2 Methods | 9.3 Constructors | 9.4 Events | 9.6 Instance and Shared Variables | 9.7 Properties | 4.6 Accessibility | 4.3.3 Shadowing | 2.2 Identifiers | Constants and Enumerations (Visual Basic Language Reference) | Constants and Enumerations (Visual Basic Language Concepts)

9.6 Instance and Shared Variables

An instance or shared variable is a member of a type that can store information. The **Dim** modifier must be specified if no modifiers are specified, but may be omitted otherwise. A single variable declaration may include multiple variable declarators; each variable declarator introduces a new instance or shared member. If an initializer is specified, only one instance or shared variable may be declared by the variable declarator.

A variable declared with the **Shared** modifier is a *shared variable*. A shared variable identifies exactly one storage location regardless of the number of instances of the type that are created. A shared variable comes into existence when a program begins executing, and ceases to exist when the program terminates.

A variable declared without the **Shared** modifier is called an *instance variable*. Every instance of a class contains a separate copy of all instance variables of the class. An instance variable of a reference type comes into existence when a new instance of that type is created, and ceases to exist when there are no references to that instance and the **Finalize** method has executed. An instance variable of a value type has exactly the same lifetime as the variable to which it belongs. In other words, when a variable of a value type comes into existence or ceases to exist, so does the instance variable of the value type.

If the declarator contains an **As** clause, the clause specifies the type of the members introduced by the declaration. If the type is omitted and strict semantics are being used, a compile-time error occurs. Otherwise the type of the members is implicitly **Object** or the type of the members' type character.

> **Note** There is no ambiguity in the syntax: if a declarator omits a type, it will always use the type of a following declarator.

The accessibility domain of an instance or shared variable's type or array element type must be the same as or a superset of the accessibility domain of the instance or shared variable itself.

The following example shows a `Color` class that has internal instance variables named `redPart`, `greenPart`, and `bluePart`:

```
Class Color
    Friend redPart As Short
    Friend bluePart As Short
    Friend greenPart As Short

    Public Sub New(red As Short, blue As Short, green As Short)
        redPart = red
        bluePart = blue
        greenPart = green
    End Sub
End Class
```

```
VariableMemberDeclaration ::=
   [ Attributes ] [ VariableModifier+ ] [ Dim ] VariableDeclarators
      LineTerminator
VariableModifier ::=
   AccessModifier |
   Shadows |
   Shared |
   ReadOnly |
   WithEvents
VariableDeclarators ::=
   VariableDeclarator |
   VariableDeclarators , VariableDeclarator
VariableDeclarator ::=
   VariableIdentifiers [ As TypeName ] |
   VariableIdentifier [ As [ New ] TypeName [ ( ArgumentList ) ] ]
      [ = VariableInitializer ]
VariableIdentifiers ::=
   VariableIdentifier |
   VariableIdentifiers , VariableIdentifier
VariableIdentifier ::= Identifier [ ArrayNameModifier ]
```

See Also

9.6.1 Read-Only Variables | 9.6.2 WithEvents Variables | 9.6.3 Variable Initializers | 9.2 Methods | 9.3 Constructors | 9.4 Events | 9.5 Constants | 9.7 Properties | 4.6 Accessibility | 4.3.3 Shadowing |
Declarations and Constants Summary (Visual Basic Language Reference) | Variables (Visual Basic Language Concepts)

9.6.1 Read-Only Variables

When an instance or shared variable declaration includes a **ReadOnly** modifier, assignments to the variables introduced by the declaration may only occur as part of the declaration or in a constructor in the same class. Specifically, assignments to a read-only instance or shared variable are permitted only in the following situations:

- In the variable declaration that introduces the instance or shared variable (by including a variable initializer in the declaration).
- For an instance variable, in the instance constructors of the class that contains the variable declaration. The instance variable can only be accessed in an unqualified manner or through **Me** or **MyClass**.
- For a shared variable, in the shared constructor of the class that contains the shared variable declaration.

A shared read-only variable is useful when a symbolic name for a constant value is desired, but when the type of the value is not permitted in a constant declaration, or when the value cannot be computed at compile time by a constant expression.

An example of the first such application follows, in which color shared variables are declared **ReadOnly** to prevent them from being changed by other programs:

```
Class Color
    Friend redPart As Short
    Friend bluePart As Short
    Friend greenPart As Short

    Public Sub New(red As Short, blue As Short, green As Short)
        redPart = red
        bluePart = blue
        greenPart = green
    End Sub

    Public Shared ReadOnly Red As Color = New Color(&HFF, 0, 0)
    Public Shared ReadOnly Blue As Color = New Color(0, &HFF, 0)
    Public Shared ReadOnly Green As Color = New Color(0, 0, &HFF)
    Public Shared ReadOnly White As Color = New Color(&HFF, &HFF, &HFF)
End Class
```

Constants and read-only shared variables have different semantics. When an expression references a constant, the value of the constant is obtained at compile time, but when an expression references a read-only shared variable, the value of the shared variable is not obtained until run time. Consider the following application, which consists of two separate programs.

file1.vb:

```
Namespace Program1
    Public Class Utils
        Public Shared ReadOnly X As Integer = 1
    End Class
End Namespace
```

file2.vb:

```
Imports System

Namespace Program2
    Module Test
        Sub Main()
            Console.WriteLine(Program1.Utils.X)
        End Sub
    End Module
End Namespace
```

The namespaces `Program1` and `Program2` denote two programs that are compiled separately. Because variable `Program1.Utils.X` is declared as **Shared ReadOnly**, the value output by the `Console.WriteLine` statement is not known at compile time, but rather is obtained at run time. Thus, if the value of `X` is changed and `Program1` is recompiled, the `Console.WriteLine` statement will

output the new value even if `Program2` is not recompiled. However, if `X` had been a constant, the value of `X` would have been obtained at the time `Program2` was compiled, and would have remained unaffected by changes in `Program1` until `Program2` was recompiled.

See Also

9.6.2 WithEvents Variables | 9.6.3 Variable Initializers | 9.6 Instance and Shared Variables | 4.6 Accessibility | 4.3.3 Shadowing | Declarations and Constants Summary (Visual Basic Language Reference) | ReadOnly (Visual Basic Language Reference) | Variables (Visual Basic Language Concepts)

9.6.2 WithEvents Variables

A type can declare that it handles some set of events raised by one of its instance or shared variables by declaring the instance or shared variable that raises the events with the **WithEvents** modifier. For example:

```
Imports System

Class Raiser
    Public Event Constructed()

    Public Sub New()
        RaiseEvent Constructed
    End Sub
End Class

Module Test
    Private WithEvents x As Raiser

    Private Sub HandleConstructed() Handles x.Constructed
        Console.WriteLine("Constructed")
    End Sub

    Public Sub Main()
        x = New Raiser()
    End Sub
End Module
```

In this example, the method `HandleConstructed` handles the event `Constructed` that is raised by the instance of the type `Raiser` stored in the instance variable `x`.

The **WithEvents** modifier causes the variable to be renamed with a leading underscore and replaced with a property of the same name that does the event hookup. For example, if the variable's name is `F`, it is renamed to `_F` and a property `F` is implicitly declared. If there is a collision between the variable's new name and another declaration, a compile-time error will be reported.

The implicit property created by a **WithEvents** declaration takes care of hooking and unhooking the relevant event handlers. When a value is assigned to the variable, the property first calls the `remove` method for the event on the instance currently in the variable (unhooking the existing event handler, if any). Next the assignment is made, and the property calls the `add` method for the event on the new instance in the variable (hooking up the new event handler). The following code is equivalent to the code above for the standard module `Test`:

```
Module Test
    Private _x As Raiser

    Public Property x() As Raiser
        Get
            Return _x
        End Get

        Set (ByVal Value As Raiser)
            ' Unhook any existing handlers.
            If Not _x Is Nothing Then
                RemoveHandler _x.Constructed, AddressOf HandleConstructed
            End If

            ' Change value.
            _x = Value

            ' Hook-up new handlers.
            If Not _x Is Nothing Then
                AddHandler _x.Constructed, AddressOf HandleConstructed
            End If
        End Set
    End Property

    Sub HandleConstructed()
```

```
        Console.WriteLine("Constructed")
    End Sub

    Sub Main()
        x = New Raiser()
    End Sub
End Module
```

It is not valid to declare an instance or shared variable as **WithEvents** if it does not raise any events or if the variable is typed as a structure. In addition, **WithEvents** may not be specified in a structure, and **WithEvents** and **ReadOnly** cannot be combined.

See Also

9.6.1 Read-Only Variables | 9.6.3 Variable Initializers | 9.6 Instance and Shared Variables | 4.6 Accessibility | 4.3.3 Shadowing | Declarations and Constants Summary (Visual Basic Language Reference) | WithEvents (Visual Basic Language Reference) | Variables (Visual Basic Language Concepts)

9.6.3 Variable Initializers

Instance and shared variable declarations in classes (but not structures) may include variable initializers. For **Shared** variables, variable initializers correspond to assignment statements that are executed after the program begins, but before the **Shared** variable is first referenced. For instance variables, variable initializers correspond to assignment statements that are executed when an instance of the class is created. Structures cannot have variable initializers because their parameterless constructors cannot be modified.

Consider the following example:

```
Imports System

Class Test
    Public Shared x As Double = Math.Sqrt(2.0)
    Public i As Integer = 100
    Public s As String = "Hello"
End Class

Module TestModule
    Sub Main()
        Dim a As Test = New Test()

        Console.WriteLine("x = " & x & ", i = " & a.i & ", s = " & a.s)
    End Sub
End Module
```

The example produces the following output:

```
x = 1.414213562373095, i = 100, s = Hello
```

An assignment to `x` occurs when the class is loaded, and assignments to `i` and `s` occur when a new instance of the class is created.

It is useful to think of variable initializers as assignment statements that are automatically inserted in the block of the type's constructor. The following example contains several instance variable initializers.

```
Imports System.Collections

Class A
    Private x As Integer = 1
    Private y As Integer = -1
    Private count As Integer

    Public Sub New()
        count = 0
    End Sub

    Public Sub New(n As Integer)
        count = n
    End Sub
End Class

Class B
    Inherits A

    Private sqrt2 As Double = Math.Sqrt(2.0)
    Private items As ArrayList = New ArrayList(100)
    Private max As Integer

    Public Sub New()
        Me.New(100)
        items.Add("default")
    End Sub

    Public Sub New(n As Integer)
```

```
        MyBase.New(n – 1)
        max = n
    End Sub
End Class
```

The example corresponds to the code shown below, where each comment indicates an automatically inserted statement.

```
Imports System.Collections

Class A
    Private x, y, count As Integer

    Public Sub New()
        MyBase.New ' Invoke object() constructor.
        x = 1 ' This is a variable initializer.
        y = -1 ' This is a variable initializer.
        count = 0
    End Sub

    Public Sub New(n As Integer)
        MyBase.New ' Invoke object() constructor.
        x = 1 ' This is a variable initializer.
        y = - 1 ' This is a variable initializer.
        count = n
    End Sub
End Class

Class B
    Inherits A

    Private sqrt2 As Double
    Private items As ArrayList
    Private max As Integer

    Public Sub New()
        Me.New(100)
        items.Add("default")
    End Sub

    Public Sub New(n As Integer)
        MyBase.New(n - 1)
        sqrt2 = Math.Sqrt(2.0) ' This is a variable initializer.
        items = New ArrayList(100) ' This is a variable initializer.
        max = n
    End Sub
End Class
```

All variables are initialized to the default value of their type before any variable initializers are executed. For example:

```
Imports System

Class Test
    Public Shared b As Boolean
    Public i As Integer
End Class

Module TestModule
    Sub Main()
        Dim t As Test = New Test()
        Console.WriteLine("b = " & b & ", i = " & t.i)
    End Sub
End Module
```

Because `b` is automatically initialized to its default value when the class is loaded and `i` is automatically initialized to its default value when an instance of the class is created, the preceding code produces the following output:

```
b = False, i = 0
```

Each variable initializer must yield a value of the variable's type or of a type that is implicitly convertible to the variable's type. A variable initializer may be circular or refer to a variable that will be initialized after it, in which case the value of the referenced variable is its default value for the purposes of the initializer. Such an initializer is of dubious value.

There are four forms of variable initializers: regular initializers, array-element initializers, array-size initializers, and object initializers. The first two forms appear after an equal sign that follows the type name, the latter two are part of the declaration itself. Only one form of initializer may be used on any particular declaration.

```
VariableInitializer ::= RegularInitializer | ArrayElementInitializer
```

See Also

9.6.3.1 Regular Initializers | 9.6.3.4 Array-Element Initializers | 9.6.3.3 Array-Size Initializers | | 9.6.1 Read-Only Variables | 9.6.2 WithEvents Variables | 9.6 Instance and Shared Variables | 4.6 Accessibility | 4.3.3 Shadowing | Declarations and Constants Summary (Visual Basic Language Reference) | Variables (Visual Basic Language Concepts)

9.6.3.1 Regular Initializers

A regular initializer is an expression that is implicitly convertible to the type of the variable. It appears after an equal sign that follows the type name and must be classified as a value. For example:

```
Imports System

Module Test
    Dim x As Integer = 10
    Dim y As Integer = 20

    Sub Main()
        Console.WriteLine("x = " & x & ", y = " & y)
    End Sub
End Module
```

This program produces the output:

```
x = 10, y = 20
```

```
RegularInitializer ::= Expression
```

See Also

9.6.3.4 Array-Element Initializers | 9.6.3.3 Array-Size Initializers | | 9.6.3 Variable Initializers | 4.6 Accessibility | 4.3.3 Shadowing | 9.6 Instance and Shared Variables | 9.6.1 Read-Only Variables | 9.6.2 WithEvents Variables | 11. Expressions | Declarations and Constants Summary (Visual Basic Language Reference) | Variables (Visual Basic Language Concepts)

9.6.3.2 Object Initializers

An object initializer is specified using the **New** keyword before the type name and an optional parameter list after the type name. An object initializer is equivalent to a regular initializer of the form `= New T(A)`, where `T` is the type name and `A` is the supplied formal parameter list, if any. So

```
Module TestModule
    Sub Main()
        Dim x As New Test(10)
    End Sub
End Module
```

is equivalent to

```
Module TestModule
    Sub Main()
        Dim x As Test = New Test(10)
    End Sub
End Module
```

The parenthesis in an object initializer is always interpreted as the parameters to the constructor and never as array type modifiers.

See Also

9.6.3.1 Regular Initializers | 9.6.3.4 Array-Element Initializers | 9.6.3.3 Array-Size Initializers | 9.6.3 Variable Initializers | | 4.3.3 Shadowing | 9.6 Instance and Shared Variables | 9.6.1 Read-Only Variables | 9.6.2 WithEvents Variables | Declarations and Constants Summary (Visual Basic Language Reference) | Variables (Visual Basic Language Concepts)

9.6.3.3 Array-Size Initializers

An array-size initializer is a modifier on the name of the variable that gives a set of dimension upper bounds denoted by expressions. For shared and instance variables, the upper-bound expressions must be constant; otherwise the expressions must be classified as values. The upper bound expressions must be implicitly convertible to **Integer**. The set of upper bounds is equivalent to a variable initializer of an array-creation expression with the given upper bounds. The number of dimensions of the array type is inferred from the array size initializer. So

```
Module Test
    Sub Main()
        Dim x(5, 10) As Integer
    End Sub
End Module
```

is equivalent to

```
Module Test
    Sub Main()
        Dim x As T(,) = new Integer(5, 10) {}
    End Sub
End Module
```

All upper bounds must be equal to or greater than -1, and all dimensions must have an upper bound specified. If the element type of the array being initialized is itself an array type, the array-type modifiers go to the right of the array-size initializer. For example

```
Module Test
    Sub Main()
        Dim x(5,10)(,,) As Integer
    End Sub
End Module
```

declares a local variable `x` whose type is a two-dimensional array of three-dimensional arrays of **Integer**, initialized to an array with bounds of `0..5` in the first dimension and `0..10` in the second dimension. It is not possible to use an array size initializer to initialize the elements of a variable whose type is an array of arrays.

If an array element initializer is specified, each expression in the upper bound list must be a constant expression. Furthermore, the rank and dimension lengths specified by the rank initialization list must match those of the array element initializer. A variable declaration may not include both an array type modifier on its type and an array-size initializer.

```
ArraySizeInitializationModifier ::=
   ( UpperBoundList ) [ ArrayTypeModifiers ]
UpperBoundList::=
   Expression |
   UpperBoundList , Expression
```

See Also

9.6.3.1 Regular Initializers | 9.6.3.4 Array-Element Initializers | 9.6.3.2 Object Initializers | 4.6 Accessibility | 4.3.3 Shadowing | 9.6 Instance and Shared Variables | 9.6.1 Read-Only Variables | 9.6.2 WithEvents Variables | 9.6.3 Variable Initializers | 11. Expressions | Declarations and Constants Summary (Visual Basic Language Reference) | Variables (Visual Basic Language Concepts)

9.6.3.4 Array-Element Initializers

An array-element initializer consists of a sequence of variable initializers, enclosed by curly braces ({}) and separated by commas. Each variable initializer is an expression or, in the case of a multidimensional array, a nested array-element initializer. Each expression must be classified as a value. Array-element initializers may also be used in array-creation expressions.

The type of expression or statement in which an array-element initializer is used determines the type of the array being initialized. In an array-creation expression, the array type immediately precedes the initializer. In a variable declaration, the array type is the type of the variable being declared. When an array-element initializer is used in a variable declaration, such as:

```
Private a As Integer() = { 0, 2, 4, 6, 8 }
```

it is simply shorthand for an equivalent array-creation expression:

```
Private a As Integer() = New Integer() { 0, 2, 4, 6, 8 }
```

In an array-element initializer, the outermost nesting level corresponds to the leftmost dimension, and the innermost nesting level corresponds to the rightmost dimension. The initializer must have the same number levels of nesting as there are dimensions in the array. All of the elements in the innermost nesting level must be implicitly convertible to the element type of the array. The number of elements in each nested array-element initializer must always be consistent with the size of the other array-element initializers at the same level.

The following example creates a two-dimensional array with a length of five for the leftmost dimension and a length of two for the rightmost dimension:

```
Module Test
    Sub Main()
        Dim b As Integer(,) = _
            { { 0, 1 }, { 2, 3 }, { 4, 5 }, { 6, 7 }, { 8, 9 } }
    End Sub
End Module
```

The example is equivalent to the following:

```
Module Test
    Sub Main()
        Private b(4, 1) As Integer

        b(0, 0) = 0: b(0, 1) = 1
        b(1, 0) = 2: b(1, 1) = 3
        b(2, 0) = 4: b(2, 1) = 5
        b(3, 0) = 6: b(3, 1) = 7
        b(4, 0) = 8: b(4, 1) = 9
    End Sub
End Module
```

If the array-creation expression specifies the bounds of the dimensions, the number of elements at any particular level must be the same as the size of the corresponding dimension. If the bounds are unspecified, the length of each dimension is the number of elements in the corresponding level of nesting.

Some valid and invalid examples follow:

```
' OK.
Private x() As Integer = New Integer(2) {0, 1, 2}
' Error, length/initializer mismatch.
Private y() As Integer = New Integer(2) {0, 1, 2, 3}
```

Here, the initializer for `y` is in error because the length and the number of elements in the initializer do not agree.

An empty array-element initializer (that is, one that contains curly braces but no initializer list) is always valid regardless of the number of dimensions of the array. If the size of the dimensions of the array being initialized is known in an array-creation

expression, the empty array-element initializer represents an array instance of the specified size where all the elements have been initialized to the element type's default value. If the dimensions of the array being initialized are not known, the empty array-element initializer represents an array instance in which all dimensions are size zero.

Because context is required to determine the type of an array initializer, it is not possible to use an array initializer in an expression context. Therefore, the following code is not valid:

```
Module Test
    Sub F(ByVal a() As Integer)
    End Sub

    Sub Main()
        ' Error, can't use without array creation expression.
        F({1, 2, 3})

        ' OK.
        F(New Integer() {1, 2, 3})
    End Sub
End Module
```

At run time, the expressions in an array-element initializer are evaluated in textual order from left to right.

```
ArrayElementInitializer ::= { [ VariableInitializerList ] }
VariableInitializerList ::=
   VariableInitializer |
   VariableInitializerList , VariableInitializer
VariableInitializer ::= Expression | ArrayElementInitializer
```

See Also

9.6.3.1 Regular Initializers | 9.6.3.3 Array-Size Initializers | | 9.6.3 Variable Initializers | 4.6 Accessibility | 4.3.3 Shadowing | 9.6 Instance and Shared Variables | 9.6.1 Read-Only Variables | 9.6.2 WithEvents Variables | 11. Expressions | Declarations and Constants Summary (Visual Basic Language Reference) | Variables (Visual Basic Language Concepts)

9.6.4 System.MarshalByRefObject Classes

Classes that derive from the class **System.MarshalByRefObject** are marshaled across context boundaries using proxies (that is, by reference) rather than through copying (that is, by value). This means that an instance of such a class may not be a true instance but instead may just be a stub that marshals variable accesses and method calls across a context boundary.

As a result, it is not possible to create a reference to the storage location of variables defined on such classes. This means that variables typed as classes derived from **System.MarshalByRefObject** cannot be passed to reference parameters, and methods and variables of variables typed as value types may not be accessed. Instead, Visual Basic .NET treats variables defined on such classes as if they were properties (since the restrictions are the same on properties).

There is one exception to this rule: a member implicitly or explicitly qualified with **Me** is exempt from the above restrictions, because **Me** is always guaranteed to be an actual object, not a proxy.

See Also

9.6.1 Read-Only Variables | 9.6.2 WithEvents Variables | 9.6.3 Variable Initializers | 9.6 Instance and Shared Variables | 11.4.3 Instance Expressions

9.7 Properties

Properties are a natural extension of variables; both are named members with associated types, and the syntax for accessing variables and properties is the same. Unlike variables, however, properties do not denote storage locations. Instead, properties have *accessors*, which specify the statements to execute in order to read or write their values.

Properties are defined with property declarations. The first part of a property declaration resembles a field declaration. The second part includes a **Get** accessor and/or a **Set** accessor. In the example below, the `Button` class defines a `Caption` property.

```
Public Class Button
    Private captionValue As String

    Public Property Caption() As String
        Get
            Return captionValue
        End Get

        Set (ByVal Value As String)
            captionValue = value
            Repaint()
        End Set
    End Property
End Class
```

Based on the `Button` class above, the following is an example of use of the `Caption` property:

```
Dim okButton As Button = New Button()

okButton.Caption = "OK" ' Invokes Set accessor.
Dim s As String = okButton.Caption ' Invokes Get accessor.
```

Here, the **Set** accessor is invoked by assigning a value to the property, and the **Get** accessor is invoked by referencing the property in an expression.

If no type is specified for a property and strict semantics are being used, a compile-time error occurs; otherwise the type of the property is implicitly **Object** or the type of the property's type character. A property declaration may contain either a **Get** accessor, which retrieves the value of the property, a **Set** accessor, which stores the value of the property, or both. Because a property implicitly declares methods, a property may be declared with the same modifiers as a method. If the property is defined in an interface or defined with the **MustOverride** modifier, the property body and the **End** construct must be omitted; otherwise, a compile-time error occurs.

The index parameter list makes up the signature of the property, so properties may be overloaded on index parameters but not on the type of the property. The index parameter list is the same as for a regular method. However, none of the parameters may be modified with the **ByRef** modifier and none of them may be named `Value` (which is reserved for the implicit value parameter in the **Set** accessor).

A property may be declared as follows:

- If the property specifies no property type modifier, the property must have both a **Get** accessor and a **Set** accessor. The property is said to be a read-write property.
- If the property specifies the **ReadOnly** modifier, the property must have a **Get** accessor and may not have a **Set** accessor. The property is said to be read-only property. It is a compile-time error for a read-only property to be the target of an assignment.
- If the property specifies the **WriteOnly** modifier, the property must have a **Set** accessor and may not have a **Get** accessor. The property is said to be write-only property. It is a compile-time error to reference a write-only property in an expression except as the target of an assignment or as an argument to a method.

The **Get** and **Set** accessors of a property are not distinct members, and it is not possible to declare the accessors of a property separately. The following example does not declare a single read-write property. Rather, it declares two properties with the same name, one read-only and one write-only:

```
Class A
    Private nameValue As String
```

```
    ' Error, contains a duplicate member name.
    Public ReadOnly Property Name() As String
        Get
            Return nameValue
        End Get
    End Property

    ' Error, contains a duplicate member name.
    Public WriteOnly Property Name() As String
        Set (ByVal Value As String)
            nameValue = value
        End Set
    End Property
End Class
```

Since two members declared in the same class cannot have the same name, the example causes a compile-time error.

When a derived type shadows a property, the derived property hides the shadowed property with respect to both reading and writing. In the following example, the `P` property in `B` hides the `P` property in `A` with respect to both reading and writing:

```
Class A
    Public WriteOnly Property P() As Integer
        Set (ByVal Value As Integer)
        End Set
    End Property
End Class

Class B
    Inherits A

    Public Shadows ReadOnly Property P() As Integer
       Get
       End Get
    End Property
End Class

Module Test
    Sub Main()
        Dim x As B = New B

        B.P = 10     ' Error, B.P is read-only.
    End Sub
End Module
```

The accessibility domain of the return type or parameter types must be the same as or a superset of the accessibility domain of the property itself. A property may only have one **Set** accessor and one **Get** accessor.

Except for differences in declaration and invocation syntax, **Overridable**, **NotOverridable**, **Overrides**, **MustOverride**, and **MustInherit** properties behave exactly like **Overridable**, **NotOverridable**, **Overrides**, **MustOverride**, and **MustInherit** methods. When a property is overridden, the overriding property must be of the same type (read-write, read-only, write-only).

In the following example `X` is an **Overridable** read-only property, `Y` is an **Overridable** read-write property, and `Z` is a **MustOverride** read-write property.

```
MustInherit Class A
    Private y As Integer

    Public Overridable ReadOnly Property X() As Integer
        Get
            Return 0
        End Get
    End Property

    Public Overridable Property Y() As Integer
        Get
            Return y
```

```
         End Get
        Set (ByVal Value As Integer)
            y = value
        End Set
    End Property

    Public MustOverride Property Z() As Integer
End Class
```

Because `Z` is **MustOverride**, the containing class `A` must be declared **MustInherit**.

By contrast, a class that derives from class `A` is shown below:

```
Class B
    Inherits A

    Private zValue As Integer

    Public Overrides ReadOnly Property X() As Integer
        Get
            Return MyBase.X + 1
        End Get
    End Property

    Public Overrides Property Y() As Integer
        Get
            Return MyBase.Y
        End Get
        Set (ByVal Value As Integer)
            If value < 0 Then
                MyBase.Y = 0
            Else
                MyBase.Y = Value
            End If
        End Set
    End Property

    Public Overrides Property Z() As Integer
        Get
            Return zValue
        End Get
        Set (ByVal Value As Integer)
            zValue = Value
        End Set
    End Property
End Class
```

Here, the declarations of properties `X`, `Y`, and `Z` override the base properties. Each property declaration exactly matches the accessibility modifiers, type, and name of the corresponding inherited property. The **Get** accessor of property `X` and the **Set** accessor of property `Y` use the **MyBase** keyword to access the inherited properties. The declaration of property `Z` overrides the **MustOverride** property — thus, there are no outstanding **MustOverride** members in class `B`, and `B` is permitted to be a regular class.

Properties can be used to delay initialization of a resource until the moment it is first referenced. For example:

```
Imports System.IO

Public Class ConsoleStreams
    Private Shared reader As TextReader
    Private Shared writer As TextWriter
    Private Shared errors As TextWriter

    Public Shared ReadOnly Property [In]() As TextReader
        Get
            If reader Is Nothing Then
                reader = New StreamReader(File.OpenStandardInput())
```

```
            End If
            Return reader
        End Get
    End Property

    Public Shared ReadOnly Property Out() As TextWriter
        Get
            If writer Is Nothing Then
                writer = New StreamWriter(File.OpenStandardOutput())
            End If
            Return writer
        End Get
    End Property

    Public Shared ReadOnly Property [Error]() As TextWriter
        Get
            If errors Is Nothing Then
                errors = New StreamWriter(File.OpenStandardError())
            End If
            Return errors
        End Get
    End Property
End Class
```

The `ConsoleStreams` class contains three properties, `In`, `Out`, and `Error`, that represent the standard input, output, and error devices, respectively. By exposing these members as properties, the `ConsoleStreams` class can delay their initialization until they are actually used. For example, upon first referencing the `Out` property, as in `ConsoleStreams.Out.WriteLine("hello, world")`, the underlying `TextWriter` for the output device is created. But if the application makes no reference to the `In` and `Error` properties, then no objects are created for those devices.

```
PropertyMemberDeclaration ::=
   [ Attributes ] [ PropertyModifier+ ] Property Identifier
      [ ( [ ParameterList ] ) ] [ As TypeName ] [ ImplementsClause ]
      LineTerminator
   [ PropertyAccessorDeclaration+ ]
   [ End Property LineTerminator ]
PropertyModifier ::= ProcedureModifier | Default | ReadOnly | WriteOnly
PropertyAccessorDeclaration ::=
   PropertyGetDeclaration |
   PropertySetDeclaration
```

See Also

9.7.1 Get Accessor Declarations | 9.7.2 Set Accessor Declarations | 9.7.3 Default Properties | 9.2 Methods | 9.3 Constructors | 9.4 Events | 9.5 Constants | 9.6 Instance and Shared Variables | Properties (Visual Basic Language Reference) | Setting and Retrieving Properties (Visual Basic Language Concepts)

9.7.1 Get Accessor Declarations

A **Get** accessor (getter) is declared by using a property **Get** declaration. A property **Get** declaration consists of the keyword **Get** followed by a statement block. Given a property named `P`, a **Get** accessor declaration implicitly declares a method with the name `get_P` with the same modifiers, type, and parameter list as the property. If the type contains a declaration with that name, a compile-time error results.

A special local variable, which is implicitly declared in the **Get** accessor body's declaration space with the same name as the property, represents the return value of the property. The local variable has special name resolution semantics when used in expressions. If the local variable is used in a context that expects an expression that is classified as a method group, such as an invocation expression, then the name resolves to the function rather than to the local variable. For example:

```
ReadOnly Property F(ByVal i As Integer) As Integer
    Get
        If i = 0 Then
            F = 1    ' Sets the return value.
        Else
            F = F(i – 1) ' Recursive call.
        End If
    End Get
End Property
```

The use of parentheses can cause ambiguous situations (such as `F(1)` where `F` is a property whose type is a one-dimensional array). In all ambiguous situations, the name resolves to the property rather than the local variable. For example:

```
ReadOnly Property F(ByVal i As Integer) As Integer()
    Get
        If i = 0 Then
            F = new Integer(3) { 1, 2, 3 }
        Else
            F = F(i – 1) ' Recursive call, not index.
        End If
    End Get
End Property
```

When control flow leaves the **Get** accessor body, the value of the local variable is passed back to the invocation expression. Because invoking a **Get** accessor is conceptually equivalent to reading the value of a variable, it is considered bad programming style for **Get** accessors to have observable side effects, as illustrated in the following example:

```
Class Counter
    Private Value As Integer

    Public ReadOnly Property NextValue() As Integer
        Get
            Value += 1
            Return Value
        End Get
    End Property
End Class
```

The value of the `NextValue` property depends on the number of times the property has previously been accessed. Thus, accessing the property produces an observable side effect, and the property should instead be implemented as a method.

The "no side effects" convention for **Get** accessors does not mean that **Get** accessors should always be written to simply return values stored in variables. Indeed, **Get** accessors often compute the value of a property by accessing multiple variables or invoking methods. However, a properly designed **Get** accessor performs no actions that cause observable changes in the state of the object.

```
PropertyGetDeclaration ::=
   [ Attributes ] Get LineTerminator
   [ Block ]
   End Get
```

See Also

9.7.2 Set Accessor Declarations | 9.7.3 Default Properties |9.7 Properties |
Setting and Retrieving Properties (Visual Basic Language Concepts)

9.7.2 Set Accessor Declarations

A **Set** accessor (setter) is declared by using a property set declaration. A property set declaration consists of the keyword **Set**, an optional parameter list, and a statement block. Given a property named `P`, a setter declaration implicitly declares a method with the name `set_P` with the same modifiers and parameter list as the property. If the type contains a declaration with that name, a compile-time error results.

If a parameter list is specified, it must have one member, that member must have no modifiers except **ByVal**, and its type must be the same as the type of the property. The parameter represents the property value being set. If the parameter is omitted, a parameter named **Value** is implicitly declared.

```
PropertySetDeclaration ::=
   [ Attributes ] Set [ ( ParameterList ) ] LineTerminator
   [ Block ]
   End Set
```

See Also

9.7.1 Get Accessor Declarations | 9.7.3 Default Properties | 9.7 Properties |
Setting and Retrieving Properties (Visual Basic Language Concepts)

9.7.3 Default Properties

A property that specifies the modifier **Default** is called a *default property*. Any type that allows properties may have a default property, including interfaces. The default property may be referenced without having to qualify the instance with the name of the property. Thus, given a class

```
Class Test
    Public Default ReadOnly Property Item(ByVal i As Integer) As Integer
        Get
            Return i
        End Get
    End Property
End Class
```

the code

```
Module TestModule
    Sub Main()
        Dim x As Test = New Test()
        Dim y As Integer

        y = x(10)
    End Sub
End Module
```

is equivalent to

```
Module TestModule
    Sub Main()
        Dim x As Test = New Test()
        Dim y As Integer

        y = x.Item(10)
    End Sub
End Module
```

Default properties may be overloaded, but all default properties in a type must have the same name. If a derived type declares a default property with a different name than the default property of a base type, then a warning is issued and the default property in the derived type shadows the base type default property. Similarly, if a derived type contains a non-default property that shadows a default property in a base type, the base type default property is not shadowed. For example:

```
Imports System

Class Base
    Public ReadOnly Default Property Item(ByVal i As Integer)
        Get
            Console.WriteLine("Base = " & i)
        End Get
    End Property
End Class

Class Derived
    Inherits Base

    ' This hides Item, but not the default property.
    Public Shadows ReadOnly Property Item(ByVal i As Integer)
        Get
            Console.WriteLine("Derived = " & i)
        End Get
    End Property
End Class

Class MoreDerived
```

```
    Inherits Derived

    ' This hides the default property, but not Item.
    Public Shadows ReadOnly Default Property Value(ByVal i As Integer)
        Get
            Console.WriteLine("MoreDerived = " & i)
        End Get
    End Property
End Class

Module Test
    Sub Main()
        Dim x As MoreDerived = New MoreDerived()
        Dim y As Integer
        Dim z As Derived = x

        y = x(10)   ' Calls MoreDerived.Value.
        y = x.Item(10)   ' Calls Derived.Item
        y = z(10)   ' Calls Base.Item
    End Sub
End Module
```

This program will produce the output:

```
MoreDerived = 10
Derived = 10
Base = 10
```

Because a default property with no index parameters would cause an ambiguous situation when assigning instances of the containing class, default properties must have index parameters. Furthermore, if one property overloaded on a particular name includes the **Default** modifier, all properties overloaded on that name must specify it. Default properties may not be shared or private.

See Also

9.7 Properties | 9.7.1 Get Accessor Declarations | 9.7.2 Set Accessor Declarations | 11.9 Index Expressions | Default (Visual Basic Language Reference) | Setting and Retrieving Properties (Visual Basic Language Concepts)

10. Statements

Statements represent executable code.

A method is executed by first initializing all of its parameters to their correct values and initializing all of its local variables to the default value of their types. After parameter and local variable initialization, the method body block is executed. After the method block has been executed, execution returns to the caller of the method.

```
Statement ::=
   LocalDeclarationStatement |
   WithStatement |
   SyncLockStatement |
   EventStatement |
   AssignmentStatement |
   InvocationStatement |
   ConditionalStatement |
   LoopStatement |
   ErrorHandlingStatement |
   BranchStatement |
   ArrayHandlingStatement
StatementTerminator ::= LineTerminator | :
```

See Also

10.1 Blocks | 10.2 Local Declaration Statements | 10.3 With Statement | 10.4 SyncLock Statement | 10.5 Event Statements | 10.6 Assignment Statements | 10.7 Invocation Statements | 10.8 Conditional Statements | 10.9 Loop Statements | 10.10 Exception-Handling Statements | 10.11 Branch Statements | Statements (Visual Basic Language Reference)

10.1 Blocks

Statements are organized into blocks, which are made up of logical lines. Each logical line begins with an optional label declaration, followed by zero or more statements delimited by colons.

Labels have their own declaration space and do not interfere with other identifiers. The following example is valid and uses the name variable `x` both as a parameter and as a label.

```
Function F(ByVal x As Integer) As Integer
    If x >= 0 Then
        GoTo x
    End If
    x = -x
x:
    Return x
End Function
```

The scope of a label is the body of the method containing it.

> **Note** Because both labels and statements are delimited with colons, it is possible for a label declaration to look like a statement; label declaration always takes precedence in any ambiguous situation.

Labels may be either identifiers or integer literals.

Execution of a block begins with the first statement in the block. Once a statement has been executed, the next statement in lexical order is executed, unless a statement transfers execution elsewhere or an exception occurs.

For the sake of readability, statement productions that involve multiple substatements are treated as a single production in this specification, even though the substatements may each be by themselves on a labeled line.

```
Block ::= [ LabeledLine+ ]
LabeledLine ::= [ LabelName : ] [ Statements ] LineTerminator
LabelName ::= Identifier | IntLiteral
Statements ::=
   [ Statement ] |
   Statements : [ Statement ]
```

See Also

10.1.1 Local Variables and Parameters | 10.2 Local Declaration Statements | 10.3 With Statement | 10.4 SyncLock Statement | 10.5 Event Statements | 10.6 Assignment Statements | 10.7 Invocation Statements | 10.8 Conditional Statements | 10.9 Loop Statements | 10.10 Exception-Handling Statements | 10.11 Branch Statements | 10.12 Array-Handling Statements

10.1.1 Local Variables and Parameters

A method invocation creates a copy, specific to that invocation, of the local variables and parameters of the method. A local variable or parameter comes into existence when control enters the method body that contains the local variable declaration or parameter declaration and ceases to exist when control leaves the method. All locals are initialized to their type's default value. Local variables and parameters are always publicly accessible. It is an error to refer to a local variable in a textual position that precedes its declaration, as the following example illustrates:

```
Class A
    Private i As Integer = 0

    Sub F()
        i = 1
        Dim i As Integer       ' Error, use precedes declaration.
        i = 2
    End Sub

    Sub G()
        Dim a As Integer = 1
        Dim b As Integer = a   ' This is valid.
    End Sub
End Class
```

In the `F` method above, the first assignment to `i` specifically does not refer to the field declared in the outer scope. Rather, it refers to the local variable, and it is in error because it textually precedes the declaration of the variable. In the `G` method, a subsequent variable declaration refers to a local variable declared in an earlier variable declaration within the same local variable declaration.

Each block in a method creates a declaration space for local variables. Names are introduced into this declaration space through local variable declarations in the method body and through the parameter list of the method, which introduces names into the outermost block's declaration space. Blocks do not allow shadowing of names through nesting: once a name has been declared in a block, the name may not be redeclared in any nested blocks.

Thus, in the following example, the `F` and `G` methods are in error because the name `i` is declared in the outer block and cannot be redeclared in the inner block. However, the `H` and `I` methods are valid because the two `i`'s are declared in separate non-nested blocks.

```
Class A
    Sub F()
        Dim i As Integer = 0
        If True Then
               Dim i As Integer = 1
        End If
    End Sub

    Sub G()
        If True Then
            Dim i As Integer = 0
        End If
        Dim i As Integer = 1
    End Sub

    Sub H()
        If True Then
            Dim i As Integer = 0
        End If
        If True Then
            Dim i As Integer = 1
        End If
    End Sub

    Sub I()
        Dim i As Integer
        For i = 0 To 9
            H()
        Next I
```

```
        Dim i As Integer
        For i = 0 To 9
            H()
        Next I
    End Sub
End Class
```

When the method is a function, a special local variable is implicitly declared in the method body's declaration space with the same name as the method representing the return value of the function. The local variable has special name resolution semantics when used in expressions. If the local variable is used in a context that expects an expression classified as a method group, such as an invocation expression, then the name resolves to the function rather than to the local variable. For example:

```
Function F(ByVal i As Integer) As Integer
    If i = 0 Then
        F = 1        ' Sets the return value.
    Else
        F = F(i - 1) ' Recursive call.
    End If
End Function
```

The use of parentheses can cause ambiguous situations (such as `F(1)`, where `F` is a function whose return type is a one-dimensional array); in all ambiguous situations, the name resolves to the function rather than the local variable. For example:

```
Function F(ByVal i As Integer) As Integer()
    If i = 0 Then
        F = new Integer(3) { 1, 2, 3 }
    Else
        F = F(i - 1) ' Recursive call, not an index.
    End If
End Function
```

When control flow leaves the method body, the value of the local variable is passed back to the invocation expression. If the method is a subroutine, there is no such implicit local variable, and control simply returns to the invocation expression.

If explicit local declaration is specified by the compilation environment or by **Option Explicit**, all local variables must be explicitly declared. Otherwise, local variables may be implicitly declared simply by using an undeclared simple name expression in an expression context that can accept an expression classified as a variable, with the first textual appearance of the identifier considered its "declaration." The exception to this rule is that a local variable may not be implicitly declared when it is the target of a function invocation expression, indexing expression, or a member access expression. Implicitly declared locals are always scoped to the entire method body. In the case of implicit declaration, the type of the variable is **Object** if no type character was attached to the implicit declaration; otherwise the type of the variable is the type of the type character.

See Also

10.1 Blocks | 10.2 Local Declaration Statements | 10.3 With Statement | 10.4 SyncLock Statement | 10.5 Event Statements | 10.6 Assignment Statements | 10.7 Invocation Statements | 10.8 Conditional Statements | 10.9 Loop Statements | 10.10 Exception-Handling Statements | 10.11 Branch Statements | 10.12 Array-Handling Statements | Variables (Visual Basic Language Concepts)

10.2 Local Declaration Statements

A local declaration statement declares a new local variable, local constant, or static variable. *Local variables* and *local constants* are equivalent to instance variables and constants scoped to the method and are declared in the same way. *Static variables* are similar to **Shared** variables and are declared using the **Static** modifier.

Static variables are locals that retain their value across invocations of the method. Static variables declared within nonshared methods are per instance: each instance of the type that contains the method has its own copy of the static variable. Static variables declared within **Shared** methods are per type; there is only one copy of the static variable for all instances. While local variables are initialized to their type's default value upon each entry into the method, static variables are only initialized to their type's default value when the type or type instance is initialized. Static variables may not be declared in structures.

Local variables, local constants, and static variables always have public accessibility and may not specify accessibility modifiers. If no type is specified on a local declaration statement and strict semantics are being used, a compile-time error occurs. Otherwise the type of the property is implicitly **Object** or the type of the property's type character.

Variable initializers on local declaration statements are equivalent to assignment statements placed at the textual location of the declaration. Thus, if execution branches over the local declaration statement, the variable initializer is not executed. If the local declaration statement is executed more than once, the variable initializer is executed an equal number of times. Static variables only execute their initializer the first time. If an exception occurs while initializing a static variable, the static variable is considered initialized with the default value of the static variable's type.

The following example shows the use of initializers:

```
Imports System

Module Test
    Sub F()
        Static x As Integer = 5

        Console.WriteLine("Static variable x = " & x)
        x += 1
    End Sub

    Sub Main()
        Dim i As Integer

        For i = 1 to 3
            F()
        Next i

        i = 3
label:
        Dim y As Integer = 8

        If i > 0 Then
            Console.WriteLine("Local variable y = " & y)
            y -= 1
            i -= 1
            Goto label
        End If
    End Sub
```

This program prints:

```
Static variable x = 5
Static variable x = 6
Static variable x = 7
Local variable y = 8
Local variable y = 8
Local variable y = 8
```

Local variables, local constants, and static variables are scoped to the statement block in which they are declared. Static variables are special in that their names may only be used once throughout the entire method. For example, it is not valid to specify two

static variable declarations with the same name even if they are in different blocks.

```
LocalDeclarationStatement ::=
   LocalModifier LocalDeclarators StatementTerminator
LocalModifier ::= Static | Dim | Const
LocalDeclarators ::=
   LocalDeclarator |
   LocalDeclarators , LocalDeclarator
LocalDeclarator ::=
   LocalIdentifiers [ As TypeName ] |
   LocalIdentifier [ As [ New ] TypeName [ ( [ ArgumentList ] ) ] ]
      [ = VariableInitializer ]
LocalIdentifiers ::=
   LocalIdentifier |
   LocalIdentifiers , LocalIdentifier
LocalIdentifier ::= Identifier [ ArrayNameModifier ]
```

See Also

10.1 Blocks | 10.1.1 Local Variables and Parameters | 10.3 With Statement | 10.4 SyncLock Statement | 10.5 Event Statements | 10.6 Assignment Statements | 10.7 Invocation Statements | 10.8 Conditional Statements | 10.9 Loop Statements | 10.10 Exception-Handling Statements | 10.11 Branch Statements | 10.12 Array-Handling Statements | Declaration Statements (Visual Basic Language Concepts)

10.3 With Statement

A **With** statement allows multiple references to an expression's members without specifying the expression multiple times. The expression must be classified as a value and is evaluated once, upon entry into the block. Within the **With** statement block, a member access expression or dictionary access expression starting with a period or an exclamation point is evaluated as if the **With** expression preceded it. For example:

```
Structure Test
    Public x As Integer

    Function F() As Integer
        Return 10
    End Sub
End Structure

Module TestModule
    Sub Main()
        Dim y As Test

        With y
            .x = 10
            Console.WriteLine(.x)
            .x = .F()
        End With
    End Sub
End Module
```

It is invalid to branch into a **With** statement block from outside of the block.

```
WithStatement ::=
   With Expression StatementTerminator
   [ Block ]
   End With StatementTerminator
```

See Also

10.1 Blocks | 10.1.1 Local Variables and Parameters | 10.2 Local Declaration Statements | 10.4 SyncLock Statement | 10.5 Event Statements | 10.6 Assignment Statements | 10.7 Invocation Statements | 10.8 Conditional Statements | 10.9 Loop Statements | 10.10 Exception-Handling Statements | 10.11 Branch Statements | 10.12 Array-Handling Statements | With...End With Statements (Visual Basic Language Reference) | With...End With Statements (Visual Basic Language Concepts)

10.4 SyncLock Statement

A **SyncLock** statement allows statements to be synchronized on an expression, which ensures that multiple threads of execution do not execute the same statements at the same time. The expression must be classified as a value and is evaluated once, upon entry to the block. When entering the **SyncLock** block, the **Shared** method **System.Monitor.Enter** is called on the specified expression, which blocks until the thread of execution has an exclusive lock on the object returned by the expression. The type of the expression in a **SyncLock** statement must be a reference type. For example:

```
Class Test
    Private count As Integer = 0

    Public Function Add() As Integer
        SyncLock Me
            count += 1
            Add = count
        End SyncLock
    End Function

    Public Function Subtract() As Integer
        SyncLock Me
            count -= 1
            Subtract = count
        End SyncLock
    End Function
End Class
```

The example above synchronizes on the specific instance of the class `Test` to ensure that no more than one thread of execution can add or subtract from the count variable at a time for a particular instance.

The **SyncLock** block is implicitly contained by a **Try** statement whose **Finally** block calls the **Shared** method **System.Monitor.Exit** on the expression. This ensures the lock is freed even when an exception is thrown. As a result, it is invalid to branch into a **SyncLock** block from outside of the block, and a **SyncLock** block is treated as a single statement for the purposes of **Resume** and **Resume Next**. The above example is equivalent to the following code:

```
Class Test
    Private count As Integer = 0

    Public Function Add() As Integer
        Try
            System.Monitor.Enter(Me)

            count += 1
            Add = count
        Finally
            System.Monitor.Exit(Me)
        End Try
    End Function

    Public Function Subtract() As Integer
        Try
            System.Monitor.Enter(Me)

            count -= 1
            Subtract = count
        Finally
            System.Monitor.Exit(Me)
        End Try
    End Function
End Class
```

```
SyncLockStatement ::=
   SyncLock Expression StatementTerminator
   [ Block ]
```

```
    End SyncLock StatementTerminator
```

See Also

10.1 Blocks | 10.1.1 Local Variables and Parameters | 10.2 Local Declaration Statements | 10.3 With Statement | 10.5 Event Statements | 10.6 Assignment Statements | 10.7 Invocation Statements | 10.8 Conditional Statements | 10.9 Loop Statements | 10.10 Exception-Handling Statements | 10.11 Branch Statements | 10.12 Array-Handling Statements | SyncLock Statement (Visual Basic Language Reference)

10.5 Event Statements

The **RaiseEvent**, **AddHandler**, and **RemoveHandler** statements raise events and handle events dynamically.

```
EventStatement ::=
   RaiseEventStatement |
   AddHandlerStatement |
   RemoveHandlerStatement
```

See Also

10.5.1 RaiseEvent Statement | 10.5.2 AddHandler and RemoveHandler Statements | 10.1 Blocks |
10.1.1 Local Variables and Parameters | 10.2 Local Declaration Statements | 10.3 With Statement | 10.4 SyncLock Statement |
10.6 Assignment Statements | 10.7 Invocation Statements | 10.8 Conditional Statements | 10.9 Loop Statements |
10.10 Exception-Handling Statements | 10.11 Branch Statements | 10.12 Array-Handling Statements |
Event Statement (Visual Basic Language Reference)

10.5.1 RaiseEvent Statement

A **RaiseEvent** statement notifies event handlers that a particular event has occurred. The simple name expression in a **RaiseEvent** statement is interpreted as a member lookup on **Me**. Thus, `RaiseEvent x` is interpreted as if it were `RaiseEvent Me.x`. The result of the expression must be classified as an event access for an event defined in the class itself; events defined on base types cannot be used in a **RaiseEvent** statement.

The **RaiseEvent** statement is processed as a call to the **Invoke** method of the event's delegate, using the supplied parameters, if any. If there are no arguments, the parentheses may be omitted. For example:

```
Imports System

Class Raiser
    Public Event Constructed(ByVal Count As Integer)

    Public Sub New()
        Static CreationCount As Integer = 0

        CreationCount += 1
        RaiseEvent Constructed(CreationCount)
    End Sub
End Class

Module Test
    Private WithEvents x As Raiser

    Private Sub Constructed(ByVal Count As Integer) Handles x.Constructed
        Console.WriteLine("Constructed instance #" & Count)
    End Sub

    Public Sub Main()
        x = New Raiser   ' Causes "Constructed instance #1" to be printed.
        x = New Raiser   ' Causes "Constructed instance #2" to be printed.
        x = New Raiser   ' Causes "Constructed instance #3" to be printed.
    End Sub
End Module
```

The class `Raiser` above is equivalent to:

```
Class Raiser
    Public Event Constructed(ByVal Count As Integer)

    Public Sub New()
        Static CreationCount As Integer = 0

        CreationCount += 1
        ConstructedEvent.Invoke(CreationCount)
    End Sub
End Class
```

```
RaiseEventStatement ::= RaiseEvent SimpleNameExpression
   [ ( [ ArgumentList ] ) ] StatementTerminator
```

See Also

10.5.2 AddHandler and RemoveHandler Statements | 10.5 Event Statements |
RaiseEvent Statement (Visual Basic Language Reference)

10.5.2 AddHandler and RemoveHandler Statements

Although most event handlers are automatically hooked up through **WithEvents** variables, it may be necessary to dynamically add and remove event handlers at run time. **AddHandler** and **RemoveHandler** statements do this.

Each statement takes two arguments: the first argument must be an expression that is classified as an event access and the second argument must be an expression that is classified as a value. The second argument's type must be the delegate type associated with the event access. For example:

```
Imports System

Public Class Form1
    Public Sub New()
        ' Add Button1_Click as an event handler for Button1's Click event.
        AddHandler Button1.Click, AddressOf Button1_Click
    End Sub

    Private Button1 As Button = New Button()

    Sub Button1_Click(sender As Object, e As EventArgs)
        Console.WriteLine("Button1 was clicked!")
    End Sub

    Public Sub Disconnect()
        RemoveHandler Button1.Click, AddressOf Button1_Click
    End Sub
End Class
```

Given an event `E`, the statement calls the relevant `add_E` or `remove_E` method on the instance to add or remove the delegate as a handler for the event. Thus, the above code is equivalent to:

```
Imports System

Public Class Form1
    Public Sub New()
        Button1.add_Click(AddressOf Button1_Click)
    End Sub

    Private Button1 As Button = New Button()

    Sub Button1_Click(sender As Object, e As EventArgs)
        Console.WriteLine("Button1 was clicked!")
    End Sub

    Public Sub Disconnect()
        Button1.remove_Click(AddressOf Button1_Click)
    End Sub
End Class
```

```
AddHandlerStatement ::=
   AddHandler Expression , Expression StatementTerminator
RemoveHandlerStatement ::=
   RemoveHandler Expression , Expression StatementTerminator
```

See Also

10.5.1 RaiseEvent Statement | 10.5 Event Statements | AddHandler Statement (Visual Basic Language Reference) | RemoveHandler(Visual Basic Language Reference) | AddHandler and RemoveHandler (Visual Basic Language Concepts)

10.6 Assignment Statements

An assignment statement assigns the value of an expression to a variable. There are several types of assignment.

```
AssignmentStatement ::=
   RegularAssignmentStatement |
   CompoundAssignmentStatement |
   MidAssignmentStatement
```

See Also

10.6.1 Regular Assignment Statements | 10.6.2 Compound Assignment Statements | 10.6.3 Mid Assignment Statement | 10.1 Blocks | 10.1.1 Local Variables and Parameters | 10.2 Local Declaration Statements | 10.3 With Statement | 10.4 SyncLock Statement | 10.5 Event Statements | 10.7 Invocation Statements | 10.8 Conditional Statements | 10.9 Loop Statements | 10.10 Exception-Handling Statements | 10.11 Branch Statements | 10.12 Array-Handling Statements | Assignment Operators (Visual Basic Language Reference) | Assignment Statements (Visual Basic Language Concepts)

10.6.1 Regular Assignment Statements

A simple assignment statement stores the result of an expression in a variable. The expression on the left side of the assignment operator must be classified as a variable or a property access, while the expression on the right side of the assignment operator must be classified as a value. The type of the expression must be implicitly convertible to the type of the variable or property access.

If the variable being assigned into is an array element of a reference type, a run-time check will be performed to ensure that the expression is compatible with the array-element type. In the following example, the last assignment causes a **System.ArrayTypeMismatchException** to be thrown, because an instance of `ArrayList` cannot be stored in an element of a **String** array.

```
Dim sa(10) As String
Dim oa As Object() = sa
oa(0) = Nothing ' This is allowed.
oa(1) = "Hello" ' This is allowed.
oa(2) = New ArrayList() ' ArrayTypeMismatchException is thrown.
```

If the expression on the left side of the assignment operator is classified as a variable, then the assignment statement stores the value in the variable. If the expression is classified as a property access, then the assignment statement turns the property access into an invocation of the **Set** accessor of the property with the value substituted for the value parameter. For example:

```
Module Test
    Private PValue As Integer

    Public Property P As Integer
        Get
            Return PValue
        End Get

        Set (ByVal Value As Integer)
            PValue = Value
        End Set
    End Property

    Sub Main()
        ' The following two lines are equivalent.
        P = 10
        set_P(10)
    End Sub
End Module
```

If the associated instance expression of the property access is typed as a value type but not classified as a variable, a compile-time error occurs. For example:

```
Structure S
    Public F As Integer
End Structure

Class C
    Private PValue As S

    Public Property P As S
        Get
            Return P2Value
        End Get

        Set (ByVal Value As S)
            P2Value = Value
        End Set
    End Property
End Class

Module Test
```

```
    Sub Main()
        Dim ct As C = New C()
        Dim rt As Object = new C()

        ' Compile-time error: not classified as variable.
        ct.P.F = 10

        ' Run-time exception.
        rt.P.F = 10
    End Sub
End Module
```

> **Note** The semantics of the assignment depend on the type of the variable or property to which it is being assigned. If the variable to which it is being assigned is a value type, the assignment copies the value of the expression into the variable. If the variable to which it is being assigned is a reference type, the assignment copies the reference, not the value itself, into the variable. If the type of the variable is **Object**, the assignment semantics are determined by whether the value's type is a value type or a reference type at run time.

Because the equals character (**=**) is used both for assignment and for equality, there is an ambiguity between a simple assignment and an invocation statement in situations such as `x = y.ToString()`. In all such cases, the assignment statement takes precedence over the equality operator. This means that the example expression is interpreted as `x = (y.ToString())` rather than `(x = y).ToString()`.

```
RegularAssignmentStatement ::=
   Expression = Expression StatementTerminator
```

See Also

10.6.2 Compound Assignment Statements | 10.6.3 Mid Assignment Statement | 10.6 Assignment Statements | = Operator (Visual Basic Language Reference) | Assignment Statements (Visual Basic Language Concepts)

10.6.2 Compound Assignment Statements

A *compound assignment statement* takes the form *V OP= E* (where *OP* is a valid binary operator). The expression on the left side of the assignment operator must be classified as a variable or property access, while the expression on the right side of the assignment operator must be classified as a value. The compound assignment statement is equivalent to the statement *V = V OP E* with the difference that the variable on the left side of the compound assignment operator is only evaluated once. The following example demonstrates this difference:

```
Module Test
    Function GetIndex() As Integer
        Console.WriteLine("Getting index")
        Return 1
    End Function

    Sub Main()
        Dim a(2) As Integer

        Console.WriteLine("Simple assignment")
        a(GetIndex()) = a(GetIndex()) + 1

        Console.WriteLine("Compound assignment")
        a(GetIndex()) += 1
    End Sub
End Module
```

The expression `a(GetIndex())` is evaluated twice for simple assignment but only once for compound assignment, so the code prints:

```
Simple assignment
Getting index
Getting index
Compound assignment
Getting index
```

The rule for predefined operators is simply that *V OP= E* is permitted if both *V OP E* and *V = E* are permitted. Thus, in the following example, the reason for each error is that a corresponding simple assignment would also have been an error.

```
Option Strict On
Module Test
    Private b As Byte = 0
    Private ch As Char = ControlChars.NullChar
    Private i As Integer = 0
    Sub Main()
        b += 1 ' This is allowed.
        b += 1000 ' Error; b = 1000 is not permitted.
        b += i ' Error, b = I is not permitted.
        b += CByte(i) ' This is allowed.
        ch += 1 ' Error, ch = 1 is not permitted.
        ch += ChrW(1) ' This is allowed.
    End Sub
End Module
```

```
CompoundAssignmentStatement ::=
   Expression CompoundBinaryOperator = Expression StatementTerminator
CompoundBinaryOperator ::= ^ | * | / | \ | + | - | & | < < | > >
```

See Also

10.6.1 Regular Assignment Statements | 10.6.3 Mid Assignment Statement | 10.6 Assignment Statements | Assignment Operators (Visual Basic Language Reference)

10.6.3 Mid Assignment Statement

A **Mid** assignment statement assigns a string into another string. The left side of the assignment has the same syntax as a call to the function **Microsoft.VisualBasic.Strings.Mid**. The first argument is the target of the assignment and must be classified as a variable or a property access whose type is implicitly convertible to and from **String**. The second parameter is the 1-based start position that corresponds to where the assignment should begin in the target string and must be classified as a value whose type must be implicitly convertible to **Integer**. The optional third parameter is the number of characters from the right-side value to assign into the target string and must be classified as a value whose type is implicitly convertible to **Integer**. The right side is the source string and must be classified as a value whose type is implicitly convertible to **String**. The right side is truncated to the length parameter, if specified, and replaces the characters in the left-side string, starting at the start position.

The following example displays `ab123fg`:

```
Module Test
    Sub Main()
        Dim s1 As String = "abcdefg"
        Dim s2 As String = "1234567"

        Mid$(s1, 3, 3) = s2
        Console.WriteLine(s1)
    End Sub
End Module
```

Note **Mid** is not a reserved word.

```
MidAssignmentStatement ::=
   Mid [ $ ] ( Expression , Expression [ , Expression ] ) =
      Expression StatementTerminator
```

See Also

10.6.1 Regular Assignment Statements | 10.6.2 Compound Assignment Statements | 10.6 Assignment Statements | Mid Statement (Visual Basic Language Reference) | Assignment Statements (Visual Basic Language Concepts)

10.7 Invocation Statements

An invocation statement invokes a method preceded by the optional keyword **Call**. The invocation statement is processed in the same way as the function invocation expression. The invocation expression must be classified as a value or void. Any value resulting from the evaluation of the invocation expression is discarded.

```
InvocationStatement ::= [ Call ] InvocationExpression StatementTerminator
```

See Also

10.1 Blocks | 10.1.1 Local Variables and Parameters | 10.2 Local Declaration Statements | 10.3 With Statement | 10.4 SyncLock Statement | 10.5 Event Statements | 10.6 Assignment Statements | 10.8 Conditional Statements | 10.9 Loop Statements | 10.10 Exception-Handling Statements | 10.11 Branch Statements | 10.12 Array-Handling Statements | Call Statement (Visual Basic Language Reference)

10.8 Conditional Statements

Conditional statements allow conditional execution of statements based on expressions evaluated at run time.

```
ConditionalStatement ::= IfStatement | SelectStatement
```

See Also

10.8.1 If...Then...Else Statements | 10.8.2 Select...Case Statements | 10.1 Blocks | 10.1.1 Local Variables and Parameters | 10.2 Local Declaration Statements | 10.3 With Statement | 10.4 SyncLock Statement | 10.5 Event Statements | 10.6 Assignment Statements | 10.7 Invocation Statements | 10.9 Loop Statements | 10.10 Exception-Handling Statements | 10.11 Branch Statements | 10.12 Array-Handling Statements

10.8.1 If...Then...Else Statements

An **If...Then...Else** statement is the basic conditional statement. Each expression in an **If...Then...Else** statement must be classified as a value and be implicitly convertible to **Boolean**. If the expression in the **If** statement is **True**, the statements enclosed by the **If** block are executed. If the expression is **False**, each of the **ElseIf** expressions is evaluated. If one of the **ElseIf** expressions evaluates to **True**, the corresponding block is executed. If no expression evaluates to **True** and there is an **Else** block, the **Else** block is executed. Once a block finishes executing, execution passes to the end of the **If...Then...Else** statement.

The line version of the **If** statement has a single set of statements to be executed if the **If** expression is **True** and an optional set of statements to be executed if the expression is **False**. For example:

```
Module Test
    Sub Main()
        Dim a As Integer = 10
        Dim b As Integer = 20

        ' Block If statement.
        If a < b Then
            a = b
        Else
            b = a
        End If

        ' Line If statement
        If a < b Then a = b Else b = a
    End Sub
End Module
```

```
IfStatement ::= BlockIfStatement | LineIfThenStatement
BlockIfStatement ::=
   If Expression [ Then ] StatementTerminator
   [ Block ]
   [ ElseIfStatement+ ]
   [ ElseStatement ]
   End If StatementTerminator
ElseIfStatement ::=
   ElseIf Expression [ Then ] StatementTerminator
   [ Block ]
ElseStatement ::=
   Else StatementTerminator
   [ Block ]
LineIfThenStatement ::=
   If Expression Then [Statements ] [ Else Statements ]
      StatementTerminator
```

See Also

10.8.2 Select...Case Statements | 10.8 Conditional Statements | If...Then...Else Statements (Visual Basic Language Reference) | If...Then...Else Statements (Visual Basic Language Concepts)

10.8.2 Select...Case Statements

A **Select Case** statement executes statements based on the value of an expression. The expression must be classified as a value and its type must be a primitive type or **Object**.

When a **Select Case** statement is executed, the **Select** expression is evaluated first, and the **Case** statements are then evaluated in order of textual declaration. The first **Case** statement that evaluates to **True** has its block executed. If no **Case** statement evaluates to **True** and there is a **Case Else** statement, that block is executed. Once a block has finished executing, execution passes to the end of the **Select** statement.

Execution of a **Case** block is not permitted to "fall through" to the next switch section. This prevents a common class of bugs that occur in other languages when a **Case** terminating statement is accidentally omitted. The following example illustrates this behavior:

```
Module Test
    Sub Main()
        Dim x As Integer = 10

        Select Case x
            Case 5
                Console.WriteLine("x = 5")
            Case 10
                Console.WriteLine("x = 10")
            Case 20 - 10
                Console.WriteLine("x = 20 - 10")
            Case 30
                Console.WriteLine("x = 30")
        End Select
    End Sub
End Module
```

The code prints:

```
x = 10
```

Although `Case 10` and `Case 20 - 10` select for the same value, `Case 10` is executed because it precedes `Case 20 - 10` textually. When the next **Case** is reached, execution continues after the **Select** statement.

A **Case** clause may take two forms. One form is an optional **Is** keyword, a comparison operator, and an expression. The expression is converted to the type of the **Select** expression; if the expression is not implicitly convertible to the type of the **Select** expression, a compile-time error occurs. If the **Select** expression is *E*, the comparison operator is *Op*, and the **Case** expression is *E1*, the case is evaluated as *E OP E1*. The operator must be valid for the types of the two expressions; otherwise a compile-time error occurs.

The other form is an expression optionally followed by the keyword **To** and a second expression. Both expressions are converted to the type of the **Select** expression; if either expression is not implicitly convertible to the type of the **Select** expression, a compile-time error occurs. If the **Select** expression is *E,* the first **Case** expression is *E1*, and the second **Case** expression is *E2*, the **Case** is evaluated either as *E = E1* (if no *E2* is specified) or *(E >= E1) And (E <= E2)*. The operators must be valid for the types of the two expressions; otherwise a compile-time error occurs.

```
SelectStatement ::=
   Select [ Case ] Expression StatementTerminator
   [ CaseStatement+ ]
   [ CaseElseStatement ]
   End Select StatementTerminator
CaseStatement ::=
   Case CaseClauses StatementTerminator
   [ Block ]
CaseClauses ::=
   CaseClause |
   CaseClauses , CaseClause
CaseClause ::=
   [ Is ] ComparisonOperator Expression |
   Expression [ To Expression ]
```

```
ComparisonOperator ::= = | < > | < | > | = > | = <
CaseElseStatement ::=
   Case Else StatementTerminator
   [ Block ]
```

See Also

10.8.1 If...Then...Else Statements | 10.8 Conditional Statements | Select...Case Statements (Visual Basic Language Reference) | Select...Case Statements (Visual Basic Language Concepts)

10.9 Loop Statements

Loop statements allow repeated execution of statements.

```
LoopStatement ::=
   WhileStatement |
   DoLoopStatement |
   ForStatement |
   ForEachStatement
```

See Also

10.9.1 While...End While and Do...Loop Statements | 10.9.2 For...Next Statements |10.9.3 For Each...Next Statements | 10.1 Blocks | 10.1.1 Local Variables and Parameters | 10.2 Local Declaration Statements | 10.3 With Statement | 10.4 SyncLock Statement | 10.5 Event Statements | 10.6 Assignment Statements | 10.7 Invocation Statements | 10.8 Conditional Statements | 10.10 Exception-Handling Statements | 10.11 Branch Statements | 10.12 Array-Handling Statements | Loop Structures (Visual Basic Language Concepts)

10.9.1 While...End While and Do...Loop Statements

A **While** or **Do** loop statement loops based on a **Boolean** expression. A **While** loop statement loops as long as the **Boolean** expression evaluates to **True**; a **Do** loop statement may contain a more complex condition. In either case, the expressions must be classified as values and must be implicitly convertible to **Boolean**.

An expression may be placed after the **Do** keyword or after the **Loop** keyword, but not after both. It is also valid to specify no expression at all; in that case, the loop will never exit. If the expression is placed after **Do**, it will be evaluated before the loop block is executed on each iteration. If the expression is placed after **Loop**, it will be evaluated after the loop block has executed on each iteration. Placing the expression after **Loop** will therefore generate one more loop than placement after **Do**. The following example demonstrates this behavior:

```
Module Test
    Sub Main()
        Dim x As Integer

        x = 3
        Do While x = 1
            Console.WriteLine("First loop")
        Loop

        Do
            Console.WriteLine("Second loop")
        Loop While x = 1
    End Sub
End Module
```

The code produces the output:

```
Second Loop
Third Loop
Third Loop
```

In the case of the first loop, the condition is evaluated before the loop executes. In the case of the second loop, the condition is executed after the loop executes. The conditional expression must be prefixed with either a **While** keyword or an **Until** keyword. The former breaks the loop if the condition evaluates to **False**, the latter when the condition evaluates to **True**.

```
WhileStatement ::=
   While Expression StatementTerminator
   [ Block ]
   End While StatementTerminator
DoLoopStatement ::=
   Do [ WhileOrUntil Expression ] StatementTerminator
   [ Block ]
   Loop [ WhileOrUntil Expression ] StatementTerminator
WhileOrUntil ::= While | Until
```

See Also

10.9.2 For...Next Statements | 10.9.3 For Each...Next Statements | 10.9 Loop Statements | While...End While Statements (Visual Basic Language Reference) | While Statements (Visual Basic Language Concepts) | Do...Loop Statements (Visual Basic Language Reference) | Do...Loop Statements (Visual Basic Language Concepts)

10.9.2 For...Next Statements

A **For...Next** statement loops based on a set of bounds. A **For** statement specifies a loop control variable, a lower bound expression, an upper bound expression, and an optional step value expression.

The loop control variable is specified either through an identifier followed by an **As** clause or an expression. In the case of an identifier, the identifier defines a new local variable of the type specified in the **As** clause, scoped to the entire **For** loop. In the case of an expression, the expression must be classified as a variable.

The loop control variable of a **For** statement must be of a primitive numeric type (**Byte**, **Short**, **Integer**, **Long**, **Decimal**, **Single**, **Double**) or **Object**, and the bound and step expressions must be implicitly convertible to the type of the loop control. Enumerated types are treated as their underlying type, which allows use of enumerated types in **For** loops. The bounds and step expressions must be classified as values.

A loop control variable cannot be used by another enclosing **For...Next** statement. A **For** statement must be closed by a matching **Next** statement. A **Next** statement without a variable matches the innermost open **For** statement. A **Next** statement with one or more loop control variables will, from left to right, match the **For** loops that match each variable. If a variable matches a **For** loop that is not the most nested loop at that point, a compile-time error results.

At the beginning of the loop, the three expressions are evaluated in textual order and the lower bound expression is assigned to the control value. If the step value is omitted, it is implicitly the literal **1**. If the **For** loop variable is of type **Object**, at least one of the expressions at run time must be of a primitive numeric type, and all three expressions must be convertible to the widest numeric type among them. There is one exception to this rule. If the three expressions are typed as **String**, then all three expressions are converted to **Double**.

At the beginning of each loop, the control variable is compared to see if it is greater than the end point if the step expression is positive, or less than the end point if the step expression is negative. If it is, the **For** loop terminates; otherwise the loop block executes. At the **Next** statement, the step value is added to the control variable and execution returns to the top of the loop.

It is not valid to branch into a **For** loop from outside the loop.

```
ForStatement ::=
   For LoopControlVariable = Expression To Expression [ Step Expression ]
      StatementTerminator
   [ Block ]
   Next [ NextExpressionList ] StatementTerminator
LoopControlVariable ::=
   Identifier As TypeName |
   Expression
NextExpressionList ::=
   Expression |
   NextExpressionList , Expression
```

See Also

10.9.1 While...End While and Do...Loop Statements | 10.9.3 For Each...Next Statements | 10.9 Loop Statements | For...Next Statement (Visual Basic Language Reference) | For...Next Statements (Visual Basic Language Concepts)

10.9.3 For Each...Next Statements

A **For Each...Next** statement loops based on the elements in an expression. A **For Each** statement specifies a loop control variable and an enumerator expression.

The loop control variable is specified either through an identifier followed by an **As** clause or an expression. In the case of an identifier, the identifier defines a new local variable of the type specified in the **As** clause, scoped to the entire **For Each** loop. In the case of an expression, the expression must be classified as a variable. The enumerator expression must be classified as a value and its type must be a collection type. An implicit conversion must exist from the element type of the collection to the type of the loop control variable.

The loop control variable cannot be used by another enclosing **For Each** statement. A **For Each** statement must be closed by a matching **Next** statement. A **Next** statement without a loop control variable matches the innermost open **For Each**. A **Next** statement with one or more loop control variables will, from left to right, match the **For Each** loops that have the same loop control variable. If a variable matches a **For Each** loop that is not the most nested loop at that point, a compile-time error occurs.

A type `C` is said to be a *collection type* if it implements the interface **System.Collections.IEnumerable** or if all of the following are true:

- `C` contains an accessible instance method with the signature `GetEnumerator()` that returns a type `E`.
- `E` contains an accessible instance method with the signature `MoveNext()` and the return type **Boolean**.
- `E` contains an accessible instance property named `Current` that has a getter. The type of this property is said to be the *element type* of the collection type.

Following is an example of a class that can be enumerated:

```
Public Class IntegerCollection
    Private integers(10) As Integer

    Public Class IntegerCollectionEnumerator
        Private collection As IntegerCollection
        Private index As Integer = -1

        Friend Sub New(ByVal c As IntegerCollection)
            collection = c
        End Sub

        Public Function MoveNext() As Boolean
            index += 1

            Return index <= 10
        End Function

        Public ReadOnly Property Current As Integer
            Get
                If index < 0 OrElse index > 10 Then
                    Throw New System.InvalidOperationException()
                End If

                Return integers(index)
            End Get
        End Property
    End Class

    Public Sub New()
        Dim i As Integer

        For i = 0 To 10
            integers(i) = I
        Next i
    End Sub

    Public Function GetEnumerator() As IntegerCollectionEnumerator
        Return New IntegerCollectionEnumerator(Me)
    End Function
```

```
End Class
```

Before the loop begins, the enumerator expression is evaluated and cast to **System.Collections.IEnumerable** if it does not satisfy the requirements above. `GetEnumerator` is called on the resulting value and the return value of the function is stored in a temporary. Then at the beginning of each loop, `MoveNext` is called on the temporary. If it returns **False**, the loop terminates. If it returns **True**, the `Current` property is retrieved, coerced to the type of the iterator variable (regardless of whether the conversion is implicit or explicit), and assigned to the iterator variable; then the loop block executes. When the **Next** statement is reached, execution returns to the top of the loop. If a variable is specified after the **Next** keyword, it must be the same as the first variable after the **For Each**. For example, consider the following code:

```
Imports System

Module Test
    Sub Main()
        Dim i As Integer
        Dim c As IntegerCollection = New IntegerCollection()

        For Each i In c
            Console.WriteLine(i)
        Next i
    End Sub
End Module
```

It is equivalent to the following code:

```
Imports System

Module Test
    Sub Main()
        Dim i As Integer
        Dim c As IntegerCollection = New IntegerCollection()

        Dim e As IntegerCollectionEnumerator

        e = c.GetEnumerator()
        While e.MoveNext()
            i = e.Current

            Console.WriteLine(i)
        End While
    End Sub
End Module
```

If the type `E` of the enumerator implements **System.IDisposable**, then the enumerator is disposed upon exiting the loop by calling the `Dispose` method. This ensures that resources held by the enumerator are released. If the method containing the **For Each** statement does not use unstructured error handling, then the **For Each** statement is wrapped in a **Try** statement with the `Dispose` method called in the **Finally** to ensure cleanup.

> **Note** The **System.Array** type is a collection type, and since all array types derive from **System.Array**, any array type expression is permitted in a **For Each** statement. For single-dimensional arrays, the **For Each** statement enumerates the array elements in increasing index order, starting with index 0 and ending with index Length - 1. For multidimensional arrays, the indices of the rightmost dimension are increased first.

For example, the following code prints `1 2 3 4`:

```
Imports System

Module Test
    Sub Main()
        Dim x(,) As Integer = { { 1, 2 }, { 3, 4 } }
        Dim i As Integer

        For Each i In x
            Console.Write(i & " ")
```

```
        Next i
    End Sub
End Module
```

It is not valid to branch into a **For Each** statement block from outside the block.

```
ForEachStatement ::=
   For Each LoopControlVariable In Expression StatementTerminator
   [ Block ]
   Next [Expression ] StatementTerminator
```

See Also

10.9.1 While...End While and Do...Loop Statements | 10.9.2 For...Next Statements | 10.9 Loop Statements | For Each...Next Statements (Visual Basic Language Reference) | For Each...Next Statements (Visual Basic Language Concepts)

10.10 Exception-Handling Statements

Visual Basic .NET supports structured exception handling and unstructured exception handling. Only one style of exception handling may be used in a method, but the **Error** statement may be used in structured exception handling. If a method uses both styles of exception handling, a compile-time error results.

```
ErrorHandlingStatement ::=
   StructuredErrorStatement |
   UnstructuredErrorStatement
```

See Also

10.10.1 Structured Exception-Handling Statements | 10.10.2 Unstructured Exception-Handling Statements | 10.1 Blocks | 10.1.1 Local Variables and Parameters | 10.2 Local Declaration Statements | 10.3 With Statement | 10.4 SyncLock Statement | 10.5 Event Statements | 10.6 Assignment Statements | 10.7 Invocation Statements | 10.8 Conditional Statements | 10.9 Loop Statements | 10.11 Branch Statements | 10.12 Array-Handling Statements | Exception Handling (Visual Basic Language Concepts)

10.10.1 Structured Exception-Handling Statements

Structured exception handling is a method of handling errors by declaring explicit blocks within which certain exceptions will be handled. Structured exception handling is done through a **Try** statement.

For example:

```
Imports System

Module Test
    Sub ThrowException()
        Throw New Exception()
    End Sub

    Sub Main()
        Try
            ThrowException()
        Catch e As Exception
            Console.WriteLine("Caught exception!")
        Finally
            Console.WriteLine("Exiting try.")
        End Try
    End Sub
End Module
```

A **Try** statement is made up of three kinds of blocks: try blocks, catch blocks, and finally blocks. A *try block* is a statement block that contains the statements to be executed. A *catch block* is a statement block that handles an exception. A *finally block* is a statement block that contains statements to be run when the **Try** statement is exited, regardless of whether an exception has occurred and been handled. A **Try** statement, which can only contain one try block and one finally block, must contain at least one catch block or finally block. It is invalid to explicitly transfer execution into a try block except from within a catch block in the same statement.

```
StructuredErrorStatement ::=
   ThrowStatement |
   TryStatement
TryStatement ::=
   Try StatementTerminator
   [ Block ]
   [ CatchStatement+ ]
   [ FinallyStatement ]
   End Try StatementTerminator
```

See Also

10.10.1.1 Finally Blocks | 10.10.1.2 Catch Blocks | 10.10.2 Unstructured Exception-Handling Statements | 10.10 Exception-Handling Statements | Throw Statement (Visual Basic Language Reference) | Try...Catch Statements (Visual Basic Language Concepts) | Structured Exception Handling (Visual Basic Language Concepts)

10.10.1.1 Finally Blocks

A **Finally** block is always executed when execution leaves any part of the **Try** statement. No explicit action is required to execute the **Finally** block; when execution leaves the **Try** statement, the system will automatically execute the **Finally** block and then transfer execution to its intended destination. The **Finally** block is executed regardless of how execution leaves the **Try** statement: through the end of the **Try** block, through the end of a **Catch** block, through an **Exit Try** statement, through a **GoTo** statement, or by not handling a thrown exception.

It is invalid to explicitly transfer execution into a **Finally** block; it is also invalid to transfer execution out of a **Finally** block except through an exception.

```
FinallyStatement ::=
   Finally StatementTerminator
   [ Block ]
```

See Also

10.10.1.2 Catch Blocks | 10.10.1 Structured Exception-Handling Statements |
Try...Catch...Finally Statement (Visual Basic Language Reference)

10.10.1.2 Catch Blocks

If an exception occurs while processing the **Try** block, each **Catch** statement is examined in textual order to determine if it handles the exception. The identifier specified in a **Catch** clause represents the exception that has been thrown. If the identifier contains an **As** clause, then the identifier is considered to be declared within the **Catch** block's local declaration space. Otherwise, the identifier must be a local variable (not a static variable) that was defined in a containing block.

A **Catch** clause with no identifier will catch all exceptions derived from **System.Exception**. A **Catch** clause with an identifier will only catch exceptions whose types are the same as or derived from the type of the identifier. The type must be **System.Exception** or a type derived from **System.Exception**. A **Catch** clause with a **When** clause will only catch exceptions when the expression evaluates to **True**; the type of the expression must be implicitly convertible to **Boolean**. A **When** clause is only applied after checking the type of the exception, and the expression may refer to the identifier representing the exception, as this example demonstrates:

```
Imports System

Module Test
    Sub Main()
        Dim i As Integer = 5

        Try
            Throw New ArgumentException()
        Catch e As OverflowException When i = 5
            Console.WriteLine("First handler")
        Catch e As ArgumentException When i = 4
            Console.WriteLine("Second handler")
        Catch When i = 5
            Console.WriteLine("Third handler")
        End Try

    End Sub
End Module
```

This example prints:

```
Third handler
```

If a **Catch** clause handles the exception, execution transfers to the **Catch** block. At the end of the **Catch** block, execution transfers to the first statement following the **Try** statement. The **Try** statement will not handle any exceptions thrown in a **Catch** block. If no **Catch** clause handles the exception, execution transfers to a location determined by the system.

It is invalid to explicitly transfer execution into a **Catch** block.

A catch block can branch into a try block in the same **Try** statement, but only if there is no intervening nested **Try** statements. For example, the following produces a compile-time error:

```
Module Test
    Sub Main()
        Try
            Throw New Exception()
out:
        Catch e As Exception
            Try
                Throw New Exception()
            Catch eInner As Exception
                ' Error: Not directly contained.
                Goto out
            End Try
        End Try
    End Sub
End Module
```

```
CatchStatement ::=
```

```
    Catch [ Identifier As TypeName ] [ When BooleanExpression ]
       StatementTerminator
    [ Block ]
```

See Also

10.10.1.1 Finally Blocks | 10.10.1 Structured Exception-Handling Statements |
Try...Catch Statements (Visual Basic Language Concepts)

10.10.1.3 Throw Statement

The **Throw** statement raises an exception, which is represented by an instance of a type derived from **System.Exception**. If the expression is not classified as a value or is not a type derived from **System.Exception**, then a compile-time error occurs. If the expression evaluates to a null reference at run time, then a **System.NullReferenceException** exception is raised instead.

A **Throw** statement may omit the expression within a catch block of a **Try** statement. In that case, the statement rethrows the exception currently being handled within the catch block.

```
ThrowStatement ::= Throw [ Expression ] StatementTerminator
```

See Also

10.10.1.1 Finally Blocks | 10.10.1 Structured Exception-Handling Statements |
Try...Catch Statements (Visual Basic Language Concepts)

10.10.2 Unstructured Exception-Handling Statements

Unstructured exception handling is a method of handling errors by indicating statements to branch to when an exception occurs. Unstructured exception handling is implemented using three statements: the **Error** statement, the **On Error** statement, and the **Resume** statement. For example:

```
Imports System

Module Test
    Sub ThrowException()
        Error 5
    End Sub

    Sub Main()
        On Error Goto GotException

        ThrowException()
        Exit Sub

GotException:
        Console.WriteLine("Caught exception!")
        Resume Next
    End Sub
End Module
```

When a method uses unstructured exception handling, a single structured exception handler is established for the entire method that catches all exceptions. The method then keeps track of the most recent exception-handler location and the most recent exception that has been thrown. At entry to the method, the exception-handler location and the exception are both set to **Nothing**. When an exception is thrown in a method that uses unstructured exception handling, a reference to the exception object is stored in the object returned by the function **Microsoft.VisualBasic.Information.Err**.

```
UnstructuredErrorStatement ::=
   ErrorStatement |
   OnErrorStatement |
   ResumeStatement
```

See Also

10.10.2.1 Error Statement | 10.10.2.2 On Error Statement | 10.10.2.3 Resume Statement |
10.10.1 Structured Exception-Handling Statements | 10.10 Exception-Handling Statements |
Unstructured Exception Handling (Visual Basic Language Concepts)

10.10.2.1 Error Statement

An **Error** statement throws a **System.Exception** exception containing a Visual Basic 6 exception number. The expression must be classified as a value and its type must be implicitly convertible to **Integer**.

```
ErrorStatement ::= Error Expression StatementTerminator
```

See Also

10.10.2.2 On Error Statement | 10.10.2.3 Resume Statement | 10.10.2 Unstructured Exception-Handling Statements | Error Statement (Visual Basic Language Reference) | Unstructured Exception Handling (Visual Basic Language Concepts)

10.10.2.2 On Error Statement

An **On Error** statement modifies the most recent exception-handling state. It may be used in one of four ways:

- **On Error GoTo -1** resets the most recent exception to **Nothing**.
- **On Error GoTo 0** resets the most recent exception-handler location to **Nothing**.
- **On Error GoTo** *LabelName* establishes the label as the most recent exception-handler location.
- **On Error Resume Next**, establishes the **Resume Next** behavior as the most recent exception-handler location.

```
OnErrorStatement ::= On Error ErrorClause StatementTerminator
ErrorClause ::=
   GoTo - 1 |
   GoTo 0 |
   GotoStatement |
   Resume Next
```

See Also

10.10.2.1 Error Statement | 10.10.2.3 Resume Statement | 10.10.2 Unstructured Exception-Handling Statements |
On Error Statement (Visual Basic Language Reference) | Unstructured Exception Handling (Visual Basic Language Concepts)

10.10.2.3 Resume Statement

A **Resume** statement returns execution to the statement that caused the most recent exception. If the **Next** modifier is specified, execution returns to the statement that would have been executed after the statement that caused the most recent exception. If a label name is specified, execution returns to the label.

Because the **SyncLock** statement contains an implicit structured error-handling block, **Resume** and **Resume Next** have special behaviors for exceptions that occur in **SyncLock** statements. **Resume** returns execution to the beginning of the **SyncLock** statement, while **Resume Next** returns execution to the next statement following the **SyncLock** statement. For example, consider the following code:

```
Imports System

Class LockClass
End Class

Module Test
    Sub Main()
        Dim FirstTime As Boolean = False
        Dim Lock As LockClass = New LockClass()

        On Error Goto Handler

        SyncLock Lock
            Console.WriteLine("Before exception")
            FirstTime = False
            Throw New Exception()
            Console.WriteLine("After exception")
        End SyncLock

        Console.WriteLine("After SyncLock")
        Exit Sub

Handler:
        If FirstTime Then
            Resume
        Else
            Resume Next
        End If
    End Sub
End Module
```

It prints the following result.

```
Before exception
Before exception
After SyncLock
```

The first time through the **SyncLock** statement, **Resume** returns execution to the beginning of the **SyncLock** statement. The second time through the **SyncLock** statement, **Resume Next** returns execution to the end of the **SyncLock** statement. **Resume** and **Resume Next** are not allowed within a **SyncLock** statement.

In all cases, when a **Resume** statement is executed, the most recent exception is set to **Nothing**. If a **Resume** statement is executed with no most recent exception, the statement raises a **System.Exception** exception containing the Visual Basic error number `20` (Resume without error).

```
ResumeStatement ::= Resume [ ResumeClause ] StatementTerminator
ResumeClause ::= Next | LabelName
```

See Also

10.10.2.1 Error Statement | 10.10.2.2 On Error Statement | 10.10.2 Unstructured Exception-Handling Statements |

Resume Statement (Visual Basic Language Reference) | Unstructured Exception Handling (Visual Basic Language Concepts)

10.11 Branch Statements

Branch statements modify the flow of execution in a method. There are five branch statements:

- A **GoTo** statement causes execution to transfer to the specified label in the method.
- An **Exit** statement transfers execution to the next statement after the end of the immediately containing block statement of the specified kind. If the block is the method block, execution is transferred back to the expression that invoked the method. If the **Exit** statement is not contained within the kind of block specified in the statement, a compile-time error occurs.
- A **Stop** statement causes a debugger exception to occur.
- An **End** statement terminates the program. Finalizers are run before shutdown, but the finally blocks of any currently executing **Try** statements are not executed. This statement may not be used in programs that are not executable (for example, DLLs).
- A **Return** statement returns execution to the expression that invoked the method. If the method is a subroutine, the statement is equivalent to an **Exit Sub** statement and no expression may be supplied. If the method is a function, an expression must be supplied that is classified as a value and whose type is implicitly convertible to the return type of the function. This form is equivalent to assigning to the function return local and then executing an **Exit Function** statement.

```
BranchStatement ::=
   GotoStatement |
   ExitStatement |
   StopStatement |
   EndStatement |
   ReturnStatement
GotoStatement ::= GoTo LabelName StatementTerminator
ExitStatement ::= Exit ExitKind StatementTerminator
ExitKind ::= Do | For | While | Select | Sub | Function | Property | Try
StopStatement ::= Stop StatementTerminator
EndStatement ::= End StatementTerminator
ReturnStatement ::= Return [ Expression ]
```

See Also

10.1 Blocks | 10.1.1 Local Variables and Parameters | 10.2 Local Declaration Statements | 10.3 With Statement | 10.4 SyncLock Statement | 10.5 Event Statements | 10.6 Assignment Statements | 10.7 Invocation Statements | 10.8 Conditional Statements | 10.9 Loop Statements | 10.10 Exception-Handling Statements | 10.12 Array-Handling Statements | Control Flow Summary (Visual Basic Language Reference)

10.12 Array-Handling Statements

Two statements simplify working with arrays: **ReDim** statements and **Erase** statements.

```
ArrayHandlingStatement ::=
   RedimStatement |
   EraseStatement
```

See Also

10.12.1 ReDim Statement | 10.12.2 Erase Statement | 10.1 Blocks | 10.1.1 Local Variables and Parameters | 10.2 Local Declaration Statements | 10.3 With Statement | 10.4 SyncLock Statement | 10.5 Event Statements | 10.6 Assignment Statements | 10.7 Invocation Statements | 10.8 Conditional Statements | 10.9 Loop Statements | 10.10 Exception-Handling Statements | 10.11 Branch Statements | ReDim Statement (Visual Basic Language Reference) | Erase Statement (Visual Basic Language Reference)

10.12.1 ReDim Statement

A **ReDim** statement instantiates new arrays. Each clause in the statement must be classified as a variable or a property access whose type is an array type or **Object**, and be followed by a list of array bounds. The number of the bounds must be consistent with the type of the variable; any number of bounds is allowed for **Object**. At run time, an array is instantiated for each expression from left to right with the specified bounds and then assigned to the variable or property. If the variable type is **Object**, the number of dimensions is the number of dimensions specified, and the array element type is **Object**. If the given number of dimensions is incompatible with the variable or property at run time, a **System.InvalidCastException** will be thrown. For example:

```
Module Test
    Sub Main()
        Dim o As Object
        Dim b() As Byte
        Dim i(,) As Integer

        ' The next two statements are equivalent.
        ReDim o(10,30)
        o = New Object(10, 30) {}

        ' The next two statements are equivalent.
        ReDim b(10)
        b = New Byte(10) {}

        ' The following statement throws an InvalidCastException.
        ReDim i(10, 30, 40)
    End Sub
End Module
```

If the **Preserve** keyword is specified, then the expressions must also be classifiable as a value, and the new size for each dimension except for the rightmost one must be the same as the size of the existing array. The values in the existing array are copied into the new array: if the new array is smaller, the existing values are discarded; if the new array is bigger, the extra elements will be initialized to the default value of the element type of the array. For example, consider the following code:

```
Imports System

Module Test
    Sub Main()
        Dim x(5, 5) As Integer

        x(3, 3) = 3

        ReDim Preserve x(3, 6)
        Console.WriteLine(x(3, 3) & ", " & x(3, 6))
    End Sub
End Module
```

It prints the following result:

```
3, 0
```

If the existing array reference is null at run time, no error is given. Other than the rightmost dimension, if the size of a dimension changes, a **System.ArrayTypeMismatchException** will be thrown.

```
RedimStatement ::= ReDim [ Preserve ] RedimClauses StatementTerminator
RedimClauses ::=
   RedimClause |
   RedimClauses , RedimClause
RedimClause ::= Expression ArraySizeInitializationModifier
```

See Also

10.12.2 Erase Statement

An **Erase** statement sets each of the array variables or properties specified in the statement to **Nothing**. Each expression in the statement must be classified as a variable or property access whose type is an array type or **Object**. For example:

```
Module Test
    Sub Main()
        Dim x() As Integer = New Integer(5) {}

        ' The following two statements are equivalent.
        Erase x
        x = Nothing
    End Sub
End Module
```

```
EraseStatement ::= Erase EraseExpressions StatementTerminator
EraseExpressions ::=
   Expression |
   EraseExpressions , Expression
```

See Also

10.12.1 ReDim Statement | 10.12 Array-Handling Statements | Erase Statement (Visual Basic Language Reference)

11. Expressions

An expression is a sequence of operators and operands that specifies a computation of a value, or that designates a variable or constant. This chapter defines the syntax, order of evaluation of operands and operators, and meaning of expressions.

```
Expression ::=
   SimpleExpression |
   TypeExpression |
   MemberAccessExpression |
   DictionaryAccessExpression |
   IndexExpression |
   NewExpression |
   CastExpression |
   OperatorExpression
```

See Also

11.4 Simple Expressions | 11.8 Invocation Expressions | 11.6 Member Access Expressions | 11.9 Index Expressions | 11.10 New Expressions | 11.11 Cast Expressions | 11.12 Operator Expressions | 11.3 Late-Binding Expressions | 9.2.5.2 Reference Parameters | Expressions (Visual Basic Language Concepts)

11.1 Expression Classifications

Every expression is classified as one of the following:

- A value. Every value has an associated type.
- A variable. Every variable has an associated type, namely the declared type of the variable.
- A namespace. An expression with this classification can only appear as the left side of a member access. In any other context, an expression classified as a namespace causes a compile-time error.
- A type. An expression with this classification can only appear as the left side of a member access. In any other context, an expression classified as a type causes a compile-time error.
- A method group, which is a set of methods overloaded on the same name. A method group may have an associated instance expression.
- A method pointer, which represents the location of a method. A method pointer may have an associated instance expression.
- A property group, which is a set of properties overloaded on the same name. A property group may have an associated instance expression.
- A property access. Every property access has an associated type, namely the type of the property. A property access may have an associated instance expression.
- An event access. Every event access has an associated type, namely the type of the event. An event access may have an associated instance expression. An event access may appear as the first argument of the **RaiseEvent**, **AddHandler**, and **RemoveHandler** statements. In any other context, an expression classified as an event access causes a compile-time error.
- Void. This occurs when the expression is an invocation of a subroutine. An expression classified as void is only valid in the context of an invocation statement.

The final result of an expression is never a namespace, type, method group, or property group. Rather, as noted above, these categories of expressions are intermediate constructs that are only permitted in certain contexts.

See Also

4.8 Variables | 10.5 Event Statements | 11.1 Expression Classifications |
Constants and Enumerations (Visual Basic Language Reference) | Constants and Enumerations (Visual Basic Language Concepts)

11.1.1 Expression Reclassification

Normally, when an expression is used in a context that requires a classification different from that of the expression, a compile-time error occurs — for example, attempting to assign a value to a literal. However, in many cases it is possible to change an expression's classification through the process of *reclassification*. The following types of expressions can be reclassified:

- A variable can be reclassified as a value. The value stored in the variable is fetched.
- A method group can be reclassified as a value. The method group expression is interpreted as an invocation expression with empty parentheses (that is, `f` is interpreted as `f()`). This reclassification may actually result in the expression being reclassified as void.
- A method pointer can be reclassified as a value. This reclassification can only occur in assignment statements or as a part of interpreting a parameter list, where the target type is known. The method pointer expression is interpreted as the argument to a delegate instantiation expression of the appropriate type. For example:

```
Delegate Sub D(ByVal i As Integer)

Module Test
    Sub F(ByVal i As Integer)

    Sub Main()
        Dim del As D

        ' The next two lines are equivalent.
        del = AddressOf F
        del = New D(AddressOf F)
    End Sub
End Module
```

- A property group can be reclassified as a property access. The property group expression is interpreted as an index expression with empty parentheses (that is, `f` is interpreted as `f()`).
- A property access can be reclassified as a value. The property access expression is interpreted as an invocation expression of the **Get** accessor of the property. If the property has no getter, then a compile-time error occurs.

A namespace expression, type expression, event access expression, or void expression cannot be reclassified. Multiple reclassifications can be done at the same time. For example:

```
Module Test
    Sub F(ByVal i As Integer)
    End Sub

    ReadOnly Property P() As Integer
        Get
        End Get
    End Sub

    Sub Main()
        F(P)
    End Sub
End Module
```

In this case, the property group expression `P` is first reclassified from a property group to a property access and then reclassified from a property access to a value. The fewest number of reclassifications are performed to reach a valid classification in the context.

See Also

4.8 Variables | 10.5 Event Statements | 11.1 Expression Classifications |
Constants and Enumerations (Visual Basic Language Reference) | Constants and Enumerations (Visual Basic Language Concepts)

11.2 Constant Expressions

A *constant expression* is an expression whose value can be fully evaluated at compile time. The type of a constant expression can be **Byte**, **Short**, **Integer**, **Long**, **Char**, **Single**, **Double**, **Decimal**, **Boolean**, **String**, or any enumeration type. The following constructs are permitted in constant expressions:

- Literals (including **Nothing**).
- References to constant type members or constant locals.
- References to members of enumeration types.
- Parenthesized subexpressions.
- Coercion expressions, provided the target type is one of the types listed above. Coercions to and from **String** are an exception to this rule and not allowed because **String** conversions are always done in the current culture of the execution environment at run time.
- The **+**, **–** and **Not** unary operators.
- The **+**, **–**, *****, **^**, **Mod**, **/**, ****, **<<**, **>>**, **&**, **And**, **Or**, **Xor**, **AndAlso**, **OrElse**, **=**, **<**, **>**, **<>**, **<=**, and **=>** binary operators, provided each operand is of a type listed above.
- The following run-time functions:
  - **Microsoft.VisualBasic.Strings.ChrW**
  - **Microsoft.VisualBasic.Strings.Chr**, if the constant value is between 0 and 128
  - **Microsoft.VisualBasic.Strings.AscW**, if the constant string is not empty
  - **Microsoft.VisualBasic.Strings.Asc**, if the constant string is not empty

Constant expressions of an integral type (**Long**, **Integer**, **Short**, **Byte**) can be implicitly converted to a narrower integral type, and constant expressions of type **Double** can be implicitly converted to **Single**, provided the value of the constant expression is within the range of the destination type. These narrowing conversions are allowed regardless of whether permissive or strict semantics are being used.

```
ConstantExpression ::= Expression
```

See Also

4.8 Variables | 10.5 Event Statements | 11.1 Expression Classifications |
Constants and Enumerations (Visual Basic Language Reference) | Constants and Enumerations (Visual Basic Language Concepts)

11.3 Late-Binding Expressions

When the target of a member access expression or index expression is of type **Object**, the processing of the expression may be deferred until run time. Deferring processing this way is called *late binding*. Late binding allows **Object** variables to be used in a *typeless* way, where all resolution of members is based on the actual run-time type of the value in the variable. If strict semantics are specified by the compilation environment or by **Option Strict**, late binding causes a compile-time error. Non-public members are ignored when doing late-binding, including for the purposes of overload resolution. Note that, unlike the early-bound case, invoking or accessing a **Shared** member late bound will cause the invocation target to be evaluated at run time.

> **Note** If the expression is an invocation expression for a member defined on **System.Object**, late binding will not take place.

See Also

11.8 Invocation Expressions | 11.4 Simple Expressions | 11.8 Invocation Expressions | 11.6 Member Access Expressions | 11.9 Index Expressions | 11.10 New Expressions | 11.11 Cast Expressions | 9.2.5.2 Reference Parameters | Typeless Programming (Visual Basic Language Concepts)

11.4 Simple Expressions

Simple expressions are literals, parenthesized expressions, instance expressions, or simple name expressions.

```
SimpleExpression ::=
   LiteralExpression |
   ParenthesizedExpression |
   InstanceExpression |
   SimpleNameExpression |
   AddressOfExpression
```

See Also

11.4.1 Literal Expressions | 11.4.2 Parenthesized Expressions | 11.4.3 Instance Expressions | 11.5.1 GetType Expressions | 4.8 Variables | 11.5.2 TypeOf...Is Expressions | 11.5.3 Is Expressions | 11.3 Late-Binding Expressions | 9.2.5.2 Reference Parameters | Expressions (Visual Basic Language Concepts)

11.4.1 Literal Expressions

Literal expressions evaluate to the value represented by the literal. A literal expression is classified as a value.

```
LiteralExpression ::= Literal
```

See Also

11.4.2 Parenthesized Expressions | 11.4.3 Instance Expressions | 11.5.1 GetType Expressions | 11.5.2 TypeOf...Is Expressions | 11.5.3 Is Expressions | 11. Expressions | 2.4 Literals | Expressions (Visual Basic Language Concepts)

11.4.2 Parenthesized Expressions

A parenthesized expression consists of an expression enclosed in parentheses. A parenthesized expression is classified as a value, and the enclosed expression must be classified as a value. A parenthesized expression evaluates to the value of the expression within the parentheses.

```
ParenthesizedExpression ::= ( Expression )
```

See Also

11.4.1 Literal Expressions | 11.4.3 Instance Expressions | 11.5.1 GetType Expressions | 4.8 Variables | 11.5.2 TypeOf...Is Expressions | 11.5.3 Is Expressions | 11. Expressions | Expressions (Visual Basic Language Concepts)

11.4.3 Instance Expressions

An *instance expression* is the keyword **Me**, **MyClass**, or **MyBase**. An instance expression, which may only be used within the body of a non-shared method, constructor, or property accessor, is classified as a value. Instance expressions cannot be used in the arguments to constructor invocations. The three different instance expression types are:

- The keyword **Me** represents the instance of the type containing the method or property accessor being executed.
- The keyword **MyClass** is equivalent to **Me**, but all method invocations on it are treated as if the method being invoked is non-overridable. Thus, the method called will not be affected by the run-time type of the value on which the method is being called. **MyClass** can only be used as the target expression of a member access.
- The keyword **MyBase** represents the instance of the type containing the method or property accessor being executed cast to its direct base type. All method invocations on it are treated as if the method being invoked is non-overridable. **MyBase** can only be used as the target expression of a member access. A member access using **MyBase** is called a *base access*.

The following example demonstrates how instance expressions can be used:

```
Imports System

Class Base
    Public Overridable Sub F()
        Console.WriteLine("Base.F")
    End Sub
End Class

Class Derived
    Inherits Base

    Public Overrides Sub F()
        Console.WriteLine("Derived.F")
    End Sub

    Public Sub G()
        MyClass.F()
    End Sub
End Class

Class MoreDerived
    Inherits Derived

    Public Overrides Sub F()
        Console.WriteLine("MoreDerived.F")
    End Sub

    Public Sub H()
        MyBase.F()
    End Sub
End Class

Module Test
    Sub Main()
        Dim x As MoreDerived = new MoreDerived()

        x.F()
        x.G()
        x.H()
    End Sub

End Module
```

This code prints out:

```
MoreDerived.F
Derived.F
Derived.F
```

```
InstanceExpression ::= Me | MyClass | MyBase
```

See Also

11.4.1 Literal Expressions | 11.4.2 Parenthesized Expressions | 11.5.1 GetType Expressions | 4.8 Variables | 11.5.2 TypeOf...Is Expressions | 11.5.3 Is Expressions | 11. Expressions | Me (Visual Basic Language Reference) | Expressions (Visual Basic Language Concepts)

11.4.4 Simple Name Expressions

A *simple name expression* consists of a single identifier. The name is resolved and classified as follows:

- Starting with the immediately enclosing block and continuing with each enclosing outer block (if any), if the identifier matches the name of a local variable, static variable, constant local, or parameter, then the identifier refers to the matching entity. The expression is classified as a variable if it is a local variable, static variable, or parameter. The expression is classified as a value if it is a constant local with the following exception. If the local variable matched is the implicit function or **Get** accessor return local variable, and the expression is part of an invocation expression, invocation statement, or an **AddressOf** expression, then no match occurs and resolution continues.
- For each nested type containing the expression, starting from the innermost and going to the outermost, if a lookup of the identifier in the type produces a match with an accessible member:
  - If the type is the immediately enclosing type and the lookup identifies a non-shared type member, then the result is the same as a member access of the form `Me.E`, where `E` is the identifier.
  - Otherwise, the result is exactly the same as a member access of the form `T.E`, where `T` is the type containing the matching member and `E` is the identifier. In this case, it is an error for the identifier to refer to a non-shared member.
- For each nested namespace, starting from the innermost and going to the outermost namespace, do the following:
  - If the namespace contains an accessible namespace member with the given name, then the identifier refers to that member and, depending on the member, is classified as a namespace or a type.
  - Otherwise, if the namespace contains one or more accessible standard modules, and a member name lookup of the identifier produces an accessible match in exactly one standard module, then the result is exactly the same as a member access of the form `M.E`, where `M` is the standard module containing the matching member and `E` is the identifier. If the identifier matches accessible type members in more than one standard module, a compile-time error occurs.
- If the source file has one or more import aliases, and the identifier matches the name of one of them, then the identifier refers to that namespace or type.
- If the source file containing the name reference has one or more imports:
  - If the identifier matches the name of an accessible namespace or type member in exactly one import, then the identifier refers to that namespace or type. If the identifier matches the name of an accessible namespace or type members in more than one import, a compile-time error occurs.
  - Otherwise, if the imports contain one or more accessible standard modules, and a member name lookup of the identifier produces an accessible match in exactly one standard module, then the result is exactly the same as a member access of the form `M.E`, where `M` is the standard module containing the matching member and `E` is the identifier. If the identifier matches accessible type members in more than one standard module, a compile-time error occurs.
- If the compilation environment defines one or more import aliases, and the identifier matches the name of one of them, then the identifier refers to that namespace or type.
- If the compilation environment defines one or more imports:
  - If the identifier matches the name of an accessible namespace or type member in exactly one import, then the identifier refers to that namespace or type. If the identifier matches the name of accessible namespace or type members in more than one import, a compile-time error occurs.
  - Otherwise, if the imports contain one or more accessible standard modules, and a member name lookup of the identifier produces an accessible match in exactly one standard module, then the result is exactly the same as a member access of the form `M.E`, where `M` is the standard module containing the matching member and `E` is the identifier. If the identifier matches accessible type members in more than one standard module, a compile-time error occurs.
- Otherwise, the name given by the identifier is undefined and a compile-time error occurs.

```
SimpleNameExpression ::= Identifier
```

See Also

11.8 Invocation Expressions | AddressOf Operator (Visual Basic Language Reference) |
Expressions (Visual Basic Language Concepts)

11.4.5 AddressOf Expressions

An **AddressOf** expression is used to produce a method pointer. The expression consists of the **AddressOf** keyword and an expression that must be classified as a method group. The method group cannot be late-bound and it cannot refer to constructors.

The result is classified as a method pointer and the associated instance expression (if any) is the same as the associated instance expression of the method group.

```
AddressOfExpression ::= AddressOf Expression
```

See Also

6.2.3 Option Compare Statement | 11.8 Invocation Expressions | AddressOf Operator (Visual Basic Language Reference) | Delegates and the AddressOf Operator (Visual Basic Language Concepts)

11.5 Type Expressions

A *type expression* is a **GetType** expression, a **TypeOf...Is** expression, or an **Is** expression.

```
TypeExpression ::=
   GetTypeExpression |
   TypeOfIsExpression |
   IsExpression
```

See Also

11.4.1 Literal Expressions | 11.4.2 Parenthesized Expressions | 11.4.3 Instance Expressions | 11.5.1 GetType Expressions | 4.8 Variables | 11.5.2 TypeOf...Is Expressions | 11.5.3 Is Expressions | 11.3 Late-Binding Expressions | 9.2.5.2 Reference Parameters | Expressions (Visual Basic Language Concepts)

11.5.1 GetType Expressions

A **GetType** expression consists of the keyword **GetType** and the name of a type. A type expression is classified as a value, and the value of the expression is the reflection (**System.Type**) class that represents that type. The type name is allowed to be **System.Void**, the only place in the language where this type name may be referenced.

The following example demonstrates the GetType expression:

```
Imports System

Module Test
    Sub Main()
        Dim t As Type() = { GetType(Integer), GetType(System.Int32), _
            GetType(String), GetType(Double()) }
        Dim i As Integer

        For i = 0 To t.Length - 1
            Console.WriteLine(t(i).Name)
        Next i
    End Sub
End Module
```

The resulting output is:

```
Int32
Int32
String
Double()
```

```
GetTypeExpression ::= GetType ( TypeName )
```

See Also

11.4.1 Literal Expressions | 11.4.2 Parenthesized Expressions | 11.4.3 Instance Expressions | 4.8 Variables | 11.5.2 TypeOf...Is Expressions | 11.5.3 Is Expressions | 11. Expressions | GetType Operator (Visual Basic Language Reference) | Expressions (Visual Basic Language Concepts)

11.5.2 TypeOf...Is Expressions

A **TypeOf...Is** expression is used to check whether the run-time type of a value is compatible with a given type. The first operand must be classified as a value and must be of a reference type. The second operand must be a type name. The result of the expression is classified as a value and is a **Boolean** value. The expression evaluates to **True** if the run-time type of the operand is derived from or implements the type, **False** otherwise.

```
TypeOfIsExpression ::= TypeOf Expression Is TypeName
```

See Also

11.4.1 Literal Expressions | 11.4.2 Parenthesized Expressions | 11.4.3 Instance Expressions | 11.5.1 GetType Expressions | 4.8 Variables | 11.5.3 Is Expressions | 11. Expressions | TypeOf (Visual Basic Language Reference)

11.5.3 Is Expressions

An **Is** expression is used to do a reference equality comparison. Each expression must be classified as a value and the type of each expression must be a reference type. The result is classified as a value and is typed as **Boolean**. The expression evaluates to **True** if both values refer to the same instance or **False** otherwise.

```
IsExpression ::= Expression Is Expression
```

See Also

11.4.1 Literal Expressions | 11.4.2 Parenthesized Expressions | 11.4.3 Instance Expressions | 11.5.1 GetType Expressions | 4.8 Variables | 11.5.2 TypeOf...Is Expressions | 11. Expressions | Is Operator (Visual Basic Language Reference) | Expressions (Visual Basic Language Concepts)

11.6 Member Access Expressions

A member access expression is used to access a member of an entity. A member access of the form `E.I`, where `E` is an expression, a built-in type, or omitted and `I` is an identifier, is evaluated and classified as follows:

- If `E` is omitted, then the expression from the immediately containing **With** statement is substituted for `E` and the member access is performed. If there is no containing **With** statement, a compile-time error occurs.
- If `E` is classified as a namespace and `I` is the name of an accessible member of that namespace, then the result is that member. The result is classified as a namespace or a type depending on the member.
- If `E` is a built-in type or an expression classified as a type, and `I` is the name of an accessible member of `E`, then `E.I` is evaluated and classified as follows:
    - If `I` is the keyword **New**, then a compile-time error occurs.
    - If `I` identifies a type, then the result is that type.
    - If `I` identifies one or more methods, then the result is a method group with no associated instance expression.
    - If `I` identifies one or more properties, then the result is a property group with no associated instance expression.
    - If `I` identifies a shared variable, and if the variable is read-only, and the reference occurs outside the shared constructor of the type in which the variable is declared, then the result is the value of the shared variable `I` in `E`. Otherwise, the result is the shared variable `I` in `E`.
    - If `I` identifies a shared event, the result is an event access with no associated instance expression.
    - If `I` identifies a constant, then the result is the value of that constant.
    - If `I` identifies an enumeration member, then the result is the value of that enumeration member.
    - Otherwise, `E.I` is an invalid member reference, and a compile-time error occurs.
- If `E` is classified as a variable or value, the type of which is `T`, and `I` is the name of an accessible member of `E`, then `E.I` is evaluated and classified as follows:
    - If `I` is the keyword **New** and `E` is an instance expression (**Me**, **MyBase**, or **MyClass**), then the result is a method group representing the instance constructors of the type of `E` with an associated instance expression of `E`. Otherwise, a compile-time error occurs.
    - If `I` identifies one or more methods, then the result is a method group with an associated instance expression of `E`.
    - If `I` identifies one or more properties, then the result is a property group with an associated instance expression of `E`.
    - If `I` identifies a shared variable or an instance variable, and if the variable is read-only, and the reference occurs outside a constructor of the class in which the variable is declared appropriate for the kind of variable (shared or instance), then the result is the value of the variable `I` in the object referenced by `E`. If `T` is a reference type, then the result is the variable `I` in the object referenced by `E`. Otherwise, if `T` is a value type and the expression `E` is classified as a variable, the result is a variable; otherwise the result is a value.
    - If `I` identifies an event, the result is an event access with an associated instance expression of `E`.
    - If `I` identifies a constant, then the result is the value of that constant.
    - If `I` identifies an enumeration member, then the result is the value of that enumeration member.
    - If `T` is **Object**, then the result is a late-bound member lookup classified as a property group with an associated instance expression of `E`.
- Otherwise, `E.I` is an invalid member reference, and a compile-time error occurs.

If the member being accessed is shared, any expression on the left side of the period is superfluous and is not evaluated unless the member access is done late-bound. For example, consider the following code:

```
Class C
    Public Shared F As Integer = 10
End Class

Module Test
    Public Function ReturnC() As C
        Console.WriteLine("Returning a new instance of C.")
        Return New C()
    End Function

    Public Sub Main()
```

```
        Console.WriteLine("The value of F is: " & ReturnC().F)
    End Sub
End Module
```

It prints `The value of F is: 10` because the function `ReturnC` does not need to be called to provide an instance of `C` to access the shared member `F`.

A late-bound member lookup is always classified as a property group, even though at run-time the identifier may resolve to a method group. At run time, the identifier is looked up and classified as a property or a method. If late-bound member lookup fails at run time, a **System.MissingMemberException** exception is thrown. Because late-bound member lookup is done solely off the run-time type of the associated instance expression, an object's run-time type is never an interface. Therefore, it is impossible to access interface members in a late-bound member access expression.

```
MemberAccessExpression ::=
   [ [ MemberAccessBase ] . ] IdentifierOrKeyword
MemberAccessBase ::= Expression | BuiltInTypeName
```

See Also

11.7 Dictionary Member Access | 11.4 Simple Expressions | 11.8 Invocation Expressions | 11.9 Index Expressions | 11.10 New Expressions | 11.11 Cast Expressions | 11.12 Operator Expressions | 11.3 Late-Binding Expressions | 9.2.5.2 Reference Parameters

11.7 Dictionary Member Access

A *dictionary member access expression* is used to look up a member of a collection. A dictionary member access takes the form of `E!I`, where `E` is an expression that is classified as a value and `I` is an identifier. The type of the expression must have a default property indexed by a single **String** parameter. The dictionary member access expression `E!I` is transformed into the expression `E.D("I")`, where `D` is the default property of `E`. For example:

```
Class Keys
    Public ReadOnly Default Property Item(ByVal s As String) As Integer
        Get
            Return 10
        End Get
    End Property
End Class

Module Test
    Sub Main()
        Dim x As Keys = new Keys()
        Dim y As Integer
        ' The two statements are equivalent.
        y = x!abc
        y = x("abc")
    End Sub
End Module
```

If an exclamation point is specified with no expression, the expression from the immediately containing **With** statement is assumed. If there is no containing **With** statement, a compile-time error occurs.

```
DictionaryAccessExpression ::= [ Expression ] ! IdentifierOrKeyword
```

See Also

11.6 Member Access Expressions

11.8 Invocation Expressions

An invocation expression consists of an invocation target, followed by an opening parenthesis, an optional argument list, and a closing parenthesis. The target expression must be classified as a method group or a value whose type is a delegate type. If the target expression is a value, then the target of the invocation expression becomes the method group referring to the `Invoke` member of the delegate type.

An argument list has two sections: positional arguments and named arguments. *Positional arguments* are expressions and must precede any named arguments. *Named arguments* start with an identifier that can match keywords, followed by `:=` and an expression.

Given a method group, overload resolution is applied to pick a single method applicable to the given argument list. If the method group only contains one method and that method takes no arguments and is a function, then the method group is interpreted as an invocation expression with an empty argument list and the result is used as the target of an index expression. If no method is applicable, a compile-time error occurs. If the applicable method is a function, then the result of the invocation expression is classified as a value typed as the return type of the function. If the applicable method is a subroutine, then the result is classified as void.

```
InvocationExpression ::= Expression [ ( [ ArgumentList ] ) ]
ArgumentList ::=
   PositionalArgumentList , NamedArgumentList |
   PositionalArgumentList |
   NamedArgumentList
PositionalArgumentList ::=
   Expression |
   PositionalArgumentList , [ Expression ]
NamedArgumentList ::=
   IdentifierOrKeyword : = Expression |
   NamedArgumentList , IdentifierOrKeyword : = Expression
```

See Also

11.8.4 Conditional Methods | 11.8.2 Applicable Methods | 11.8.1 Overloaded Method Resolution | 11.4 Simple Expressions | 11.6 Member Access Expressions | 11.9 Index Expressions | 11.10 New Expressions | 11.11 Cast Expressions | 11.12 Operator Expressions | 11.3 Late-Binding Expressions | 11.4.5 AddressOf Expressions | 6.2.3 Option Compare Statement | 11.8 Invocation Expressions | 9.2.5.2 Reference Parameters | Expressions (Visual Basic Language Concepts) | Argument Passing by Position and Name (Visual Basic Language Concepts)

11.8.1 Overloaded Method Resolution

Given a method group, the applicable method in the group for an argument list is determined as follows:

- Eliminate all inaccessible members from the set.
- Eliminate all members from the set that are not applicable to the argument list. If the set is empty, a compile-time error results. If only one member remains in the set, that is the applicable member.
- Eliminate all members from the set that require narrowing coercions to be applicable to the argument list, except for the case where the argument expression type is **Object**. If the set is empty, a compile-time error results. If only one member remains in the set, that is the applicable member.
- Eliminate all remaining members from the set that require narrowing coercions to be applicable to the argument list. If the set is empty and strict semantics are not being used, the invocation target expression is reclassified as a late-bound property access and the invocation expression is reinterpreted as an index expression. If strict semantics are being used and the set is empty, a compile-time error results. If only one member remains in the set, that is the applicable member.

  > **Note** The justification for this rule is that if a program is loosely-typed (that is, most or all variables are declared as **Object**), overload resolution can be difficult because all conversions from **Object** are narrowing. Rather than have the overload resolution fail in many situations (requiring strong typing of the arguments to the method call), resolution the appropriate overloaded method to call is deferred until run time. This allows the loosely-typed call to succeed without additional casts.

- Given any two members of the set, `M` and `N`, if `M` is more applicable than `N` to the argument list, eliminate `N` from the set. If only one member remains in the set, that is the applicable member. If two or more members remain because the members are equally applicable, a compile-time error results.

A member `M` is considered *more applicable* than `N` if, for each set of parameters `Mj` and `Nj` that matches an argument `Aj`, one of the following conditions is true:

- `Mj` and `Nj` have identical types, or
- There exists a widening conversion from the type of `Mj` to the type `Nj`.

It is possible for `M` and `N` to have the same signature if one or both contains an expanded paramarray parameter. In that case, the member with the fewest number of arguments matching expanded paramarray parameters is considered more applicable.

```
Imports System

Module Test
    Sub F(ByVal ParamArray a As Object())
        Console.WriteLine("F(Object())")
    End Sub

    Sub F()
        Console.WriteLine("F()")
    End Sub

    Sub F(ByVal a As Object, ByVal b As Object)
        Console.WriteLine("F(Object, Object)")
    End Sub

    Sub F(ByVal a As Object, ByVal b As object, _
          ByVal ParamArray c As Object())
        Console.WriteLine("Object, Object, Object()")
    End Sub

    Sub Main()
        F()
        F(1)
        F(1, 2)
        F(1, 2, 3)
    End Sub
End Module
```

The above example produces the following output:

```
F()
F(Object())
F(Object, Object)
F(Object, Object, Object())
```

In the example, the first and third calls to `F` prefer `F()` and `F(Object, Object)` because they match no arguments to expanded paramarray parameters. The fourth call to `F` prefers `F(Object, Object, Object())` because two arguments match non-expanded paramarray parameters. When a class declares a method with a paramarray parameter, it is not uncommon to also include some of the expanded forms as regular methods. By doing so it is possible to avoid the allocation of an array instance that occurs when an expanded form of a method with a paramarray parameter is invoked.

> **Note** The context of the method usage is not used in overload resolution (except for determining accessibility).

Also note that because of the named parameter syntax, the ordering of the actual and formal parameters may not be the same.

See Also

11.8.4 Conditional Methods | 11.8 Invocation Expressions | 11.8.2 Applicable Methods | 11.8 Invocation Expressions | Procedure Overloading (Visual Basic Language Concepts)

11.8.2 Applicable Methods

A method is *applicable* to a set of positional arguments and named arguments if the method can be invoked using the argument list. The argument expression list is first matched against the parameter list as follows:

- First, match each positional argument in order to the list of method parameters. If there are more positional arguments than parameters and the last parameter is not a paramarray, the method is not applicable. Otherwise, the paramarray parameter is expanded with parameters of the paramarray element type to match the number of positional arguments. If a positional argument is omitted, the method is not applicable.
- Next, match each named argument to a parameter with the given name. If one of the named arguments fails to match, matches a paramarray parameter, or matches an argument already matched with another positional or named argument, the method is not applicable.
- Finally, if parameters that have not been matched are not optional, the method is not applicable. If optional parameters remain, the default value specified in the optional parameter declaration is matched to the parameter. If an **Object** parameter does not specify a default value, then the expression **System.Reflection.Missing.Value** is used. If an optional **Integer** parameter has the **Microsoft.VisualBasic.CompilerServices.OptionCompareAttribute** attribute, then the literal `1` is supplied for text comparisons and the literal `0` otherwise.

The type of each argument expression must be implicitly convertible to the type of the parameter it matches. If the parameter is a reference parameter, the argument expression must be also implicitly convertible from the type of the parameter.

If a single argument expression matches a paramarray parameter and the type of the argument expression is convertible to both the type of the paramarray parameter and the paramarray element type, the method is applicable in both its expanded and unexpanded forms, with two exceptions. If the conversion from the type of the argument expression to the paramarray type is narrowing, then the method is only applicable in its expanded form. If the argument expression is the null literal **Nothing**, then the method is only applicable in its unexpanded form. For example:

```
Imports System

Module Test
    Sub F(ParamArray a As Object())
        Dim o As Object

        For Each o In a
            Console.Write(o.GetType().FullName)
            Console.Write(" ")
        Next o
        Console.WriteLine()
    End Sub

    Sub Main()
        Dim a As Object() = { 1, "Hello", 123.456 }
        Dim o As Object = a

        F(a)
        F(CType(a, Object))
        F(o)
        F(CType(o, Object()))
    End Sub
End Module
```

The above example produces the following output:

```
System.Int32 System.String System.Double
System.Object[]
System.Object[]
System.Int32 System.String System.Double
```

In the first and last invocations of `F`, the normal form of `F` is applicable because a widening conversion exists from the argument type to the parameter type (both are of type `Object()`), and the argument is passed as a regular value parameter. In the second and third invocations, the normal form of `F` is not applicable because no widening conversion exists from the argument type to the parameter type (conversions from `Object` to `Object()` are narrowing). However, the expanded form of `F` is applicable, and a

one-element `Object()` is created by the invocation. The single element of the array is initialized with the given argument value (which itself is a reference to an `Object()`).

See Also

11.8.4 Conditional Methods | 11.8 Invocation Expressions | 11.8.1 Overloaded Method Resolution | 11.8 Invocation Expressions

11.8.3 Passing Parameters

If a parameter is a value parameter, the matching argument expression must be classified as a value. The value is converted to the type of the parameter and passed in as the parameter at run time. If the parameter is a reference parameter and the matching argument expression is classified as a variable whose type is the same as the parameter, then a reference to the variable is passed in as the parameter at run time.

Otherwise, if the matching argument expression is classified as a variable, value, or property access, then a temporary variable of the type of the parameter is allocated. Before the method invocation at run time, the argument expression is reclassified as a value, converted to the type of the parameter, and assigned to the temporary variable. Then a reference to the temporary variable is passed in as the parameter. After the method invocation is evaluated, if the argument expression is classified as a variable or property access, the temporary variable is assigned to the variable expression or the property access expression. If the property access expression has no **Set** accessor, then the assignment is not performed.

See Also

11.8.4 Conditional Methods | 11.8 Invocation Expressions | 11.8.2 Applicable Methods | 11.8.1 Overloaded Method Resolution | 11.8 Invocation Expressions | Argument Passing ByRef (Visual Basic Language Concepts)

11.8.4 Conditional Methods

If the target method to which an invocation expression refers is a subroutine that is not a member of an interface and if the method has one or more **System.Diagnostics.ConditionalAttribute** attributes, evaluation of the expression depends on the conditional compilation constants defined at that point in the source file. Each instance of the attribute specifies a string, which names a conditional compilation constant. Each conditional compilation constant is evaluated as if it were part of a conditional compilation statement. If the constant evaluates to **True**, the expression is evaluated normally at run time. If the constant evaluates to **False**, the expression is not evaluated at all.

When looking for the attribute, the most derived declaration of an overridable method is checked.

> **Note** The attribute is not valid on functions or interface methods and is ignored if specified on either kind of method. Thus, conditional methods will only appear in invocation statements.

See Also

11.8 Invocation Expressions | 11.8.2 Applicable Methods | 11.8.1 Overloaded Method Resolution | 11.8 Invocation Expressions | #Const Directive (Visual Basic Language Reference) |
Declaring Conditional Compilation Constants and Enumerations (Visual Basic Language Concepts)

11.9 Index Expressions

An *index expression* results in an array element or reclassifies a property group into a property access. An index expression consists of, in order, an expression, an opening parenthesis, an index argument list, and a closing parenthesis. The target of the index expression must be classified as either a property group or a value. An index expression is processed as follows:

- If the target expression is classified as a value and if its type is not an array type, **Object**, or **System.Array**, the type must have a default property. The index is performed on a property group that represents all of the default properties of the type (which may have different names). Although it is not valid to declare a parameterless default property in Visual Basic .NET, other languages may allow declaring such a property. Consequently, indexing a property with no arguments is allowed.
- If the expression results in a value of an array type, the number of arguments in the argument list must be the same as the rank of the array type and may not include named arguments. If any of the indexes are invalid at run time, a **System.IndexOutOfRangeException** exception is thrown. Each expression must be implicitly convertible to type **Integer**. The result of the index expression is the variable at the specified index and is classified as a variable.
- If the expression is classified as a property group, overload resolution is used to determine whether one of the properties is applicable to the index argument list. If the property group only contains one property that has a **Get** accessor and if that accessor takes no arguments, then the property group is interpreted as an index expression with an empty argument list. The result is used as the target of the current index expression. If no properties are applicable, then a compile-time error occurs. Otherwise, the expression results in a property access with the associated instance expression (if any) of the property group.
- If the expression is classified as a late-bound property group or as a value whose type is **Object** or **System.Array**, the processing of the index expression is deferred until run time and the indexing is late bound. The expression results in a late-bound property access typed as **Object**. The associated instance expression is either the target expression, if it is a value, or the associated instance expression of the property group. At run time the expression is processed as follows:
    - If the expression is classified as a late-bound property group, the expression may result in either a method group, a property group, or a value (if the member is an instance or shared variable). If the result is a method group or property group, overload resolution is applied to the group to determine the correct method for the argument list. If overload resolution fails, a **System.Reflection.AmbiguousMatchException** exception is thrown. Then the result is processed either as a property access or as an invocation and the result is returned. If the invocation is of a subroutine, the result is **Nothing**.
    - If the run-time type of the target expression is an array type or **System.Array**, the result of the index expression is the value of the variable at the specified index.
    - Otherwise, the run-time type of the expression must have a default property and the index is performed on the property group that represents all of the default properties on the type. If the type has no default property, then a **System.MissingMemberException** exception is thrown.

Because late-bound overload resolution is done on the run-time type of the arguments, it is possible that an expression might produce different results based on whether it is evaluated at compile time or run time. The following example illustrates this difference:

```
Imports System

Class Base
End Class

Class Derived
    Inherits Base
End Class

Module Test
    Sub F(ByVal b As Base)
        Console.WriteLine("F(Base)")
    End Sub

    Sub F(ByVal d As Derived)
        Console.WriteLine("F(Derived)")
    End Sub

    Sub Main()
        Dim b As Base = New Derived()
        Dim o As Object = b
```

```
        F(b)
        F(o)
    End Sub
End Module
```

This code displays:

```
F(Base)
F(Derived)
```

```
IndexExpression ::= Expression ( [ ArgumentList ] )
```

See Also

11.4 Simple Expressions | 11.8 Invocation Expressions | 11.6 Member Access Expressions | 11.10 New Expressions | 11.11 Cast Expressions | 11.12 Operator Expressions | 11.3 Late-Binding Expressions | 9.2.5.2 Reference Parameters

11.10 New Expressions

The **New** operator is used to create new instances of types. There are three forms of **New** expressions:

- Object-creation expressions are used to create new instances of class types and value types.
- Array-creation expressions are used to create new instances of array types.
- Delegate-creation expressions are used to create new instances of delegate types.

A **New** expression is classified as a value and the result is the new instance of the type.

```
NewExpression ::=
   ObjectCreationExpression |
   ArrayCreationExpression |
   DelegateCreationExpression
```

See Also

11.4 Simple Expressions | 11.8 Invocation Expressions | 11.6 Member Access Expressions | 11.9 Index Expressions | 11.11 Cast Expressions | 11.12 Operator Expressions | 11.3 Late-Binding Expressions | 9.2.5.2 Reference Parameters

11.10.1 Object-Creation Expressions

An object-creation expression is used to create a new instance of a class type or a structure type. The type of an object creation expression must be a class type or a structure type and cannot be a **MustInherit** class. Given an object creation expression of the form `New T(A)`, where `T` is a class type or structure type and `A` is an optional argument list, overload resolution determines the correct constructor of `T` to call. If no constructor is callable, a compile-time error occurs; otherwise the expression results in the creation of a new instance of `T` using the chosen constructor. If there are no arguments, the parentheses may be omitted.

Where an instance is allocated depends on whether the instance is a class type or a value type. **New** instances of class types are created on the system heap, while new instances of value types are created directly on the stack.

```
ObjectCreationExpression ::= New TypeName [ ( [ ArgumentList ] ) ]
```

See Also

11.10.2 Array-Creation Expressions | 11.10.3 Delegate-Creation Expressions | 11.10 New Expressions |
New (Visual Basic Language Reference) | Using the New Keyword (Visual Basic Language Concepts)

11.10.2 Array-Creation Expressions

An array-creation expression is used to create a new instance of an array type. If an array size initialization modifier is supplied, the resulting array type is derived by deleting each of the individual arguments from the array size initialization argument list. The value of each argument determines the upper bound of the corresponding dimension in the newly allocated array instance. If the array-element initializer in the expression is not empty, each argument in the argument list must be a constant, and the rank and dimension lengths specified by the expression list must match those of the array element initializer.

If an array size initialization modifier is not supplied, then the type name must be an array type and the array element initializer must be empty or the rank of the specified array type must match that of the array element initializer. The individual dimension lengths are inferred from the number of elements in each of the corresponding nesting levels of the array element initializer. If the array-element initializer is empty, the length of each dimension is zero.

An array instance's rank and length of each dimension are constant for the entire lifetime of the instance. In other words, it is not possible to change the rank of an existing array instance, nor is it possible to resize its dimensions. Elements of arrays created by array-creation expressions are always initialized to their default value.

```
ArrayCreationExpression ::=
   New TypeName [ ArraySizeInitializationModifier ]
      ArrayElementInitializer
```

See Also

11.10.1 Object-Creation Expressions | 11.10.3 Delegate-Creation Expressions | 11.10 New Expressions |
New (Visual Basic Language Reference) | Using the New Keyword (Visual Basic Language Concepts)

11.10.3 Delegate-Creation Expressions

A delegate-creation expression is used to create a new instance of a delegate type. The argument of a delegate-creation expression must be an expression classified as a method pointer.

One of the methods referenced by the method pointer must exactly match the signature of the delegate type. A method matches the delegate type if the method is not declared **MustOverride** and if both their signatures and return types match. In the following example, the `A.f` variable is initialized with a delegate that refers to the second `Square` method because that method exactly matches the signature and return type of `DoubleFunc`.

```
Delegate Function DoubleFunc(x As Double) As Double
Class A
    Private f As New DoubleFunc(AddressOf Square)
    Overloads Shared Function Square(x As Single) As Single
        Return x * x
    End Function
    Overloads Shared Function Square(x As Double) As Double
        Return x * x
    End Function
End Class
```

Had the second `Square` method not been present, a compile-time error would have occurred.

The result of the expression is a delegate instance that refers to the matching method with the associated instance expression (if any) from the method pointer expression. If the instance expression is typed as a value type, then the value type is copied onto the system heap because a delegate can only point to a method of an object on the heap. The method and object to which a delegate refers remain constant for the entire lifetime of the delegate. In other words, it is not possible to change the target or object of a delegate after it has been created.

```
DelegateCreationExpression ::= New TypeName ( Expression )
```

See Also

11.10.1 Object-Creation Expressions | 11.10.2 Array-Creation Expressions | 11.10 New Expressions |
New (Visual Basic Language Reference) | Using the New Keyword (Visual Basic Language Concepts)

11.11 Cast Expressions

A cast expression coerces an expression to a given type. Specific cast keywords coerce expressions into the primitive types. Two general cast keywords, **CType** and **DirectCast**, coerce an expression into any type.

DirectCast is different from **CType** in the way it converts expressions that are typed as **Object**. When converting an expression of type **Object** whose run-time type is a primitive value type, **DirectCast** throws a **System.InvalidCastException** exception if the specified type is not the same as the run-time type of the expression or a **System.NullReferenceException** if the expression evaluates to **Nothing**. If the specified type and the run-time type of the expression are the same, however, the run-time performance of **DirectCast** is better than that of **CType**.

If no conversion exists from the type of the expression to the specified type, a compile-time error occurs. Otherwise, the expression is classified as a value and the result is the value produced by the conversion.

```
CastExpression ::=
   DirectCast ( Expression , TypeName ) |
   CType ( Expression , TypeName ) |
   CastTarget ( Expression )
CastTarget ::=
   CBool | CByte | CChar | CDate | CDec | CDbl |
   CInt | CLng | CObj | CShort | CSng | CStr
```

See Also

11.4 Simple Expressions | 11.8 Invocation Expressions | 11.6 Member Access Expressions | 11.9 Index Expressions | 11.10 New Expressions | 11.3 Late-Binding Expressions | 9.2.5.2 Reference Parameters | Conversion Functions (Visual Basic Language Reference) | Type Conversions (Visual Basic Language Concepts)

11.12 Operator Expressions

There are two kinds of operators. *Unary operators* take one operand and use prefix notation (for example, `-x`). *Binary operators* take two operands and use infix notation (for example, `x + y`). With the exception of the relational operators, which always result in **Boolean**, an operator defined for a particular type results in that type. The operands to an operator must always be classified as a value; the result of an operator expression is classified as a value.

```
OperatorExpression ::=
   ArithmeticOperatorExpression |
   RelationalOperatorExpression |
   LikeOperatorExpression |
   ConcatenationOperatorExpression |
   ShortCircuitLogicalOperatorExpression |
   LogicalOperatorExpression |
   ShiftOperatorExpression
```

See Also

11.12.1 Operator Precedence and Associativity | 11.12.2 Object Operands | 11.12.3 Operator Resolution | 11.13 Arithmetic Operators | 11.14 Relational Operators | 11.15 Like Operator | 11.16 Concatenation Operator | 11.17 Logical Operators | 11.18 Shift Operators | Operators (Visual Basic Language Reference) | Operators (Visual Basic Language Concepts)

11.12.1 Operator Precedence and Associativity

When an expression contains multiple binary operators, the *precedence* of the operators controls the order in which the individual binary operators are evaluated. For example, the expression `x + y * z` is evaluated as `x + (y * z)` because the ***** operator has higher precedence than the **+** operator. The following table lists the binary operators in descending order of precedence:

Category	**Operators**
Primary	All nonoperator expressions
Exponentiation	**^**
Unary negation	**+**, **-**
Multiplicative	*****, **/**
Integer division	****
Modulus	**Mod**
Additive	**+**, **-**
Concatenation	**&**
Shift	**<<**, **>>**
Relational	**=**, **<>**, **<**, **>**, **<=**, **>=**, **Like**, **Is**, **TypeOf**...**Is**
Logical NOT	**Not**
Logical AND	**And**, **AndAlso**
Logical OR	**Or**, **OrElse**
Logical XOR	**Xor**

When an expression contains two operators with the same precedence, the *associativity* of the operators controls the order in which the operations are performed. All binary operators are left-associative, meaning that operations are performed from left to right. Precedence and associativity can be controlled using parenthetical expressions.

See Also

11.12.2 Object Operands | 11.12.3 Operator Resolution | 11.13 Arithmetic Operators | 11.14 Relational Operators | 11.15 Like Operator | 11.16 Concatenation Operator | 11.17 Logical Operators |
Operator Precedence in Visual Basic (Visual Basic Language Reference) |
Efficient Combination of Operators (Visual Basic Language Concepts)

11.12.2 Object Operands

In addition to the regular types supported by each operator, all operators support operands of type **Object**. Operators applied to **Object** operands are handled similarly to late-bound method calls made on **Object** values: the run-time type of the operands, rather than the compile-time type, determines the validity and type of the operation. If strict semantics are specified by the compilation environment or by **Option Strict**, any operators with operands of type **Object** cause a compile-time error, except for the equality (**=**), inequality (**<>**), **TypeOf...Is**, and **Is** operators.

When given one or more operands of type **Object**, the outcome of the operation is the result of applying the operator to the operand types if the run-time types of the operands are types that are supported by the operator,. The value **Nothing** is treated as the default value of the type of the other operand in a binary operator expression. In a unary operator expression, or if both operands are **Nothing** in a binary operator expression, the type of the operation is **Integer**. The result of the operation is always then cast back to **Object**. If the operand types have no valid operator, a **System.InvalidCastException** exception is thrown. Conversions at run time are done without regard to whether they are implicit or explicit.

If the result of a numeric binary operation would produce an overflow exception (regardless of whether integer overflow checking is on or off), then the result type is promoted to the next wider numeric type, if possible. For example, consider the following code:

```
Imports System

Module Test
    Sub Main()
        Dim o As Object = CObj(CByte(2)) * CObj(CByte(255))

        Console.WriteLine(o.GetType().ToString() & " = " & o)
    End Sub
End Module
```

It prints the following result:

```
Short = 512
```

If no wider numeric type is available to hold the number, a **System.OverflowException** exception is thrown.

See Also

11.12.1 Operator Precedence and Associativity | 11.12.3 Operator Resolution | 11.13 Arithmetic Operators | 11.14 Relational Operators | 11.15 Like Operator | 11.16 Concatenation Operator | 11.17 Logical Operators | Operator (Visual Basic Language Reference) | Efficient Combination of Operators (Visual Basic Language Concepts)

11.12.3 Operator Resolution

Given a set of operands and an operator, the correct operator type to use is resolved as follows:

- If all operands are of the same type, and the operator is defined for the type, then no conversion occurs and the operator for that type is used.
- Any operand whose type is not defined for the operator is converted using the following steps and the operator is resolved against the new types:
    - The operand is converted to the next widest type that is defined for both the operator and the operand and to which it is implicitly convertible.
    - If there is no such type, then the operand is converted to the next narrowest type that is defined for both the operator and the operand and to which it is implicitly convertible.
    - If there is no such type or the conversion cannot occur, a compile-time error occurs.
- If one of the operands is typed as **String**, it is converted using the following rules and the operator is resolved against the new types. If **String** cannot be implicitly converted to the new type, a compile-time error occurs.
    - **Date** if the other operand is **Date**
    - **Boolean** if the other operand is **Boolean**
    - **Double** in all other cases
- Otherwise, the operands are converted to the wider of the operand types and the operator for that type is used. If the narrower operand type cannot be implicitly converted to the wider operator type, a compile-time error occurs.

For the purposes of operator resolution, the following type relationships are implicit:

- **Boolean** is the same width as **Byte**.
- **Boolean** is narrower than **Short**.
- **String** is wider than **Double**.
- **String** is wider than **Date**.
- **Char** is narrower than **String**.

Enumerated types are considered to be the same width as their underlying type.

See Also

11.12.1 Operator Precedence and Associativity | 11.12.2 Object Operands | 11.13 Arithmetic Operators | 11.14 Relational Operators | 11.15 Like Operator | 11.16 Concatenation Operator | 11.17 Logical Operators | Operator (Visual Basic Language Reference) | Efficient Combination of Operators (Visual Basic Language Concepts)

11.13 Arithmetic Operators

The *****, **/**, ****, **^**, **Mod**, **+**, and **–** operators are the *arithmetic operators*.

Floating-point arithmetic operations may be performed with higher precision than the result type of the operation. For example, some hardware architectures support an "extended" or "long double" floating-point type with greater range and precision than the **Double** type, and implicitly perform all floating-point operations using this higher-precision type. Hardware architectures can be made to perform floating-point operations with less precision only at excessive cost in performance; rather than require an implementation to forfeit both performance and precision, Visual Basic .NET allows the higher-precision type to be used for all floating-point operations. Other than delivering more precise results, this rarely has any measurable effects. However, in expressions of the form `x * y / z`, where the multiplication produces a result that is outside the **Double** range, but the subsequent division brings the temporary result back into the **Double** range, the fact that the expression is evaluated in a higher-range format may cause a finite result to be produced instead of infinity.

```
ArithmeticOperatorExpression ::=
   UnaryPlusExpression |
   UnaryMinusExpression |
   AdditionOperatorExpression |
   SubtractionOperatorExpression |
   MultiplicationOperatorExpression |
   DivisionOperatorExpression |
   ModuloOperatorExpression |
   ExponentOperatorExpression
```

See Also

11.13.1 Unary Plus Operator | 11.13.2 Unary Minus Operator | 11.13.3 Addition Operator | 11.13.4 Subtraction Operator | 11.13.5 Multiplication Operator | 11.13.6 Division Operators | 11.13.7 Mod Operator | 11.13.8 Exponentiation Operator | | 11.12.2 Object Operands | 11.12.3 Operator Resolution | 11.14 Relational Operators | 11.15 Like Operator | 11.16 Concatenation Operator | 11.17 Logical Operators | Arithmetic Operators (Visual Basic Language Reference) | Operators Listed by Functionality (Visual Basic Language Reference) | Arithmetic Operators (Visual Basic Language Concepts)

11.13.1 Unary Plus Operator

The unary plus operator is defined for the **Byte**, **Short**, **Integer**, **Long**, **Single**, **Double**, and **Decimal** types. The unary plus operator results in the value of the operand.

```
UnaryPlusExpression ::= + Expression
```

See Also

11.13.2 Unary Minus Operator | 11.17 Logical Operators | 11.13 Arithmetic Operators |
+ Operator (Visual Basic Language Reference) | Arithmetic Operators (Visual Basic Language Concepts)

11.13.2 Unary Minus Operator

The unary minus operator is defined for the following types:

- **Short**, **Integer**, and **Long**. The result is computed by subtracting the operand from zero. If integer overflow checking is on and the value of the operand is the maximum negative **Short**, **Integer**, or **Long**, a **System.OverflowException** exception is thrown. Otherwise, if the value of the operand is the maximum negative **Short**, **Integer**, or **Long**, the result is that same value, and the overflow is not reported.
- **Single** and **Double**. The result is the value of the operand with its sign inverted, including the values 0 and Infinity. If the operand is NaN, the result is also NaN.
- **Decimal**. The result is computed by subtracting the operand from zero.

```
UnaryMinusExpression ::= - Expression
```

See Also

11.13.1 Unary Plus Operator | 11.17 Logical Operators | 11.13 Arithmetic Operators |
- Operator (Visual Basic Language Reference) | Arithmetic Operators (Visual Basic Language Concepts)

11.13.3 Addition Operator

The addition operator computes the sum of the two operands. The addition operator is defined for the following types:

- **Byte**, **Short**, **Integer**, and **Long**. If integer overflow checking is on and the sum is outside the range of the result type, a **System.OverflowException** exception is thrown. Otherwise, overflows are not reported, and any significant high-order bits of the result are discarded.
- **Single** and **Double**. The sum is computed according to the rules of IEEE 754 arithmetic.
- **Decimal**. If the resulting value is too large to represent in the decimal format, a **System.OverflowException** exception is thrown. If the result value is too small to represent in the decimal format, the result is 0.
- **String**. The two **String** operands are concatenated together.

> **Note** When resolving the addition operator for two **Char** operands, the operands are converted to **String** values.

```
AdditionOperatorExpression ::= Expression + Expression
```

See Also

11.13.4 Subtraction Operator | 11.13.5 Multiplication Operator | 11.13.6 Division Operators | 11.13.7 Mod Operator | 11.13.8 Exponentiation Operator | 11.13 Arithmetic Operators | + Operator (Visual Basic Language Reference) | Arithmetic Operators (Visual Basic Language Concepts)

11.13.4 Subtraction Operator

The subtraction operator subtracts the second operand from the first operand. The subtraction operator is defined for the following types:

- **Byte**, **Short**, **Integer**, and **Long**. If integer overflow checking is on and the difference is outside the range of the result type, a **System.OverflowException** exception is thrown. Otherwise, overflows are not reported, and any significant high-order bits of the result are discarded.
- **Single** and **Double**. The difference is computed according to the rules of IEEE 754 arithmetic.
- **Decimal**. If the resulting value is too large to represent in the decimal format, a **System.OverflowException** exception is thrown. If the result value is too small to represent in the decimal format, the result is 0.

```
SubtractionOperatorExpression ::= Expression - Expression
```

See Also

11.13.3 Addition Operator | 11.13.5 Multiplication Operator | 11.13.6 Division Operators | 11.13.7 Mod Operator | 11.13.8 Exponentiation Operator | 11.13 Arithmetic Operators | - Operator (Visual Basic Language Reference) | Arithmetic Operators (Visual Basic Language Concepts)

11.13.5 Multiplication Operator

The multiplication operator computes the product of two operands. The multiplication operator is defined for the following types:

- **Byte**, **Short**, **Integer**, and **Long**. If integer overflow checking is on and the product is outside the range of the result type, a **System.OverflowException** exception is thrown. Otherwise, overflows are not reported, and any significant high-order bits of the result are discarded.
- **Single** and **Double**. The product is computed according to the rules of IEEE 754 arithmetic.
- **Decimal**. If the resulting value is too large to represent in the decimal format, a **System.OverflowException** exception is thrown. If the result value is too small to represent in the decimal format, the result is 0.

```
MultiplicationOperatorExpression ::= Expression * Expression
```

See Also

11.13.3 Addition Operator | 11.13.4 Subtraction Operator | 11.13.6 Division Operators | 11.13.7 Mod Operator | 11.13.8 Exponentiation Operator | 11.13 Arithmetic Operators | * Operator (Visual Basic Language Reference) | Arithmetic Operators (Visual Basic Language Concepts)

11.13.6 Division Operators

Division operators compute the quotient of two operands. There are two division operators: the regular (floating-point) division operator and the integer division operator.

The regular division operator is defined for the following types:

- **Single** and **Double**. The quotient is computed according to the rules of IEEE 754 arithmetic.
- **Decimal**. If the value of the right operand is zero, a **System.DivideByZeroException** exception is thrown. If the resulting value is too large to represent in the decimal format, a **System.OverflowException** exception is thrown. If the result value is too small to represent in the decimal format, the result is zero.

According to normal operator resolution rules, regular division purely between operands of types such as **Byte**, **Short**, **Integer**, and **Long** would cause both operands to be converted to type **Decimal**. However, when doing operator resolution on the division operator when neither type is **Decimal**, **Double** is considered narrower than **Decimal**. This convention is followed because **Double** division is more efficient than **Decimal** division.

The integer division operator is defined for **Byte**, **Short**, **Integer**, and **Long**. If the value of the right operand is zero, a **System.DivideByZeroException** exception is thrown. The division rounds the result towards zero, and the absolute value of the result is the largest possible integer that is less than the absolute value of the quotient of the two operands. The result is zero or positive when the two operands have the same sign, and zero or negative when the two operands have opposite signs. If the left operand is the maximum negative **Short**, **Integer**, or **Long**, and the right operand is –1, an overflow occurs; if integer overflow checking is on, a **System.OverflowException** exception is thrown. Otherwise, the overflow is not reported and the result is instead the value of the left operand.

```
DivisionOperatorExpression ::=
   FPDivisionOperatorExpression |
   IntegerDivisionOperatorExpression
FPDivisionOperatorExpression ::= Expression / Expression
IntegerDivisionOperatorExpression ::= Expression \ Expression
```

See Also

11.13.3 Addition Operator | 11.13.4 Subtraction Operator | 11.13.5 Multiplication Operator | 11.13.7 Mod Operator | 11.13.8 Exponentiation Operator |11.13 Arithmetic Operators | / Operator (Visual Basic Language Reference) | \ Operator (Visual Basic Language Reference) | Arithmetic Operators (Visual Basic Language Concepts)

11.13.7 Mod Operator

The **Mod** (modulo) operator computes the remainder of the division between two operands. The **Mod** operator is defined for the following types:

- **Byte**, **Short**, **Integer** and **Long**. The result of `x Mod y` is the value produced by `x - (x \ y) * y`. If `y` is zero, a **System.DivideByZeroException** exception is thrown. The modulo operator never causes an overflow.
- **Single** and **Double**. The remainder is computed according to the rules of IEEE 754 arithmetic.
- **Decimal**. If the value of the right operand is zero, a **System.DivideByZeroException** exception is thrown. If the resulting value is too large to represent in the decimal format, a **System.OverflowException** exception is thrown. If the result value is too small to represent in the decimal format, the result is zero.

```
ModuloOperatorExpression ::= Expression Mod Expression
```

See Also

11.13.3 Addition Operator | 11.13.4 Subtraction Operator | 11.13.5 Multiplication Operator | 11.13.6 Division Operators | 11.13.8 Exponentiation Operator | 11.13 Arithmetic Operators | Mod Operator (Visual Basic Language Reference) | Arithmetic Operators (Visual Basic Language Concepts)

11.13.8 Exponentiation Operator

The exponentiation operator computes the first operand raised to the power of the second operand. The exponentiation operator is defined for type **Double**. The value is computed according to the rules of IEEE 754 arithmetic.

```
ExponentOperatorExpression ::= Expression ^ Expression
```

See Also

11.13.3 Addition Operator | 11.13.4 Subtraction Operator | 11.13.5 Multiplication Operator | 11.13.6 Division Operators | 11.13.7 Mod Operator | 11.13 Arithmetic Operators | ^ Operator (Visual Basic Language Reference) | Arithmetic Operators (Visual Basic Language Concepts)

11.14 Relational Operators

The *relational operators* compare values to one other. The comparison operators are **=**, **<>**, **<**, **>**, **<=**, and **>=**. All of the relational operators result in a **Boolean** value.

The relational operators have the following general meaning:

- The **=** operator tests whether the two operands are equal.
- The **<>** operator tests whether the two operands are not equal.
- The **<** operator tests whether the first operand is less than the second operand.
- The **>** operator tests whether the first operand is greater than the second operand.
- The **<=** operator tests whether the first operand is less than or equal to the second operand.
- The **>=** operator tests whether the first operand is greater than or equal to the second operand.

The relational operators are defined for the following types:

- **Boolean**. The operators compare the truth values of the two operands. **True** is considered to be less than **False**, which matches with their numeric values.
- **Byte**, **Short**, **Integer**, and **Long**. The operators compare the numeric values of the two integral operands.
- **Single** and **Double**. The operators compare the operands according to the rules of the IEEE 754 standard.
- **Decimal**. The operators compare the numeric values of the two decimal operands.
- **Date**. The operators return the result of comparing the two date/time values.
- **Char**. The operators return the result of comparing the two Unicode values.
- **String**. The operators return the result of comparing the two values using either a binary comparison or a text comparison. The comparison used is determined by the compilation environment and the **Option Compare** statement. A binary comparison determines whether the numeric Unicode value of each character in each string is the same. A text comparison does a Unicode text comparison based on the current culture in use on the .NET Framework. When doing a string comparison, a null reference is equivalent to the string literal **""**.

```
RelationalOperatorExpression ::=
   Expression = Expression |
   Expression < > Expression |
   Expression < Expression |
   Expression > Expression |
   Expression < = Expression |
   Expression > = Expression
```

See Also

11.12.1 Operator Precedence and Associativity | 11.12.2 Object Operands | 11.12.3 Operator Resolution | 11.13 Arithmetic Operators | 11.15 Like Operator | 11.16 Concatenation Operator | 11.17 Logical Operators | Comparison Operators (Visual Basic Language Reference) | Comparison Operators (Visual Basic Language Concepts)

11.15 Like Operator

The **Like** operator is defined for the **String** type and determines whether a string matches a given pattern. The first operand is the string being matched, and the second operand is the pattern to match against. The pattern is made up of Unicode characters. The following character sequences have special meanings:

- The character `?` matches any single character.
- The character `*` matches zero or more characters.
- The character `#` matches any single digit (0–9).
- A list of characters surrounded by brackets (`[ab...]`) matches any single character in the list.
- A list of characters surrounded by brackets and prefixed by an exclamation point (`[!ab...]`) matches any single character not in the character list.

Two characters in a character list separated by a hyphen (`-`) specify a range of Unicode characters starting with the first character and ending with the second character. If the second character is not later in the sort order than the first character, a run-time exception occurs. A hyphen that appears at the beginning or end of a character list specifies itself.

> **Note** To match the special characters left bracket (`[`), question mark (`?`), number sign (`#`), and asterisk (`*`), brackets must enclose them. The right bracket (`]`) can not be used within a group to match itself, but it can be used outside a group as an individual character. The character sequence `[]` is considered to be the string literal `""`.

Also note that character comparisons and ordering for character lists are dependent on the type of comparisons being used. If binary comparisons are being used, character comparisons and ordering are based on the numeric Unicode values. If text comparisons are being used, character comparisons and ordering are based on the current locale being used on the .NET Framework.

In some languages, special characters in the alphabet represent two separate characters and vice versa. For example, several languages use the character æ to represent the characters `a` and `e` when they appear together, while the characters ˆ and `O` can be used to represent the character Ô. When using text comparisons, the **Like** operator recognizes such cultural equivalences. In that case, an occurrence of the single special character in either pattern or string matches the equivalent two-character sequence in the other string. Similarly, a single special character in pattern enclosed in brackets (by itself, in a list, or in a range) matches the equivalent two-character sequence in the string and vice versa.

```
LikeOperatorExpression ::= Expression Like Expression
```

See Also

11.12.1 Operator Precedence and Associativity | 11.12.2 Object Operands | 11.12.3 Operator Resolution | 11.13 Arithmetic Operators | 11.14 Relational Operators | 11.16 Concatenation Operator | 11.17 Logical Operators | Like Operator (Visual Basic Language Reference)

11.16 Concatenation Operator

The *concatenation operator* is defined for the **String** type. To make string concatenation simpler, for the purposes of operator resolution, all conversions to **String** are considered to be widening, regardless of whether strict semantics are used. A concatenation operation results in a string that is the concatenation of the two operands in order from left to right.

In concatenation expressions, **Nothing** is treated as if it were the empty string literal **""**. Also, if only one operand is typed as **System.DBNull**, that operand is treated as if it was the literal **Nothing**.

```
ConcatenationOperatorExpression ::= Expression & Expression
```

See Also

11.12.1 Operator Precedence and Associativity | 11.12.2 Object Operands | 11.12.3 Operator Resolution | 11.13 Arithmetic Operators | 11.14 Relational Operators | 11.15 Like Operator | 11.17 Logical Operators | & Operator (Visual Basic Language Reference)

11.17 Logical Operators

The **And**, **Not**, **Or**, and **Xor** operators, which are called the logical operators, are evaluated as follows:

- For the **Boolean** type:
  - A logical **And** operation is performed on its two operands.
  - A logical **Not** operation is performed on its operand.
  - A logical **Or** operation is performed on its two operands.
  - A logical exclusive-**Or** operation is performed on its two operands.
- For **Byte**, **Short**, **Integer**, **Long**, and all enumerated types, the specified operation is performed on each bit of the binary representation of the two operand(s):
  - **And**: The result bit is 1 if both bits are 1; otherwise the result bit is 0.
  - **Not**: The result bit is 1 if the bit is 0; otherwise the result bit is 1.
  - **Or**: The result bit is 1 if either bit is 1; otherwise the result bit is 0.
  - **Xor**: The result bit is 1 if either bit is 1 but not both bits; otherwise the result bit is 0 (that is, 1 **Xor** 0 = 1, 1 **Xor** 1 = 0).

No overflows are possible from these operations. The enumerated type operators do the bitwise operation on the underlying type of the enumerated type, but the return value is the enumerated type.

```
LogicalOperatorExpression ::=
   Not Expression |
   Expression And Expression |
   Expression Or Expression |
   Expression Xor Expression
```

See Also

11.12.1 Operator Precedence and Associativity | 11.12.2 Object Operands | 11.12.3 Operator Resolution | 11.13 Arithmetic Operators | 11.14 Relational Operators | 11.15 Like Operator | 11.16 Concatenation Operator | Operators Listed by Functionality (Visual Basic Language Reference)

11.17.1 Short-circuiting Logical Operators

The **AndAlso** and **OrElse** operators, which are the short-circuiting logical operators, are only defined for the type **Boolean**. Because of their short circuiting behavior, the second operand is not evaluated at run time if the operator result is known after evaluating the first operand.

The short-circuiting logical operators are evaluated as follows:

- If the first operand in an **AndAlso** operation evaluates to **False**, the expression returns **False**. Otherwise, the second operand is evaluated and a logical **And** operation is performed on the two results.
- If the first operand in an **OrElse** operation evaluates to **True**, the expression returns **True**. Otherwise, the second operand is evaluated and a logical **Or** operation is performed on its two results.

When evaluating the **AndAlso** or **OrElse** operators, the first operand is evaluated only once, and the second operand is either not evaluated or evaluated exactly once. For example, consider the following code:

```
Imports System

Module Test
    Function TrueValue() As Boolean
        Console.Write(" True")
        Return True
    End Function

    Function FalseValue() As Boolean
        Console.Write(" False")
        Return False
    End Function

    Sub Main()
        Console.Write("And:")
        If FalseValue() And TrueValue() Then
        End If
        Console.WriteLine()

        Console.Write("Or:")
        If TrueValue() Or FalseValue() Then
        End If
        Console.WriteLine()

        Console.Write("AndAlso:")
        If FalseValue() AndAlso TrueValue() Then
        End If
        Console.WriteLine()

        Console.WriteLine("OrElse:")
        If TrueValue() OrElse FalseValue() Then
        End If
        Console.WriteLine()
    End Sub
End Module
```

It prints the following result:

```
And: False True
Or: True False
AndAlso: False
OrElse: True
```

```
ShortCircuitLogicalOperatorExpression ::=
   Expression AndAlso Expression |
   Expression OrElse Expression
```

See Also

11.12.1 Operator Precedence and Associativity | 11.12.2 Object Operands | 11.12.3 Operator Resolution | 11.13 Arithmetic Operators | 11.14 Relational Operators | 11.15 Like Operator | 11.16 Concatenation Operator | Operators Listed by Functionality (Visual Basic Language Reference)

11.18 Shift Operators

The binary operators **<<** and **>>** perform bit shifting operations. The operators are defined for the **Byte**, **Short**, **Integer** and **Long** types. Unlike the other binary operators, the result type of a shift operation is determined as if the operator was a unary operator with just the left operand. The type of the right operand must be implicitly convertible to **Integer** and is not used in determining the result type of the operation.

The **<<** operator causes the bits in the first operand to be shifted left the number of places specified by the shift amount. The high-order bits outside the range of the result type are discarded and the low-order vacated bit positions are zero-filled.

The **>>** operator causes the bits in the first operand to be shifted right the number of places specified by the shift amount. The low-order bits are discarded and the high-order vacated bit positions are set to zero if the left operand is positive or to one if negative. If the left operand is of type **Byte**, the vacant high-order bits are zero-filled.

The shift operators shift the bits of the underlying representation of the first operand by the amount of the second operand. If the value of the second operand is greater than the number of bits in the first operand, or is negative, then the shift amount is computed as `RightOperand And SizeMask` where `SizeMask` is:

LeftOperand Type	SizeMask
Byte	7 (`&H7`)
Short	15 (`&HF`)
Integer	31 (`&H1F`)
Long	63 (`&H3F`)

If the shift amount is zero, the result of the operation is identical to the value of the first operand. No overflows are possible from these operations.

```
ShiftOperatorExpression ::=
   Expression << Expression |
   Expression >> Expression
```

See Also

11.12.1 Operator Precedence and Associativity | 11.12.2 Object Operands | 11.12.3 Operator Resolution | 11.13 Arithmetic Operators | 11.14 Relational Operators | 11.15 Like Operator | 11.16 Concatenation Operator | Operators Listed by Functionality (Visual Basic Language Reference)

12. Visual Basic .NET Grammar Summary

This section summarizes the Visual Basic language grammar. For information on how to read the grammar, see 1.1 Grammar Notation.

2. Lexical Grammar

```
Start ::= [ LogicalLine+ ]
LogicalLine ::= [ LogicalLineElement+ ] [ Comment ] LineTerminator
LogicalLineElement ::= WhiteSpace | LineContinuation | Token
Token ::= Identifier | Keyword | Literal | Separator | Operator
```

2.1 Characters and Lines

```
Character ::= < any Unicode character except a LineTerminator >
```

2.1.1 Line Terminators

```
LineTerminator ::=
   < Unicode carriage return character (0x000D) > |
   < Unicode linefeed character (0x000A) > |
   < Unicode carriage return character > < Unicode linefeed character > |
   < Unicode line separator character (0x2028) > |
   < Unicode paragraph separator character (0x2029) >
```

2.1.2 Line Continuation

```
LineContinuation ::= WhiteSpace _ [ WhiteSpace+ ] LineTerminator
```

2.1.3 White Space

```
WhiteSpace ::=
   < Unicode blank characters (class Zs) > |
   < Unicode tab character (0x0009) >
```

2.1.4 Comments

```
Comment ::= CommentMarker [ Character+ ]
CommentMarker ::= SingleQuoteCharacter | REM
SingleQuoteCharacter ::=
   ' |
   < Unicode left single-quote character (0x2018) |
   < Unicode right single-quote character (0x2019)
```

2.2 Identifiers

```
Identifier ::=
   NonEscapedIdentifier [ TypeCharacter ] |
   EscapedIdentifier
NonEscapedIdentifier ::= < IdentifierName but not Keyword >
EscapedIdentifier ::= [ IdentifierName ]
IdentifierName ::= IdentifierStart [ IdentifierCharacter+ ]
IdentifierStart ::=
   AlphaCharacter |
```

```
   UnderscoreCharacter IdentifierCharacter
IdentifierCharacter ::=
   UnderscoreCharacter |
   AlphaCharacter |
   NumericCharacter |
   CombiningCharacter |
   FormattingCharacter
AlphaCharacter ::=
   < Unicode alphabetic character (classes Lu, Ll, Lt, Lm, Lo, Nl) >
NumericCharacter ::= < Unicode decimal digit character (class Nd) >
CombiningCharacter ::= < Unicode combining character (classes Mn, Mc) >
FormattingCharacter ::= < Unicode formatting character (class Cf) >
UnderscoreCharacter ::= < Unicode connection character (class Pc) >
IdentifierOrKeyword ::= Identifier | Keyword
```

2.2.1 Type Characters

```
TypeCharacter ::=
   IntegerTypeCharacter |
   LongTypeCharacter |
   DecimalTypeCharacter |
   SingleTypeCharacter |
   DoubleTypeCharacter |
   StringTypeCharacter
IntegerTypeCharacter ::= %
LongTypeCharacter ::= &
DecimalTypeCharacter ::= @
SingleTypeCharacter ::= !
DoubleTypeCharacter ::= #
StringTypeCharacter ::= $
```

2.3 Keywords

```
Keyword ::= < member of keyword table at 2.3 Keywords >
```

2.4 Literals

```
Literal ::=
   BooleanLiteral |
   IntegerLiteral |
   FloatingPointLiteral |
   StringLiteral |
   CharacterLiteral |
   DateLiteral |
   Nothing
```

2.4.1 Boolean Literals

```
BooleanLiteral ::= True | False
```

2.4.2 Integer Literals

```
IntegerLiteral ::= IntegralLiteralValue [ IntegralTypeCharacter ]
IntegralLiteralValue ::= IntLiteral | HexLiteral | OctalLiteral
IntegralTypeCharacter ::=
   ShortCharacter |
   IntegerCharacter |
   LongCharacter |
```

```
   IntegerTypeCharacter |
   LongTypeCharacter
ShortCharacter ::= S
IntegerCharacter ::= I
LongCharacter ::= L
IntLiteral ::= Digit+
HexLiteral ::= & H HexDigit+
OctalLiteral ::= & O OctalDigit+
Digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
HexDigit ::=
   0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F
OctalDigit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
```

2.4.3 Floating-Point Literals

```
FloatingPointLiteral ::=
   FloatingPointLiteralValue [ FloatingPointTypeCharacter ] |
   IntLiteral FloatingPointTypeCharacter
FloatingPointTypeCharacter ::=
   SingleCharacter |
   DoubleCharacter |
   DecimalCharacter |
   SingleTypeCharacter |
   DoubleTypeCharacter |
   DecimalTypeCharacter
SingleCharacter ::= F
DoubleCharacter ::= R
DecimalCharacter ::= D
FloatingPointLiteralValue ::=
   IntLiteral . IntLiteral [ Exponent ] |
   . IntLiteral [ Exponent ] |
   IntLiteral Exponent
Exponent ::= E [ Sign ] IntLiteral
Sign ::= + | -
```

2.4.4 String Literals

```
StringLiteral ::=
   DoubleQuoteCharacter [ StringCharacter+ ] DoubleQuoteCharacter
DoubleQuoteCharacter ::=
   " |
   < Unicode left double-quote character (0x201C) > |
   < Unicode right double-quote character (0x201D) >
StringCharacter ::=
   < Character except for DoubleQuoteCharacter > |
   DoubleQuoteCharacter DoubleQuoteCharacter
```

2.4.5 Character Literals

```
CharacterLiteral ::=
   DoubleQuoteCharacter StringCharacter DoubleQuoteCharacter C
```

2.4.6 Date Literals

```
DateLiteral ::= # [ Whitespace+ ] DateOrTime [ Whitespace+ ] #
DateOrTime ::=
   DateValue Whitespace+ TimeValue |
   DateValue |
   TimeValue
DateValue ::=
```

```
   MonthValue / DayValue / YearValue |
   MonthValue – DayValue - YearValue
TimeValue ::=
   HourValue : MinuteValue [ : SecondValue ] [ WhiteSpace+ ] [ AMPM ]
MonthValue ::= IntLiteral
DayValue ::= IntLiteral
YearValue ::= IntLiteral
HourValue ::= IntLiteral
MinuteValue ::= IntLiteral
SecondValue ::= IntLiteral
AMPM ::= AM | PM
```

2.4.7 Nothing

```
Nothing ::= Nothing
```

2.5 Separators

```
Separator ::= ( | ) | { | } | ! | # | , | . | :
```

2.6 Operator Characters

```
Operator ::= & | * | + | - | / | \ | ^ | < | = | >
```

3. Preprocessing Directives

3.1 Conditional Compilation

```
Start ::= [ ConditionalElement+ ]
ConditionalElement ::=
   ConditionalConstantDeclaration |
   ConditionalIfGroup |
   LogicalLine
```

3.1.1 Conditional Constant Directives

```
ConditionalConstantDeclaration ::=
   # Const Identifier = ConstantExpression LineTerminator
```

3.1.2 Conditional Compilation Directives

```
ConditionalIfGroup ::=
   # If ConstantExpression [ Then ] LineTerminator
   [ ConditionalElement+ ]
   [ ConditionalElseIfGroup+ ]
   [ ConditionalElseGroup ]
   # End If LineTerminator
ConditionalElseIfGroup ::=
   # ElseIf ConstantExpression [ Then ] LineTerminator
   [ ConditionalElement+ ]
ConditionalElseGroup ::=
   # Else LineTerminator
   [ ConditionalElement+ ]
```

3.2 External Source Directives

```
Start ::= [ ExternalSourceElement+ ]
ExternalSourceElement ::= ExternalSourceGroup | LogicalLine
ExternalSourceGroup ::=
   # ExternalSource ( StringLiteral , IntLiteral ) LineTerminator
   [ LogicalLine+ ]
   # End ExternalSource LineTerminator
```

3.3 Region Directives

```
Start ::= [ RegionElement+ ]
RegionElement ::= RegionGroup | LogicalLine
RegionGroup ::=
   # Region StringLiteral LineTerminator
   [ LogicalLine+ ]
   # End Region LineTerminator
```

4. General Concepts

4.6 Accessibility

```
AccessModifier ::= Public | Protected | Friend | Private
```

4.7 Type and Namespace Names

```
QualifiedIdentifier ::=
   Identifier |
   QualifiedIdentifier . IdentifierOrKeyword
NamespaceOrTypeName ::= QualifiedIdentifier
```

5. Attributes

5.2 Attribute Blocks

```
Attributes ::= < AttributeList >
AttributeList ::=
   Attribute |
   AttributeList , Attribute
Attribute ::=
   [ AttributeModifier : ] TypeName [ ( [ AttributeArguments ] ) ]
AttributeModifier ::= Assembly | Module
```

5.2.2 Attribute Arguments

```
AttributeArguments ::=
   AttributePositionalArgumentList |
   AttributePositionalArgumentList , VariablePropertyInitializerList |
   VariablePropertyInitializerList
AttributePositionalArgumentList ::=
   ConstantExpression |
   AttributePositionalArgumentList , ConstantExpression
VariablePropertyInitializerList ::=
   VariablePropertyInitializer |
   VariablePropertyInitializerList , VariablePropertyInitializer
VariablePropertyInitializer ::=
   IdentifierOrKeyword : = ConstantExpression
```

6. Source Files and Namespaces

```
Start ::=
   [ OptionStatement+ ]
   [ ImportsStatement+ ]
   [ Attributes ]
   [ NamespaceMemberDeclaration+ ]
```

6.2 Compilation Options

```
OptionStatement ::=
   OptionExplicitStatement |
   OptionStrictStatement |
   OptionCompareStatement
```

6.2.1 Option Explicit Statement

```
OptionExplicitStatement ::= Option Explicit [ OnOff ] LineTerminator
OnOff ::= On | Off
```

6.2.2 Option Strict Statement

```
OptionStrictStatement ::= Option Strict [ OnOff ] LineTerminator
```

6.2.3 Option Compare Statement

```
OptionCompareStatement ::= Option Compare CompareOption LineTerminator
CompareOption ::= Binary | Text
```

6.3 Imports Statement

```
ImportsStatement ::= Imports ImportsClauses LineTerminator
ImportsClauses ::=
   ImportsClause |
   ImportsClauses , ImportsClause
ImportsClause ::= ImportsAliasClause | ImportsNamespaceClause
```

6.3.1 Import Aliases

```
ImportsAliasClause ::= Identifier = NamespaceOrTypeName
```

6.3.2 Namespace Imports

```
ImportsNamespaceClause ::= NamespaceOrTypeName
```

6.4.1 Namespace Declarations

```
NamespaceDeclaration ::=
   Namespace QualifiedIdentifier LineTerminator
   [ NamespaceMemberDeclaration+ ]
   End Namespace LineTerminator
```

6.4.2 Namespace Members

```
NamespaceMemberDeclaration ::=
   NamespaceDeclaration |
   TypeDeclaration
TypeDeclaration ::=
   ModuleDeclaration |
   NonModuleDeclaration
NonModuleDeclaration ::=
   EnumDeclaration |
   StructureDeclaration |
   InterfaceDeclaration |
   ClassDeclaration |
   DelegateDeclaration
```

7. Types

```
TypeName ::=
   NamespaceOrTypeName |
   BuiltInTypeName |
   ArrayTypeName
BuiltInTypeName ::= Object | PrimitiveTypeName
```

7.2 Interface Implementation

```
TypeImplementsClause ::= Implements Implements LineTerminator
Implements ::=
   TypeName |
   Implements , TypeName
```

7.3 Primitive Types

```
PrimitiveTypeName ::= NumericTypeName | Boolean | Date | Char | String
NumericTypeName ::= IntegralTypeName | FloatingPointTypeName | Decimal
IntegralTypeName ::= Byte | Short | Integer | Long
FloatingPointTypeName ::= Single | Double
```

7.4 Enumerations

```
EnumDeclaration ::=
   [ Attributes ] [ EnumModifier+ ] Enum Identifier
      [ As IntegralTypeName ] LineTerminator
   EnumMemberDeclaration+
   End Enum LineTerminator
EnumModifier ::= AccessModifier | Shadows
```

7.4.1 Enumeration Members

```
EnumMemberDeclaration ::=
   [ Attributes ] Identifier [ = ConstantExpression ] LineTerminator
```

7.5 Classes

```
ClassDeclaration ::=
```

```
   [ Attributes ] [ ClassModifier+ ] Class Identifier LineTerminator
   [ ClassBase ]
   [ TypeImplementsClause+ ]
   [ ClassMemberDeclaration+ ]
   End Class LineTerminator
ClassModifier ::= AccessModifier | Shadows | MustInherit | NotInheritable
```

7.5.1 Class Base Specification

```
ClassBase ::= Inherits TypeName LineTerminator
```

7.5.2 Class Members

```
ClassMemberDeclaration ::=
   NonModuleDeclaration |
   EventMemberDeclaration |
   VariableMemberDeclaration |
   ConstantMemberDeclaration |
   MethodMemberDeclaration |
   PropertyMemberDeclaration |
   ConstructorMemberDeclaration
```

7.6 Structures

```
StructureDeclaration ::=
   [ Attributes ] [ StructureModifier+ ] Structure Identifier
      LineTerminator
   [ TypeImplementsClause+ ]
   [ StructMemberDeclaration+ ]
   End Structure LineTerminator
StructureModifier ::= AccessModifier | Shadows
```

7.6.1 Structure Members

```
StructMemberDeclaration ::=
   NonModuleDeclaration |
   VariableMemberDeclaration |
   ConstantMemberDeclaration |
   EventMemberDeclaration |
   MethodMemberDeclaration |
   PropertyMemberDeclaration |
   ConstructorMemberDeclaration
```

7.7 Standard Modules

```
ModuleDeclaration ::=
   [ Attributes ] [ AccessModifier+ ] Module Identifier LineTerminator
   [ ModuleMemberDeclaration+ ]
   End Module LineTerminator
```

7.7.1 Standard Module Members

```
ModuleMemberDeclaration ::=
   NonModuleDeclaration |
   VariableMemberDeclaration |
   ConstantMemberDeclaration |
```

```
   EventMemberDeclaration |
   MethodMemberDeclaration |
   PropertyMemberDeclaration |
   ConstructorMemberDeclaration
```

7.8 Interfaces

```
InterfaceDeclaration ::=
   [ Attributes ] [ InterfaceModifier+ ] Interface Identifier
      LineTerminator
   [ InterfaceBase+ ]
   [ InterfaceMemberDeclaration+ ]
   End Interface LineTerminator
InterfaceModifier ::= AccessModifier | Shadows
```

7.8.1 Interface Inheritance

```
InterfaceBase ::= Inherits InterfaceBases LineTerminator
InterfaceBases ::=
   TypeName |
   InterfaceBases , TypeName
```

7.8.2 Interface Members

```
InterfaceMemberDeclaration ::=
   NonModuleDeclaration |
   EventMemberDeclaration |
   MethodMemberDeclaration |
   PropertyMemberDeclaration
```

7.9 Arrays

```
ArrayTypeName ::= TypeName ArrayTypeModifiers
ArrayTypeModifiers ::= ArrayTypeModifier+
ArrayTypeModifier ::= ( [ RankList ] )
RankList ::=
   , |
   RankList ,
ArrayNameModifier ::=
   ArrayTypeModifiers |
   ArraySizeInitializationModifier
```

7.10 Delegates

```
DelegateDeclaration ::=
   [ Attributes ] [ DelegateModifier+ ] Delegate MethodDeclaration
DelegateModifier ::= AccessModifier | Shadows
```

9. Type Members

9.1 Interface Method Implementation

```
ImplementsClause ::= [ Implements ImplementsList ]
ImplementsList ::=
   InterfaceMemberSpecifier |
   ImplementsList , InterfaceMemberSpecifier
```

```
InterfaceMemberSpecifier ::= TypeName . Identifier
```

9.2 Methods

```
MethodMemberDeclaration ::= MethodDeclaration | ExternalMethodDeclaration
```

9.2.1 Regular Method Declarations

```
MethodDeclaration ::= SubDeclaration | FunctionDeclaration
SubDeclaration ::=
   [ Attributes ] [ ProcedureModifier+ ] Sub Identifier
      [ ( [ ParameterList ] ) ] [ HandlesOrImplements ] LineTerminator
   [ Block ]
   [ End Sub LineTerminator ]
FunctionDeclaration ::=
   [ Attributes ] [ ProcedureModifier+ ] Function Identifier
      [ ( [ ParameterList ] ) ] [ As [ Attributes ] TypeName ]
      [ HandlesOrImplements ] LineTerminator
   [ Block ]
   [ End Function LineTerminator ]
ProcedureModifier ::=
   AccessModifier |
   Shadows |
   Shared |
   Overridable |
   NotOverridable |
   MustOverride |
   Overrides |
   Overloads
HandlesOrImplements ::= HandlesClause | ImplementsClause
```

9.2.2 External Method Declarations

```
ExternalMethodDeclaration ::=
   ExternalSubDeclaration |
   ExternalFunctionDeclaration
ExternalSubDeclaration ::=
   [ Attributes ] [ ExternalMethodModifier+ ] Declare
      [ CharsetModifier ] Sub Identifier LibraryClause
      [ AliasClause ] [ ( [ ParameterList ] ) ] LineTerminator
ExternalFunctionDeclaration ::=
   [ Attributes ] [ ExternalMethodModifier+ ] Declare
      [ CharsetModifier ] Function Identifier LibraryClause
      [ AliasClause ] [ ( [ ParameterList ] ) ]
      [ As [ Attributes ] TypeName ] LineTerminator
ExternalMethodModifier ::= AccessModifier | Shadows | Overloads
CharsetModifier ::= Ansi | Unicode | Auto
LibraryClause ::= Lib StringLiteral
AliasClause ::= Alias StringLiteral
```

9.2.5 Method Parameters

```
ParameterList ::=
   Parameter |
   ParameterList , Parameter
Parameter ::=
   [ Attributes ] ParameterModifier+ Identifier [ As TypeName ]
      [ = ConstantExpression ]
ParameterModifier ::= ByVal | ByRef | Optional | ParamArray
```

9.2.6 Event Handling

```
HandlesClause ::= [ Handles EventHandlesList ]
EventHandlesList ::=
   EventMemberSpecifier |
   EventHandlesList , EventMemberSpecifier
EventMemberSpecifier ::=
   Identifier . Identifier
   MyBase . Identifier
```

9.3 Constructors

```
ConstructorMemberDeclaration ::=
   [ Attributes ] [ ConstructorModifier+ ] Sub New
      [ ( [ ParameterList ] ) ] LineTerminator
   [ Block ]
   End Sub LineTerminator
ConstructorModifier ::= AccessModifier | Shared
```

9.4 Events

```
EventMemberDeclaration ::=
   [ Attributes ] [ EventModifiers+ ] Event Identifier ParametersOrType
      [ ImplementsClause ] LineTerminator
ParametersOrType ::=
   [ ( [ ParameterList ] ) ] |
   As TypeName
EventModifiers ::= AccessModifier | Shadows | Shared
```

9.5 Constants

```
ConstantMemberDeclaration ::=
   [ Attributes ] [ ConstantModifier+ ] Const
      ConstantDeclarators LineTerminator
ConstantModifier ::= AccessModifier | Shadows
ConstantDeclarators ::=
   ConstantDeclarator |
   ConstantDeclarators , ConstantDeclarator
ConstantDeclarator ::= Identifier [ As TypeName ] =
   ConstantExpression LineTerminator
```

9.6 Instance and Shared Variables

```
VariableMemberDeclaration ::=
   [ Attributes ] [ VariableModifier+ ] [ Dim ] VariableDeclarators
      LineTerminator
VariableModifier ::=
   AccessModifier |
   Shadows |
   Shared |
   ReadOnly |
   WithEvents
VariableDeclarators ::=
   VariableDeclarator |
   VariableDeclarators , VariableDeclarator
VariableDeclarator ::=
   VariableIdentifiers [ As TypeName ] |
   VariableIdentifier [ As [ New ] TypeName [ ( ArgumentList ) ] ]
      [ = VariableInitializer ]
```

```
VariableIdentifiers ::=
   VariableIdentifier |
   VariableIdentifiers , VariableIdentifier
VariableIdentifier ::= Identifier [ ArrayNameModifier ]
```

9.6.3 Variable Initializers

```
VariableInitializer ::= RegularInitializer | ArrayElementInitializer
```

9.6.3.1 Regular Initializers

```
RegularInitializer ::= Expression
```

9.6.3.3 Array-Size Initializers

```
ArraySizeInitializationModifier ::=
   ( UpperBoundList ) [ ArrayTypeModifiers ]
UpperBoundList::=
   Expression |
   UpperBoundList , Expression
```

9.6.3.4 Array-Element Initializers

```
ArrayElementInitializer ::= { [ VariableInitializerList ] }
VariableInitializerList ::=
   VariableInitializer |
   VariableInitializerList , VariableInitializer
VariableInitializer ::= Expression | ArrayElementInitializer
```

9.7 Properties

```
PropertyMemberDeclaration ::=
   [ Attributes ] [ PropertyModifier+ ] Property Identifier
      [ ( [ ParameterList ] ) ] [ As TypeName ] [ ImplementsClause ]
      LineTerminator
   [ PropertyAccessorDeclaration+ ]
   [ End Property LineTerminator ]
PropertyModifier ::= ProcedureModifier | Default | ReadOnly | WriteOnly
PropertyAccessorDeclaration ::=
   PropertyGetDeclaration |
   PropertySetDeclaration
```

9.7.1 Get Accessor Declarations

```
PropertyGetDeclaration ::=
   [ Attributes ] Get LineTerminator
   [ Block ]
   End Get
```

9.7.2 Set Accessor Declarations

```
PropertySetDeclaration ::=
   [ Attributes ] Set [ ( ParameterList ) ] LineTerminator
   [ Block ]
```

```
   End Set
```

10. Statements

```
Statement ::=
   LocalDeclarationStatement |
   WithStatement |
   SyncLockStatement |
   EventStatement |
   AssignmentStatement |
   InvocationStatement |
   ConditionalStatement |
   LoopStatement |
   ErrorHandlingStatement |
   BranchStatement |
   ArrayHandlingStatement
StatementTerminator ::= LineTerminator | :
```

10.1 Blocks

```
Block ::= [ LabeledLine+ ]
LabeledLine ::= [ LabelName : ] [ Statements ] LineTerminator
LabelName ::= Identifier | IntLiteral
Statements ::=
   [ Statement ] |
   Statements : [ Statement ]
```

10.2 Local Declaration Statements

```
LocalDeclarationStatement ::=
   LocalModifier LocalDeclarators StatementTerminator
LocalModifier ::= Static | Dim | Const
LocalDeclarators ::=
   LocalDeclarator |
   LocalDeclarators , LocalDeclarator
LocalDeclarator ::=
   LocalIdentifiers [ As TypeName ] |
   LocalIdentifier [ As [ New ] TypeName [ ( [ ArgumentList ] ) ] ]
      [ = VariableInitializer ]
LocalIdentifiers ::=
   LocalIdentifier |
   LocalIdentifiers , LocalIdentifier
LocalIdentifier ::= Identifier [ ArrayNameModifier ]
```

10.3 With Statement

```
WithStatement ::=
   With Expression StatementTerminator
   [ Block ]
   End With StatementTerminator
```

10.4 SyncLock Statement

```
SyncLockStatement ::=
   SyncLock Expression StatementTerminator
   [ Block ]
   End SyncLock StatementTerminator
```

10.5 Event Statements

```
EventStatement ::=
   RaiseEventStatement |
   AddHandlerStatement |
   RemoveHandlerStatement
```

10.5.1 RaiseEvent Statement

```
RaiseEventStatement ::= RaiseEvent SimpleNameExpression
   [ ( [ ArgumentList ] ) ] StatementTerminator
```

10.5.2 AddHandler and RemoveHandler Statements

```
AddHandlerStatement ::=
   AddHandler Expression , Expression StatementTerminator
RemoveHandlerStatement ::=
   RemoveHandler Expression , Expression StatementTerminator
```

10.6 Assignment Statements

```
AssignmentStatement ::=
   RegularAssignmentStatement |
   CompoundAssignmentStatement |
   MidAssignmentStatement
```

10.6.1 Regular Assignment Statements

```
RegularAssignmentStatement ::=
   Expression = Expression StatementTerminator
```

10.6.2 Compound Assignment Statements

```
CompoundAssignmentStatement ::=
   Expression CompoundBinaryOperator = Expression StatementTerminator
CompoundBinaryOperator ::= ^ | * | / | \ | + | - | & | < < | > >
```

10.6.3 Mid Assignment Statement

```
MidAssignmentStatement ::=
   Mid [ $ ] ( Expression , Expression [ , Expression ] ) =
      Expression StatementTerminator
```

10.7 Invocation Statements

```
InvocationStatement ::= [ Call ] InvocationExpression StatementTerminator
```

10.8 Conditional Statements

```
ConditionalStatement ::= IfStatement | SelectStatement
```

10.8.1 If...Then...Else Statements

```
IfStatement ::= BlockIfStatement | LineIfThenStatement
BlockIfStatement ::=
   If Expression [ Then ] StatementTerminator
   [ Block ]
   [ ElseIfStatement+ ]
   [ ElseStatement ]
   End If StatementTerminator
ElseIfStatement ::=
   ElseIf Expression [ Then ] StatementTerminator
   [ Block ]
ElseStatement ::=
   Else StatementTerminator
   [ Block ]
LineIfThenStatement ::=
   If Expression Then [Statements ] [ Else Statements ]
      StatementTerminator
```

10.8.2 Select...Case Statements

```
SelectStatement ::=
   Select [ Case ] Expression StatementTerminator
   [ CaseStatement+ ]
   [ CaseElseStatement ]
   End Select StatementTerminator
CaseStatement ::=
   Case CaseClauses StatementTerminator
   [ Block ]
CaseClauses ::=
   CaseClause |
   CaseClauses , CaseClause
CaseClause ::=
   [ Is ] ComparisonOperator Expression |
   Expression [ To Expression ]
ComparisonOperator ::= = | < > | < | > | = > | = <
CaseElseStatement ::=
   Case Else StatementTerminator
   [ Block ]
```

10.9 Loop Statements

```
LoopStatement ::=
   WhileStatement |
   DoLoopStatement |
   ForStatement |
   ForEachStatement
```

10.9.1 While...End While and Do...Loop Statements

```
WhileStatement ::=
   While Expression StatementTerminator
   [ Block ]
   End While StatementTerminator
DoLoopStatement ::=
   Do [ WhileOrUntil Expression ] StatementTerminator
   [ Block ]
   Loop [ WhileOrUntil Expression ] StatementTerminator
WhileOrUntil ::= While | Until
```

10.9.2 For...Next Statements

```
ForStatement ::=
   For LoopControlVariable = Expression To Expression [ Step Expression ]
      StatementTerminator
   [ Block ]
   Next [ NextExpressionList ] StatementTerminator
LoopControlVariable ::=
   Identifier As TypeName |
   Expression
NextExpressionList ::=
   Expression |
   NextExpressionList , Expression
```

10.9.3 For Each...Next Statements

```
ForEachStatement ::=
   For Each LoopControlVariable In Expression StatementTerminator
   [ Block ]
   Next [Expression ] StatementTerminator
```

10.10 Exception-Handling Statements

```
ErrorHandlingStatement ::=
   StructuredErrorStatement |
   UnstructuredErrorStatement
```

10.10.1 Structured Exception-Handling Statements

```
StructuredErrorStatement ::=
   ThrowStatement |
   TryStatement
TryStatement ::=
   Try StatementTerminator
   [ Block ]
   [ CatchStatement+ ]
   [ FinallyStatement ]
   End Try StatementTerminator
```

10.10.1.1 Finally Blocks

```
FinallyStatement ::=
   Finally StatementTerminator
   [ Block ]
```

10.10.1.2 Catch Blocks

```
CatchStatement ::=
   Catch [ Identifier As TypeName ] [ When BooleanExpression ]
      StatementTerminator
   [ Block ]
```

10.10.1.3 Throw Statement

```
ThrowStatement ::= Throw [ Expression ] StatementTerminator
```

10.10.2 Unstructured Exception-Handling Statements

```
UnstructuredErrorStatement ::=
   ErrorStatement |
   OnErrorStatement |
   ResumeStatement
```

10.10.2.1 Error Statement

```
ErrorStatement ::= Error Expression StatementTerminator
```

10.10.2.2 On Error Statement

```
OnErrorStatement ::= On Error ErrorClause StatementTerminator
ErrorClause ::=
   GoTo - 1 |
   GoTo 0 |
   GotoStatement |
   Resume Next
```

10.10.2.3 Resume Statement

```
ResumeStatement ::= Resume [ ResumeClause ] StatementTerminator
ResumeClause ::= Next | LabelName
```

10.11 Branch Statements

```
BranchStatement ::=
   GotoStatement |
   ExitStatement |
   StopStatement |
   EndStatement |
   ReturnStatement
GotoStatement ::= GoTo LabelName StatementTerminator
ExitStatement ::= Exit ExitKind StatementTerminator
ExitKind ::= Do | For | While | Select | Sub | Function | Property | Try
StopStatement ::= Stop StatementTerminator
EndStatement ::= End StatementTerminator
ReturnStatement ::= Return [ Expression ]
```

10.12 Array-Handling Statements

```
ArrayHandlingStatement ::=
   RedimStatement |
   EraseStatement
```

10.12.1 ReDim Statement

```
RedimStatement ::= ReDim [ Preserve ] RedimClauses StatementTerminator
RedimClauses ::=
   RedimClause |
   RedimClauses , RedimClause
RedimClause ::= Expression ArraySizeInitializationModifier
```

10.12.2 Erase Statement

```
EraseStatement ::= Erase EraseExpressions StatementTerminator
EraseExpressions ::=
   Expression |
   EraseExpressions , Expression
```

11. Expressions

```
Expression ::=
   SimpleExpression |
   TypeExpression |
   MemberAccessExpression |
   DictionaryAccessExpression |
   IndexExpression |
   NewExpression |
   CastExpression |
   OperatorExpression
```

11.2 Constant Expressions

```
ConstantExpression ::= Expression
```

11.4 Simple Expressions

```
SimpleExpression ::=
   LiteralExpression |
   ParenthesizedExpression |
   InstanceExpression |
   SimpleNameExpression |
   AddressOfExpression
```

11.4.1 Literal Expressions

```
LiteralExpression ::= Literal
```

11.4.2 Parenthesized Expressions

```
ParenthesizedExpression ::= ( Expression )
```

11.4.3 Instance Expressions

```
InstanceExpression ::= Me | MyClass | MyBase
```

11.4.4 Simple Name Expressions

```
SimpleNameExpression ::= Identifier
```

11.4.5 AddressOf Expressions

```
AddressOfExpression ::= AddressOf Expression
```

11.5 Type Expressions

```
TypeExpression ::=
   GetTypeExpression |
   TypeOfIsExpression |
   IsExpression
```

11.5.1 GetType Expressions

```
GetTypeExpression ::= GetType ( TypeName )
```

11.5.2 TypeOf...Is Expressions

```
TypeOfIsExpression ::= TypeOf Expression Is TypeName
```

11.5.3 Is Expressions

```
IsExpression ::= Expression Is Expression
```

11.6 Member Access Expressions

```
MemberAccessExpression ::=
   [ [ MemberAccessBase ] . ] IdentifierOrKeyword
MemberAccessBase ::= Expression | BuiltInTypeName
```

11.7 Dictionary Member Access

```
DictionaryAccessExpression ::= [ Expression ] ! IdentifierOrKeyword
```

11.8 Invocation Expressions

```
InvocationExpression ::= Expression [ ( [ ArgumentList ] ) ]
ArgumentList ::=
   PositionalArgumentList , NamedArgumentList |
   PositionalArgumentList |
   NamedArgumentList
PositionalArgumentList ::=
   Expression |
   PositionalArgumentList , [ Expression ]
NamedArgumentList ::=
   IdentifierOrKeyword : = Expression |
   NamedArgumentList , IdentifierOrKeyword : = Expression
```

11.9 Index Expressions

```
IndexExpression ::= Expression ( [ ArgumentList ] )
```

11.10 New Expressions

```
NewExpression ::=
   ObjectCreationExpression |
```

```
   ArrayCreationExpression |
   DelegateCreationExpression
```

11.10.1 Object-Creation Expressions

```
ObjectCreationExpression ::= New TypeName [ ( [ ArgumentList ] ) ]
```

11.10.2 Array-Creation Expressions

```
ArrayCreationExpression ::=
   New TypeName [ ArraySizeInitializationModifier ]
      ArrayElementInitializer
```

11.10.3 Delegate-Creation Expressions

```
DelegateCreationExpression ::= New TypeName ( Expression )
```

11.11 Cast Expressions

```
CastExpression ::=
   DirectCast ( Expression , TypeName ) |
   CType ( Expression , TypeName ) |
   CastTarget ( Expression )
CastTarget ::=
   CBool | CByte | CChar | CDate | CDec | CDbl |
   CInt | CLng | CObj | CShort | CSng | CStr
```

11.12 Operator Expressions

```
OperatorExpression ::=
   ArithmeticOperatorExpression |
   RelationalOperatorExpression |
   LikeOperatorExpression |
   ConcatenationOperatorExpression |
   ShortCircuitLogicalOperatorExpression |
   LogicalOperatorExpression |
   ShiftOperatorExpression
```

11.13 Arithmetic Operators

```
ArithmeticOperatorExpression ::=
   UnaryPlusExpression |
   UnaryMinusExpression |
   AdditionOperatorExpression |
   SubtractionOperatorExpression |
   MultiplicationOperatorExpression |
   DivisionOperatorExpression |
   ModuloOperatorExpression |
   ExponentOperatorExpression
```

11.13.1 Unary Plus Operator

```
UnaryPlusExpression ::= + Expression
```

11.13.2 Unary Minus Operator

```
UnaryMinusExpression ::= - Expression
```

11.13.3 Addition Operator

```
AdditionOperatorExpression ::= Expression + Expression
```

11.13.4 Subtraction Operator

```
SubtractionOperatorExpression ::= Expression - Expression
```

11.13.5 Multiplication Operator

```
MultiplicationOperatorExpression ::= Expression * Expression
```

11.13.6 Division Operators

```
DivisionOperatorExpression ::=
   FPDivisionOperatorExpression |
   IntegerDivisionOperatorExpression
FPDivisionOperatorExpression ::= Expression / Expression
IntegerDivisionOperatorExpression ::= Expression \ Expression
```

11.13.7 Mod Operator

```
ModuloOperatorExpression ::= Expression Mod Expression
```

11.13.8 Exponentiation Operator

```
ExponentOperatorExpression ::= Expression ^ Expression
```

11.14 Relational Operators

```
RelationalOperatorExpression ::=
   Expression = Expression |
   Expression < > Expression |
   Expression < Expression |
   Expression > Expression |
   Expression < = Expression |
   Expression > = Expression
```

11.15 Like Operator

```
LikeOperatorExpression ::= Expression Like Expression
```

11.16 Concatenation Operator

```
ConcatenationOperatorExpression ::= Expression & Expression
```

11.17 Logical Operators

```
LogicalOperatorExpression ::=
   Not Expression |
   Expression And Expression |
   Expression Or Expression |
   Expression Xor Expression
```

11.17.1 Short-circuiting Logical Operators

```
ShortCircuitLogicalOperatorExpression ::=
   Expression AndAlso Expression |
   Expression OrElse Expression
```

11.18 Shift Operators

```
ShiftOperatorExpression ::=
   Expression << Expression |
   Expression >> Expression
```

Visual C# Language

Microsoft C# (pronounced C sharp) is a new programming language designed for building a wide range of enterprise applications that run on the .NET Framework. An evolution of Microsoft C and Microsoft C++, C# is simple, modern, type safe, and object oriented. C# code is compiled as managed code, which means it benefits from the services of the common language runtime. These services include language interoperability, garbage collection, enhanced security, and improved versioning support.

C# is introduced as Visual C# in the Visual Studio .NET suite. Support for Visual C# includes project templates, designers, property pages, code wizards, an object model, and other features of the development environment. The library for Visual C# programming is the .NET Framework.

In This Section

C# Language Tour
: Discusses features and reasons for using C#. Includes a Documentation Map and a Comparison Between C++ and C#.

C# Compiler Options
: A reference to the C# compiler options and build errors. Includes information on Building from the Command Line.

Visual C# Code Wizards
: Wizards you can use to add files, classes, methods, properties, and other items to your projects.

C# Language Specification
: The syntax, semantics, and design of the language.

C# Programmer's Reference
: Keywords, types, operators, attributes, and preprocessor directives. Includes the C# Tutorials and topics on language features.

Related Sections

Visual Basic and Visual C#
: Designing, developing, debugging, testing, and deploying applications using Visual Basic or Visual C#.

Visual C# Samples
: Sample source code demonstrating how to program using Visual C#.

C# Language Tour

This section provides a quick tour of the C# language. It also introduces a simple C# program and examines its structure and important features.

The following topics are discussed:

- What's New in Visual C# .NET 2003
- Quick Survey of C# Features
- Why Use C#
- The C# Version of Hello World!
- General Structure of a C# Program
- Using Fully Qualified Names
- Creating and Using C# DLLs
- Comparison Between C++ and C#

For a list of entry points into the documentation, see Documentation Map.

See Also

Visual C# Language

What's New in Visual C# .NET 2003

Microsoft Visual C# .NET 2003 includes new features in the following areas:

- Compiler and Language
- Development Environment
- IntelliSense
- Code Examples

Compiler and Language

The compiler contains the following enhancements for this release:

- #line hidden is now added as an option to the **#line** directive. See C# Preprocessor Directives.
- The compiler now recognizes the `/** */` delimiters for documentation tags. See Delimiters for Documentation Tags for more information.

Development Environment

The development environment contains the following enhancements for this release:

- /nowarn is now available in the property pages of a Visual C# project. See Build, Configuration Properties for more information.
- /nostdlib is now available in the property pages of a Visual C# project. See Advanced, Configuration Properties, <Projectname> Property Pages Dialog Box for more information.
- Build events are now available for Visual C# projects. See Build Events, Common Properties, <Projectname> Property Pages Dialog Box for more information.
- The General, Common Properties property page now includes the Supported Runtimes property.

IntelliSense

IntelliSense contains the following enhancements for this release:

- Automatic hook up of events in the Code Editor. See Automatically Hooking Up to Events for more information.
- Interface implementation assistance in the Code Editor. See Implementing Interfaces for more information.
- Recognition of your most frequently used members for convenient syntax completion. See IntelliSense for Most Frequently Used Members for more information.
- Assistance with overriding base class members in the Code Editor. See IntelliSense for Override for more information.
- Accessibility from the Immediate window of the debugger. See IntelliSense in Immediate Windows for more information.

Code Examples

New code example topics have been added to Visual C# .NET documentation. The topics are designed to present concise, real-world examples of common tasks. The following example categories are shipped with this release:

- General Language Example Topics
- Graphics Programming Example Topics
- Web Applications Example Topics
- Windows Forms Example Topics

See Also

What's New in Visual Basic and Visual C#

Quick Survey of C# Features

C# is a simple but powerful programming language intended for writing enterprise applications.

The C# language is an evolution of C and C++. It uses many C++ features in the areas of statements, expressions, and operators.

C# introduces considerable improvement and innovations in areas such as type safety, versioning, events, and garbage collection.

C# provides access to the common API styles: .NET Framework, COM, Automation, and C-style APIs. It also supports unsafe mode, where you can use pointers to manipulate memory that is not under the control of the garbage collector.

See Also

C# Language Tour

Why Use C#

C# is an elegant, simple, type-safe, object-oriented language that allows enterprise programmers to build a breadth of applications.

C# also gives you the capability to build durable system-level components by virtue of the following features:

- Full COM/Platform support for existing code integration.
- Robustness through garbage collection and type safety.
- Security provided through intrinsic code trust mechanisms.
- Full support of extensible metadata concepts.

You can also interoperate with other languages, across platforms, with legacy data, by virtue of the following features:

- Full interoperability support through COM+ 1.0 and .NET Framework services with tight library-based access.
- XML support for wWeb-based component interaction.
- Versioning to provide ease of administration and deployment.

See Also

C# Language Tour

The C# Version of Hello World!

The following console program is the C# version of the traditional "Hello World!" program, which displays the string `Hello World!`.

```
// A "Hello World!" program in C#
class Hello
{
   static void Main()
   {
      System.Console.WriteLine("Hello World!");
   }
}
```

The following are important points of this program:

- Comments
- The Main Method
- Input and Output
- Compilation and Execution

Comments

The first line contains a comment:

```
// A "Hello World!" program in C#
```

The characters `//` convert the rest of the line to a comment. You can also comment a block of text by placing it between the characters `/*` and `*/`, for example:

```
/* A "Hello World!" program in C#.
This program displays the string "Hello World!" on the screen. */
```

The Main Method

The C# program must contain a `Main` method, in which control starts and ends. The `Main` method is where you create objects and execute other methods.

The `Main` method is a static method that resides inside a class or a struct. In the "Hello World!" example, it resides inside the `Hello` class. Declare the `Main` method in one of the following ways:

- It can return **void**:

  ```
  static void Main()
  {
  ...
  }
  ```

- It can also return an **int**:

  ```
  static int Main()
  {
     ...
     return 0;
  }
  ```

- With both of the return types, it can take arguments:

  ```
  static int Main(string[] args)
  {
  ```

```
    ...
    return 0;
}
```

-or-

```
static void Main(string[] args)
{
    ...
}
```

The parameter of the `Main` method is a `string` array that represents the command-line arguments used to invoke the program. Notice that, unlike C++, this array does not include the name of the executable (exe) file.

For more information on using command-line arguments, see the example in Main and Creating and Using C# DLLs.

Input and Output

C# programs generally use the input/output services provided by run-time library of the .NET Framework. The statement: `System.Console.WriteLine("Hello World!");` uses the **WriteLine** method, one of the output methods of the **Console** class in the run-time library. It displays its string parameter on the standard output stream followed by a new line. Other **Console** methods are used for different input and output operations. If you include the `using System;` statement at the beginning of the program, you can directly use the **System** classes and methods without fully qualifying them. For example:

```
Console.WriteLine("Hello World!");
```

For more information on input/output methods, see the Reference section of the .NET Framework SDK.

Compilation and Execution

You can compile the "Hello World!" program either by creating a project in the Visual Studio IDE, or by using the command line.

To compile the program from the command line:

- Create the source file using any text editor and save it using a name such as `Hello.cs`. C# source code files use the extension `.cs`.
- To invoke the compiler, enter the command:

  ```
  csc Hello.cs
  ```

  If your program does not contain any compilation errors, a `Hello.exe` file will be created.

- To run the program, enter the command:

  ```
  Hello
  ```

For more information on the C# compiler and its options, see C# Compiler Options.

See Also

C# Language Tour

General Structure of a C# Program

C# programs can consist of one or more files. Each file can contain one or more namespaces. A namespace can contain types such as classes, structs, interfaces, enumerations, and delegates, in addition to other namespaces. The following is the skeleton of a C# program that contains all of these elements.

```
// A skeleton of a C# program
using System;
namespace MyNamespace1
{
   class MyClass1
   {
   }
   struct MyStruct
   {
   }
   interface IMyInterface
   {
   }
   delegate int MyDelegate();
   enum MyEnum
   {
   }
   namespace MyNamespace2
   {
   }
   class MyClass2
   {
      public static void Main(string[] args)
      {
      }
   }
}
```

See Also

C# Language Tour

Using Fully Qualified Names

Namespaces and types have unique names, which are described by fully qualified names that indicate a logical hierarchy. For example, `A.B` is the name of the namespace or type `B` nested in the namespace or type `A`.

In the following code segment, there are nested classes and namespaces. The fully qualified name is indicated as a comment following each entity.

```
namespace N1     // N1
{
   class C1      // N1.C1
   {
      class C2   // N1.C1.C2
      {
      }
   }
   namespace N2  // N1.N2
   {
      class C2   // N1.N2.C2
      {
      }
   }
}
```

In the preceding code segment:

- The namespace `N1` is a member of the global namespace. Its fully qualified name is `N1`.
- The namespace `N2` is a member of `N1`. Its fully qualified name is `N1.N2`.
- The class `C1` is a member of the `N1`. Its fully qualified name is `N1.C1`.
- The class name `C2` is used twice in this code. However, the fully qualified names are unique. The first one is declared inside `C1`; thus its fully qualified name is: `N1.C1.C2`. The second is declared inside a namespace `N2`; thus, its fully qualified name is `N1.N2.C2`.

Using the preceding code segment, you can add a new class member `C3` to the namespace `N1.N2` as follows:

```
namespace N1.N2
{
   class C3   // N1.N2.C3
   {
   }
}
```

For more information on namespaces, see Namespaces.

See Also

C# Language Tour

Creating and Using C# DLLs

A dynamic linking library (DLL) is linked to your program at run time. To demonstrate building and using a DLL, consider the following scenario:

- `MyLibrary.DLL`: The library file that contains the methods to be called at run time. In this example, the DLL contains two methods, `Add` and `Multiply`.
- `Add.cs`: The source file that contains the method `Add(long i, long j)`. It returns the sum of its parameters. The class `AddClass` that contains the method `Add` is a member of the namespace `MyMethods`.
- `Mult.cs`: The source code that contains the method `Multiply(long x, long y)`. It returns the product of its parameters. The class `MultiplyClass` that contains the method `Multiply` is also a member of the namespace `MyMethods`.
- `MyClient.cs`: The file that contains the `Main` method. It uses the methods in the DLL file to calculate the sum and the product of the run-time arguments.

Source files

File: Add.cs

```
// Add two numbers
using System;
namespace MyMethods
{
   public class AddClass
   {
      public static long Add(long i, long j)
      {
         return(i+j);
      }
   }
}
```

File: Mult.cs

```
// Multiply two numbers
using System;
namespace MyMethods
{
   public class MultiplyClass
   {
      public static long Multiply(long x, long y)
      {
         return (x*y);
      }
   }
}
```

File: MyClient.cs

```
// Calling methods from a DLL file
using System;
using MyMethods;
class MyClient
{
   public static void Main(string[] args)
   {
      Console.WriteLine("Calling methods from MyLibrary.DLL:");
      if (args.Length != 2)
      {
         Console.WriteLine("Usage: MyClient <num1> <num2>");
```

```
            return;
        }
        long num1 = long.Parse(args[0]);
        long num2 = long.Parse(args[1]);
        long sum = AddClass.Add(num1, num2);
        long product = MultiplyClass.Multiply(num1, num2);
        Console.WriteLine("The sum of {0} and {1} is {2}",
                          num1, num2, sum);
        Console.WriteLine("The product of {0} and {1} is {2}",
                          num1, num2, product);
    }
}
```

This file contains the algorithm that uses the DLL methods, `Add` and `Multiply`. It starts with parsing the arguments entered from the command line, `num1` and `num2`. Then it calculates the sum by using the `Add` method on the `AddClass` class, and the product by using the `Multiply` method on the `MultiplyClass` class.

Notice that the **using** directive at the beginning of the file allows you to use the unqualified class names to reference the DLL methods at compile time, for example:

```
MultiplyClass.Multiply(num1, num2);
```

Otherwise, you have to use the fully qualified names, for example:

```
MyMethods.MultiplyClass.Multiply(num1, num2);
```

Compilation

To build the file `MyLibrary.DLL`, compile the two files `Add.cs` and `Mult.cs` using the following command line:

```
csc /target:library /out:MyLibrary.DLL Add.cs Mult.cs
```

The /target:library compiler option tells the compiler to output a DLL instead of an EXE file. The /out compiler option followed by a file name is used to specify the DLL file name. Otherwise, the compiler uses the first file (`Add.cs`) as the name of the DLL.

To build the executable file, `MyClient.exe`, use the following command line:

```
csc /out:MyClient.exe /reference:MyLibrary.DLL MyClient.cs
```

The `/out` compiler option tells the compiler to output an EXE file and specifies the name of the output file (`MyClient.exe`). This compiler option is optional. The /reference compiler option specifies the DLL file(s) that this program uses.

Execution

To run the program, enter the name of the EXE file, followed by two numbers, for example:

```
MyClient 1234 5678
```

Output

```
Calling methods from MyLibrary.DLL:
The sum of 1234 and 5678 is 6912
The product of 1234 and 5678 is 7006652
```

See Also

C# Language Tour | Libraries Tutorial

Comparison Between C++ and C#

The following table contains important comparisons between C++ and C# features. If you are a C++ programmer, this table will give you the most important differences between the two languages at a glance. For details, use the link to browse to the detailed topic.

> **Note** C++ and C# projects are derived from different project models. For information on the differences between C++ and C# projects, see Item Management in Projects and Using Solution Explorer.

Feature	Refer to the topic
Inheritance: A class can inherit implementation from one base class only. Also a class or an interface can implement multiple interfaces.	class interface
Arrays: The syntax of declaring C# arrays is different from that of C++ arrays. The tokens "[]" appear following the array type in C#.	Arrays
The **bool** type: There is no conversion between the **bool** type and other types (specifically **int**).	bool
The **long** type: In C#, the **long** data type is 64 bits, while in C++, it is 32 bits.	long
The **struct** type: In C#, classes and structs are semantically different. A **struct** is a value type, while a **class** is a reference type.	struct class
The **switch** statement: Unlike the C++ **switch** statement, C# does not support fall through from one case label to another.	switch
The **delegate** type: Delegates are roughly similar to function pointers in C++, but they are type-safe and secure.	delegate
Calling the overridden base class members from derived classes.	base See also the examples for override
Using the **new** modifier to explicitly hide an inherited member.	new
Declaring override methods requires the **override** keyword.	override
Preprocessor directives are used for conditional compilation. No header files are used in C#.	C# Preprocessor Directives
Exception handling: Using the **finally** statement.	try-finally try-catch-finally
C# operators: C# supports additional operators such as **is** and **typeof**. It also introduces different functionality of some logical operators.	& Operator | Operator ^ Operator is typeof
Use of the **extern** keyword.	extern
Use of the **static** keyword.	static
An alternative way to the C++ initialization list in constructing the base class.	See the examples for virtual
The general structure of the C# program: namespaces, classes, structs, delegates, and enumerations.	General Structure of a C# Program
The **Main** method is declared differently from the **main** function in C++. Also, compare the usage of the command-line arguments for each language.	Main
Method parameters: C# supports **ref** and **out** parameters, which are used instead of pointers in passing parameters by reference.	ref out
Pointers are allowed in C# but only in unsafe mode.	unsafe

Overloading operators is performed differently in C#.	C# Operators
Strings: C# strings are different from C++ strings.	string
The **foreach** keyword allows you to iterate through arrays and collections.	foreach, in
No global methods or variables in C#: Methods and variables must be contained within a type declaration (such as **class** or **struct**).	General Structure of a C# Program
No header files or #include directives in C#: The **using** directive is used to reference types in other namespaces without fully qualifying the type names.	using
Local variables in C# cannot be used before they are initialized.	5. Variables
Destructors: In C#, you don't have control over when a destructor is called because destructors are called automatically by the garbage collector.	Destructors
Constructors: Similar to C++, if you don't provide a class constructor in C#, a default constructor is automatically generated for you. The default constructor initializes all the fields to their default values.	Instance Constructors Default Values Table
C# does not support bit fields.	C++ Bit Fields
C# input/output services and formatting rely on the run-time library of the .NET Framework.	C# Language Tour Formatting Numeric Results Table
In C#, method parameters cannot have default values. Use method overloads if you want to achieve the same effect.	Compiler Error CS0241

For comparisons between C# and other programming languages, see the Language Equivalents documentation.

See Also

C# Language Tour | Item Management in Projects | Using Solution Explorer

Documentation Map

Visual Studio .NET and the MSDN library include the following documentation to help you write Visual C# applications.

For information on	See
Features, quick tour, and reasons for using C#	C# Language Tour
Sample programs	Visual C# Samples
Tutorials	C# Tutorials
Compiler options and build errors	C# Compiler Options
The syntax, semantics, and design of the C# language	C# Language Specification
C++ compared to C#	Comparison Between C++ and C#
Keywords, types, operators, attributes, and preprocessor directives	C# Programmer's Reference
Arrays, properties, indexers, and other language features	C# Language Features
Documenting your code using XML	XML Documentation
The kinds of projects you can create	Visual Basic and Visual C# Projects
Project settings	Setting Visual C# Project Properties
Code assistants	Visual C# Code Wizards
Writing Windows applications	Creating Windows Applications
Writing Web applications	Creating Web Applications and Services
Components, including those you use on servers	Programming with Components
Data access	Accessing Data
Debugging code	Debugging
Distributing a finished program	Deploying Applications and Components
Microsoft .NET Framework SDK	Inside the .NET Framework

See Also

Visual C# Language

C# Compiler Options

The compiler produces executable (.exe) files, dynamic-link libraries (.dll), or code modules (.netmodule).

Every compiler option is available in two forms: **-***option* and **/***option*. The documentation only shows the **/***option* form.

In This Section

Building from the Command Line
Information about building a Visual C# application from the command line.
C# Compiler Options Listed by Category
A categorical listing of the compiler options.
C# Compiler Options Listed Alphabetically
An alphabetical listing of the compiler options.
Compiler Errors
A reference to the errors generated by the C# compiler.

Related Sections

Setting Visual C# Project Properties
Setting properties that govern how your project is compiled, built, and debugged. Includes information about custom build steps in Visual C# projects.
Default and Custom Builds
Information on build types and configurations.
Preparing and Managing Builds
Procedures for building within the Visual Studio development environment.

Building from the Command Line

The C# compiler can be invoked at the command line by typing the name of its executable file (csc.exe) on the command line. You may need to adjust your path if you want csc.exe to be invoked from any subdirectory on your computer.

This topic provides details on the following:

- Running VCVARS32.BAT
- Differences Between C# Compiler and C++ Compiler Output
- Rules for Command-Line Syntax
- Sample Command Lines

Running VCVARS32.BAT

vcvars32.bat sets the appropriate environment variables to enable command line builds.

To run VCVARS32.BAT

1. At the command prompt, change to the \bin subdirectory of your installation.
2. Run VCVARS32.bat by typing `VCVARS32`.

> **Caution** VCVARS32.bat can vary from machine to machine. Do not replace a missing or damaged VCVARS32.bat file with a VCVARS32.bat from another machine. Rerun setup to replace the missing file.

For more information about vcvars32.bat, see the following Knowledge Base article:

- Q248802 : Vcvars32.bat Generates Out of Environment Message

If the current version of Visual Studio is installed on a computer that also has a previous version of Visual Studio, you should not run vcvars32.bat from different versions in the same command window.

Differences Between C# Compiler and C++ Compiler Output

There are no object (.obj) files created as a result of invoking the C# compiler; output files are created directly. As a consequence of this, the C# compiler does not need a linker.

Rules for Command-Line Syntax

The C# compiler code uses the following rules when interpreting arguments given on the operating system command line:

- Arguments are delimited by white space, which is either a space or a tab.
- The caret character (^) is not recognized as an escape character or delimiter. The character is handled completely by the command-line parser in the operating system before being passed to the argv array in the program.
- A string surrounded by double quotation marks ("string") is interpreted as a single argument, regardless of white space contained within. A quoted string can be embedded in an argument.
- A double quotation mark preceded by a backslash (\") is interpreted as a literal double quotation mark character (").
- Backslashes are interpreted literally, unless they immediately precede a double quotation mark.
- If an even number of backslashes is followed by a double quotation mark, one backslash is placed in the argv array for every pair of backslashes, and the double quotation mark is interpreted as a string delimiter.
- If an odd number of backslashes is followed by a double quotation mark, one backslash is placed in the argv array for every pair of backslashes, and the double quotation mark is "escaped" by the remaining backslash, causing a literal double quotation mark (") to be placed in argv.

Sample Command Lines

- Compiles File.cs producing File.exe:

```
csc File.cs
```

- Compiles File.cs producing File.dll:

```
csc /target:library File.cs
```

- Compiles File.cs and creates My.exe:

```
csc /out:My.exe File.cs
```

- Compiles all of the C# files in the current directory, with optimizations on and defines the DEBUG symbol. The output is File2.exe:

```
csc /define:DEBUG /optimize /out:File2.exe *.cs
```

- Compiles all of the C# files in the current directory producing a debug version of File2.dll. No logo and no warnings are displayed:

```
csc /target:library /out:File2.dll /warn:0 /nologo /debug *.cs
```

- Compiles all of the C# files in the current directory to Something.xyz (a DLL):

```
csc /target:library /out:Something.xyz *.cs
```

See Also

C# Compiler Options | C# Compiler Options Listed Alphabetically | C# Compiler Options Listed by Category

C# Compiler Options Listed by Category

The following compiler options are sorted by category. For an alphabetical list, see C# Compiler Options Listed Alphabetically.

Optimization

Option	Purpose
/filealign	Specify the size of sections in the output file.
/optimize	Enable/disable optimizations.

Output Files

Option	Purpose
/doc	Process documentation comments to an XML file.
/out	Specify output file.
/target	Specify the format of the output file using one of four options: /target:exe /target:library /target:module /target:winexe

.NET Framework Assemblies

Option	Purpose
/addmodule	Specify one or more modules to be part of this assembly.
/lib	Specify the location of assemblies referenced by means of /reference.
/nostdlib	Do not import standard library (mscorlib.dll).
/reference	Import metadata from a file that contains an assembly.

Debugging/Error Checking

Option	Purpose
/bugreport	Create a file that contains information that makes it easy to report a bug.
/checked	Specify whether integer arithmetic that overflows the bounds of the data type will cause an exception at run time.
/debug	Emit debugging information.
/fullpaths	Specify the absolute path to the file in compiler output.
/nowarn	Suppress the compiler's ability to generate specified warnings.
/warn	Set warning level.
/warnaserror	Promote warnings to errors.

Preprocessor

Option	Purpose
/define	Define preprocessor symbols.

Resources

Option	Purpose
/linkresource	Create a link to a managed resource.
/resource	Embed a .NET Framework resource into the output file.
/win32icon	Insert an .ico file into the output file.
/win32res	Insert a Win32 resource into the output file.

Miscellaneous

Option	Purpose

@	Specify a response file.
/?	List compiler options to stdout.
/baseaddress	Specify the preferred base address at which to load a DLL.
/codepage	Specify the code page to use for all source code files in the compilation .
/help	List compiler options to stdout.
/incremental	Enable incremental compilation of source code files.
/main	Specify the location of the **Main** method.
/noconfig	Do not compile with csc.rsp.
/nologo	Suppress compiler banner information.
/recurse	Search subdirectories for source files to compile.
/unsafe	Compile code that uses the unsafe keyword.
/utf8output	Display compiler output using UTF-8 encoding.

See Also

C# Compiler Options | C# Compiler Options Listed Alphabetically | Building from the Command Line

C# Compiler Options Listed Alphabetically

The following compiler options are sorted alphabetically. For a categorical list, see C# Compiler Options Listed By Category.

Option	Purpose
@	Specify a response file.
/?	List compiler options to stdout.
/addmodule	Specify one or more modules to be part of this assembly.
/baseaddress	Specify the preferred base address at which to load a DLL.
/bugreport	Create a file that contains information that makes it easy to report a bug.
/checked	Specify whether integer arithmetic that overflows the bounds of the data type will cause an exception at run time.
/codepage	Specify the code page to use for all source code files in the compilation.
/debug	Emit debugging information.
/define	Define preprocessor symbols.
/doc	Process documentation comments to an XML file.
/filealign	Specify the size of sections in the output file.
/fullpaths	Specify the absolute path to the file in compiler output.
/help	List compiler options to stdout.
/incremental	Enable incremental compilation of source code files.
/lib	Specify the location of assemblies referenced by means of /reference.
/linkresource	Create a link to a managed resource.
/main	Specify the location of the **Main** method.
/noconfig	Do not compile with csc.rsp.
/nologo	Suppress compiler banner information.
/nostdlib	Do not import standard library (mscorlib.dll).
/nowarn	Suppress the compiler's ability to generate specified warnings.
/optimize	Enable/disable optimizations.
/out	Specify output file.
/recurse	Search subdirectories for source files to compile.
/reference	Import metadata from a file that contains an assembly.
/resource	Embed a .NET Framework resource into the output file.
/target	Specify the format of the output file using one of four options: /target:exe /target:library /target:module /target:winexe
/unsafe	Compile code that uses the unsafe keyword.
/utf8output	Display compiler output using UTF-8 encoding.
/warn	Set warning level.
/warnaserror	Promote warnings to errors.
/win32icon	Insert an .ico file into the output file.
/win32res	Insert a Win32 resource into the output file.

See Also

C# Compiler Options | C# Compiler Options Listed by Category | Building from the Command Line

@ (Specify Response File)

```
@response_file
```

where:

response_file
A file that lists compiler options or source code files to compile.

Remarks

The @ option lets you specify a file that contains compiler options and source code files to compile. These compiler options and source code files will be processed by the compiler just as if they had been specified on the command line.

To specify more than one response file in a compilation, specify multiple response file options. For example:

```
@file1.rsp @file2.rsp
```

In a response file, multiple compiler options and source code files can appear on one line. A single compiler option specification must appear on one line (cannot span multiple lines). Response files can have comments that begin with the # symbol.

Specifying compiler options from within a response file is just like issuing those commands on the command line. See Building from the Command Line for more information.

The compiler processes the command options as they are encountered. Therefore, command line arguments can override previously listed options in response files. Conversely, options in a response file will override options listed previously on the command line or in other response files.

C# provides the csc.rsp file, which is located in the same directory as the csc.exe file. See /noconfig for more information on csc.rsp.

To set this compiler option in the Visual Studio development environment

This compiler option is unavailable in Visual Studio.

To set this compiler option programmatically

This compiler option cannot be changed programmatically.

Example

The following are a few lines from a sample response file:

```
# build the first output file
/target:exe /out:MyExe.exe source1.cs source2.cs
```

See Also

C# Compiler Options

/addmodule (Import Metadata)

```
/addmodule:file[;file2]
```

where:

file, *file2*
An output file that contains metadata. The file cannot contain an assembly manifest. To import more than one file, separate file names with either a comma or a semicolon.

Remarks

All modules added with **/addmodule** must be in the same directory as the output file at run time. That is, you can specify a module in any directory at compile time but the module must be in the application directory at run time. If the module is not in the application directory at run time, you will get a **System.TypeLoadException**.

file cannot contain an assembly. For example, if the output file was created with /target:module, its metadata can be imported with **/addmodule**.

If the output file was created with a **/target** option other than **/target:module**, its metadata cannot be imported with **/addmodule** but can be imported with /reference.

To set this compiler option in the Visual Studio development environment

This compiler option is unavailable in Visual Studio; a project cannot reference a module.

To set this compiler option programmatically

This compiler option cannot be changed programmatically.

Example

Compile source file `input.cs` and add metadata from `metad1.netmodule` and `metad2.netmodule` to produce `out.exe`:

```
csc /addmodule:metad1.netmodule;metad2.netmodule /out:out.exe input.cs
```

See Also

C# Compiler Options

/baseaddress (Specify Base Address of DLL)

```
/baseaddress:address
```

where:

address
The base address for the DLL. This address can be specified as a decimal, hexadecimal, or octal number.

Remarks

The **/baseaddress** option lets you specify the preferred base address at which to load a DLL. The default base address for a DLL is set by the .NET Framework common language runtime.

Be aware that the lower-order word in this address will be rounded. For example, if you specify 0x11110001, it will be rounded to 0x11110000.

To complete the signing process for a DLL, use –R of SN.EXE.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Setting Visual C# Project Properties.
2. Click the **Configuration Properties** folder.
3. Click the **Advanced** property page.
4. Modify the **Base Address** property.

To set this compiler option programmatically

See BaseAddress Property.

See Also

C# Compiler Options

/bugreport (Report a Problem)

```
/bugreport:file
```

where:

file
The name of the file that you want to contain your bug report.

Remarks

The **/bugreport** option will cause the following information to be placed in *file*:

- A copy of all source code files in the compilation.
- A listing of the compiler options used in the compilation.
- Version information about your compiler, runtime, and operating system.
- Referenced assemblies and modules (saved as hexadecimal digits), except assemblies that ship with the .NET Framework SDK.
- Compiler output, if any.
- A description of the problem, which you will be prompted for.
- A description of how you think the problem should be fixed, which you will be prompted for.

Because a copy of all source code files will be placed in *file*, you may want to reproduce the (suspected) code defect in the shortest possible program.

To set this compiler option in the Visual Studio development environment

This compiler option is unavailable in Visual Studio.

To set this compiler option programmatically

This compiler option cannot be changed programmatically.

Example

Compile `t2.cs` and put all bug reporting information in the file `problem.txt`:

```
csc /bugreport:problem.txt t2.cs
```

See Also

C# Compiler Options

/checked (Check Integer Arithmetic)

```
/checked[+ | -]
```

Remarks

The **/checked** option specifies whether an integer arithmetic statement that is not in the scope of the checked or unchecked keywords and that results in a value outside the range of the data type shall cause a run-time exception.

An integer arithmetic statement that is in the scope of the **checked** or **unchecked** keywords is not subject to the effect of the **/checked** option.

If an integer arithmetic statement that is not in the scope of the **checked** or **unchecked** keywords results in a value outside the range of the data type, and **/checked+** (**/checked**) is used in the compilation, then that statement will cause an exception at run time. If **/checked-** is used in the compilation, then that statement will not cause an exception at run time.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Setting Visual C# Project Properties.
2. Click the **Configuration Properties** folder.
3. Click the **Build** property page.
4. Modify the **Check for Arithmetic Overflow/Underflow** property.

To set this compiler option programmatically

See CheckForOverflowUnderflow Property.

Example

Compile `t2.cs` and specify that an integer arithmetic statement that is not in the scope of the **checked** or **unchecked** keywords and results in a value outside the range of the data type will cause an exception at run time.

```
csc t2.cs /checked
```

See Also

C# Compiler Options

/codepage (Specify Code Page for Source Code Files)

```
/codepage:id
```

where:

id
The id of the code page to use for all source code files in the compilation.

Remarks

If you compile one or more source code files that were not created to use the default code page on your computer, you can use the **/codepage** option to specify which code page should be used. **/codepage** applies to all source code files in your compilation.

If the source code files were created with the same codepage that is in effect on your computer or if the source code files were created with UNICODE or UTF-8, you need not use **/codepage**.

See GetCPInfo for information on how to find which code pages are supported on your system.

To set this compiler option in the Visual Studio development environment

This compiler option is unavailable in Visual Studio.

To set this compiler option programmatically

This compiler option cannot be changed programmatically.

See Also

C# Compiler Options

/debug (Emit Debugging Information)

```
/debug[+ | -]
/debug:{full | pdbonly}
```

where:

+ | -
Specifying +, or just **/debug**, causes the compiler to generate debugging information and place it in a program database (.pdb file). Specifying -, which is in effect if you do not specify **/debug**, causes no debug information to be created.

full | pdbonly
Specifies the type of debugging information generated by the compiler. The full argument, which is in effect if you do not specify **/debug:pdbonly**, enables attaching a debugger to the running program. Specifying pdbonly allows source code debugging when the program is started in the debugger but will only display assembler when the running program is attached to the debugger.

Remarks

The **/debug** option causes the compiler to generate debugging information and place it in the output file(s). Use this option to create debug builds. If **/debug**, **/debug+**, or **/debug:full** is not specified, you will not be able to debug the output file of your program.

For information on how to configure the debug performance of an application, see Making an Image Easier to Debug.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Setting Visual C# Project Properties.
2. Click the **Configuration Properties** folder.
3. Click the **Build** property page.
4. Modify the **Generate Debugging Information** property.

To set this compiler option programmatically

See DebugSymbols Property.

Example

Place debugging information in output file `app.exe`:

```
csc /debug /out:app.exe test.cs
```

See Also

C# Compiler Options

/define (Preprocessor Definition)

```
/define:name[;name2]
```

where:

name, *name2*
The name of one or more symbols that you want to define.

Remarks

The **/define** option defines *name* as a symbol in your program. It has the same effect as using a #define preprocessor directive in your source file. A symbol remains defined until an #undef directive in the source file removes the definition or the compiler reaches the end of the file.

You can use symbols created by this option with #if, #else, #elif, and #endif to compile source files conditionally.

/d is the short form of **/define**.

You can define multiple symbols with **/define** by using a semicolon or comma to separate symbol names. For example:

```
/define:DEBUG;TUESDAY
```

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Setting Visual C# Project Properties.
2. Click the **Configuration Properties** folder.
3. Click the **Build** property page.
4. Modify the **Conditional Compilation Constants** property.

To set this compiler option programmatically

See DefineConstants Property.

Example

```
// preprocessor_define.cs
// compile with: /define:xx
// or uncomment the next line
// #define xx
using System;
public class Test
{
   public static void Main()
   {
      #if (xx)
         Console.WriteLine("xx exists");
      #else
         Console.WriteLine("xx does not exist");
      #endif
   }
}
```

See Also

C# Compiler Options

/doc (Process Documentation Comments)

```
/doc:file
```

where:

file
The output file for XML, which is populated with the comments in the source code files of the compilation.

Remarks

The **/doc** option allows you to place documentation comments in an XML file.

For Visual Studio .NET, only the C# compiler provides this documentation comments feature.

In source code files, documentation comments that precede the following can be processed and added to the XML file:

- Such user-defined types as a class, delegate, or interface
- Such members as a field, event, property, or method

The source code file that contains Main is output first into the XML.

To use the generated .xml file for use with the IntelliSense feature, let the file name of the .xml file be the same as the assembly you want to support and then make sure the .xml file is in the same directory as the assembly. Thus, when the assembly is referenced in the Visual Studio project, the .xml file is found as well.

/doc will be ignored in a compilation that uses /incremental; use /incremental- to ensure that *file* is up to date.

Unless you compile with /target:module, *file* will contain <assembly></assembly> tags specifying the name of the file containing the assembly manifest for the output file of the compilation.

See Recommended Tags for Documentation Comments for ways to generate documentation from comments in your code.

See the GotDotNet User Samples Search page (http://www.gotdotnet.com/userarea/keywordsrch.aspx?keyword=ansonh) for a sample of how to format an XML file.

See Commenting Code for information on how Visual Studio will display a report on the structure of your code, to include XML documentation comments.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Setting Visual C# Project Properties.
2. Click the **Configuration Properties** folder.
3. Click the **Build** property page.
4. Modify the **XML Documentation File** property.

To set this compiler option programmatically

See DocumentationFile Property.

Example

See XML Documentation Tutorial for a sample.

See Also

C# Compiler Options | XML Documentation Tutorial

/filealign (Specify Section Alignment)

```
/filealign:number
```

where:

number
A value that specifies the size of sections in the output file. Valid values are 512, 1024, 2048, 4096, and 8192. These values are in bytes.

Remarks

The **/filealign** option lets you specify the size of sections in your output file.

Each section will be aligned on a boundary that is a multiple of the **/filealign** value. There is no fixed default. If **/filealign** is not specified, the common language runtime picks a default at compile time.

By specifying the section size, you affect the size of the output file. Modifying section size may be useful for programs that will run on smaller devices.

Use DUMPBIN to see information about sections in your output file.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Setting Visual C# Project Properties.
2. Click the **Configuration Properties** folder.
3. Click the **Advanced** property page.
4. Modify the **File Alignment** property.

To set this compiler option programmatically

See FileAlignment Property.

See Also

C# Compiler Options

/fullpaths (Specify Absolute Path in Compiler Output)

```
/fullpaths
```

Remarks

By default, errors and warnings that result from compilation specify the name of the file in which an error was found. The **/fullpaths** option causes the compiler to specify the full path to the file.

To set this compiler option in the Visual Studio development environment

This compiler option is unavailable in Visual Studio.

To set this compiler option programmatically

This compiler option cannot be changed programmatically.

See Also

C# Compiler Options

/help, /? (Compiler Command-Line Help)

```
/help
/?
```

Remarks

This option sends a listing of compiler options, and a brief description of each option, to stdout. If this option is included in a compilation, no output file will be created and no compilation will take place.

To set this compiler option in the Visual Studio development environment

This compiler option is unavailable in Visual Studio.

To set this compiler option programmatically

This compiler option cannot be changed programmatically.

See Also

C# Compiler Options

/incremental (Enable Incremental Compilation)

```
/incremental[+ | -]
```

Remarks

The **/incremental** compiler option enables the incremental compiler, which compiles only those methods that have changed since the last compilation. Information about the state of the previous compilation is stored in the following files, which are created by the compiler:

output_file_name.dbg
When compiling with /debug, the status of debug information is stored in the program database (.pdb file).

output_file_name.extension.incr
Information about the status of the compilation, other than debug information, is stored in a .incr file.

The first time you use **/incremental**, the .incr and .pdb files are updated and all subsequent compilations will be incremental. Changing the compiler options from the previous **/incremental** compilation will cause a full rebuild of the .incr and .pdb files.

If the compiler detects many changes since the last incremental build, it may perform a full build, as a full build would be at least as efficient as an incremental build. If the compiler cannot find the project's .pdb or .incr files, a full rebuild will occur.

In an incremental build, /doc is ignored. To ensure that your documentation file is up to date, build with **/incremental-**.

By default, **/incremental-** is in effect. Specifying **/incremental** is the same as specifying **/incremental+**.

Output files created with the **/incremental** option may be larger than those created with incremental compilation disabled. Because the output files can be larger, you should use **/incremental-** in the final build of your output file.

/incremental will speed compilation on projects with many, smaller files, and will have little or no speed improvement on projects with fewer, large files.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Setting Visual C# Project Properties.
2. Click the **Configuration Properties** folder.
3. Click the **Advanced** property page.
4. Modify the **Incremental build** property.

To set this compiler option programmatically

See IncrementalBuild Property.

Example

Compile `in.cs` using the incremental compiler:

```
csc /incremental in.cs
```

See Also

C# Compiler Options

/lib (Specify Assembly Reference Locations)

```
/lib:dir1[, dir2]
```

where:

dir1
A directory for the compiler to look in if a referenced assembly is not found in the current working directory (the directory from which you are invoking the compiler) or in the common language runtime's system directory.

dir2
One or more additional directories to search in for assembly references. Separate additional directory names with a semicolon.

Remarks

The **/lib** option specifies the location of assemblies referenced by means of the /reference option.

The compiler searches for assembly references that are not fully qualified in the following order:

1. Current working directory. This is the directory from which the compiler is invoked.
2. The common language runtime system directory.
3. Directories specified by **/lib**.
4. Directories specified by the LIB environment variable.

Use /reference to specify an assembly reference.

/lib is additive; specifying it more than once appends to any prior values.

An alternative to using **/lib** is to copy into the working directory any required assemblies; this will allow you to simply pass the assembly name to **/reference**. You can then delete the assemblies from the working directory. Since the path to the dependent assembly is not specified in the assembly manifest, the application can be started on the target computer and will find and use the assembly in the global assembly cache.

Because the compiler can reference the assembly does not imply the common language runtime will be able to find and load the assembly at runtime. See How the Runtime Locates Assemblies for details on how the runtime searches for referenced assemblies.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Setting Visual C# Project Properties.
2. Click the **Common Properties** folder.
3. Click the **References Path** property page.
4. Modify the contents of the list box.

To set this compiler option programmatically

See ReferencePath Property.

Example

Compile t2.cs to create an .exe file. The compiler will look in the working directory and in the root directory of the C drive for assembly references.

```
csc /lib:c:\ /reference:t2.dll t2.cs
```

See Also

C# Compiler Options

/linkresource (Link to .NET Framework Resource)

```
/linkresource:filename[,identifier]
```

where:

filename
The .NET Framework resource file to which you want to link from the assembly.
identifier (optional)
The logical name for the resource; the name used to load the resource. The default is the name of the file name.

Remarks

The **/linkresource** option creates a link to a .NET Framework resource in the output file; the resource file is not placed in the output file. /resource embeds a resource file in the output file.

Linked resources are public in the assembly when created with the C# compiler.

/linkresource requires one of the /target options other than **/target:module**.

If *filename* is a .NET Framework resource file created, for example, by Resgen.exe or in the development environment, it can be accessed with members in the **System.Resources** namespace (see System.Resources.ResourceManager for more information). For all other resources, use the **GetManifestResource*** methods in the **System.Reflection.Assembly** class to access the resource at run time.

The file specified in *filename* can be any format. For example, you may want to make a native DLL part of the assembly, so it can be installed into the Global Assembly Cache and accessed from managed code in the assembly.

/linkres is the short form of **/linkresource**.

To set this compiler option in the Visual Studio development environment

This compiler option is unavailable in Visual Studio.

To set this compiler option programmatically

This compiler option cannot be changed programmatically.

Example

Compile `in.cs` and link to resource file `rf.resource`:

```
csc /linkresource:rf.resource in.cs
```

See Also

C# Compiler Options

/main (Specify Location of Main Method)

```
/main:class
```

where:

class
The type that contains the **Main** method.

Remarks

If your compilation includes more than one type with a Main method, you can specify which type contains the **Main** method that you want to use as the entry point into the program.

This option is for use when compiling an .exe file.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Setting Visual C# Project Properties.
2. Click the **Common Properties** folder.
3. Click the **General** property page.
4. Modify the **Startup Object** property.

To set this compiler option programmatically

See StartupObject Property.

Example

Compile `t2.cs` and `t3.cs`, specifying that the **Main** method will be found in `Test2`:

```
csc t2.cs t3.cs /main:Test2
```

See Also

C# Compiler Options

/noconfig (Ignore csc.rsp)

```
/noconfig
```

Remarks

The **/noconfig** option tells the compiler not to compile with the csc.rsp file, which is located and loaded from the same directory as the csc.exe file. The csc.rsp file references all the assemblies shipped with the .NET Framework. The actual references that the Visual Studio .NET development environment includes depend on the project type.

You can modify the csc.rsp file and specify additional compiler options that should be included in every compilation from the command line with csc.exe (except the **/noconfig** option).

The compiler processes the options passed to the **csc** command last. Therefore, any option on the command line overrides the setting of the same option in the csc.rsp file.

If you do not want the compiler to look for and use the settings in the csc.rsp file, specify **/noconfig**.

To set this compiler option in the Visual Studio development environment

This compiler option is unavailable in Visual Studio.

To set this compiler option programmatically

This compiler option cannot be changed programmatically.

See Also

C# Compiler Options

/nologo (Suppress Banner Information)

```
/nologo
```

Remarks

The **/nologo** option suppresses display of the sign-on banner when the compiler starts up and display of informational messages during compiling.

This option is not available from within the development environment; it is only available when compiling from the command line.

To set this compiler option in the Visual Studio development environment

This compiler option is unavailable in Visual Studio.

To set this compiler option programmatically

This compiler option cannot be changed programmatically.

See Also

C# Compiler Options

/nostdlib (Do Not Import Standard Library)

```
/nostdlib[+ | -]
```

Remarks

The **/nostdlib** option prevents the import of mscorlib.dll, which defines the entire System namespace. Use this option if you want to define or create your own System namespace and objects.

If you do not specify **/nostdlib**, mscorlib.dll will be imported into your program (same as specifying **/nostdlib-**). If you specify **/nostdlib**, it is the same as specifying **/nostdlib+**.

To set this compiler option in the Visual Studio development environment

1. Open the Property Pages dialog box for the project. For details, see Setting Visual C# Project Properties.
2. Click the **Configuration Properties** folder.
3. Click the **Advanced** property page.
4. Modify the **Do not use Mscorlib** property.

To set this compiler option programmatically

See NoStdLib.

See Also

C# Compiler Options

/nowarn (Suppress Specified Warnings)

```
/nowarn:number1[,number2[...]]
```

where:

number1, *number2*
A warning number that you want the compiler to suppress.

Remarks

The **/nowarn** option lets you suppress the compiler's ability to generate one or more warnings. Separate multiple warning numbers with a comma.

You only need to specify the numeric part of the warning identifier. For example, if you want to suppress CS0028, you could specify `/nowarn:28`.

The compiler will silently ignore warning numbers passed to **/nowarn** that were valid in previous releases, but that have been removed from the compiler. For example, CS0679 was valid in the compiler in Visual Studio .NET 2002 but was subsequently removed.

To set this compiler option in the Visual Studio development environment

1. Open the Property Pages dialog box for the project. For details, see Setting Visual C# Project Properties.
2. Click the **Configuration Properties** folder.
3. Click the **Build** property page.
4. Modify the **Suppress specific warnings** property.

To set this compiler option programmatically

See NoWarn.

See Also

C# Compiler Options

/optimize (Enable/Disable Optimizations)

```
/optimize[+ | -]
```

Remarks

The **/optimize** option enables or disables optimizations performed by the compiler to make your output file smaller, faster, and more efficient. **/optimize** also tells the common language runtime to optimize code at runtime.

By default, **/optimize-** is in effect. Specify **/optimize+** to enable optimizations.

When building a module to be used by an assembly, use the same **/optimize** settings as those of the assembly.

/o is the short form of **/optimize**.

It is possible to combine the **/optimize** and /debug options.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Setting Visual C# Project Properties.
2. Click the **Configuration Properties** folder.
3. Click the **Build** property page.
4. Modify the **Optimize code** property.

To set this compiler option programmatically

See Optimize Property.

Example

Compile `t2.cs` and enable compiler optimizations:

```
csc t2.cs /optimize
```

See Also

C# Compiler Options

/out (Set Output File Name)

```
/out:filename
```

where:

filename
The name of the output file created by the compiler.

Remarks

On the command line, it is possible to specify multiple output files for your compilation. The **/out** option specifies the name of the output file. The compiler expects to find one or more source code files following the **/out** option. Then, all source code files will be compiled into the output file specified by that **/out** option.

If you do not specify the name of the output file:

- An .exe will take its name from the source code file that contains the **Main** method.
- A .dll or .netmodule will take its name from the first source code file.

A source code file used to compile one output file cannot be used in the same compilation for the compilation of another output file.

When producing multiple output files in a command-line compilation, keep in mind that only one of the output files can be an assembly and that only the first output file specified (implicitly or explicitly with **/out**) can be the assembly.

Any modules produced as part of a compilation become files associated with any assembly also produced in the compilation. Use ildasm.exe to view the assembly manifest to see the associated files.

Specify the full name and extension of the file you want to create.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Setting Visual C# Project Properties.
2. Click the **Common Properties** folder.
3. Click the **General** property page.
4. Modify the **Assembly Name** property.

To set this compiler option programmatically

See OutputFileName Property.

Example

Compile `t.cs` and create output file `t.exe`, as well as build `t2.cs` and create module output file `mymodule.netmodule`:

```
csc t.cs /out:mymodule.netmodule /target:module t2.cs
```

See Also

C# Compiler Options

/recurse (Find Source Files in Subdirectories)

```
/recurse:[dir\]file
```

where:

dir (optional)
The directory in which you want the search to begin. If this is not specified, the search begins in the project directory.

file
The file(s) to search for. Wildcard characters are allowed.

Remarks

The **/recurse** option lets you compile source code files in all child directories of either the specified directory (*dir*) or of the project directory.

You can use wildcards in a file name to compile all matching files in the project directory without using **/recurse**.

To set this compiler option in the Visual Studio development environment

This compiler option is unavailable in Visual Studio.

To set this compiler option programmatically

This compiler option cannot be changed programmatically.

Example

Compiles all C# files in the current directory:

```
csc *.cs
```

Compiles all of the C# files in the dir1\dir2 directory and any directories below it and generates dir2.dll:

```
csc /target:library /out:dir2.dll /recurse: dir1\dir2\*.cs
```

See Also

C# Compiler Options

/reference (Import Metadata)

```
/reference:file[;file2]
```

where:

file, *file2*
One or more files that contain an assembly manifest. To import more than one file, separate file names with either a comma or a semicolon.

Remarks

The **/reference** option causes the compiler to make public type information in the specified files available to the project you are currently compiling.

The file(s) you import must contain a manifest; the output file must have been compiled with one of the /target options other than /target:module.

Use /addmodule to import metadata from an output file that does not contain an assembly manifest.

At run time, you should anticipate that only one .exe assembly can be loaded per process, even though, there may be times when more than one .exe might be loaded in the same process. Therefore, do not pass an assembly built with **/target:exe** or **/target:winexe** to **/reference**. This condition may be modified in future versions of the common language runtime.

If you reference an assembly (Assembly A), which itself references another assembly (Assembly B), you will need to reference Assembly B if:

- A type you use from Assembly A inherits from a type or implements an interface from Assembly B.
- If you invoke a field, property, event, or method that has a return type or parameter type from Assembly B.

Use /lib to specify the directory in which one or more of your assembly references is located. The **/lib** topic also discusses which directories the compiler searches for assemblies.

In order for the compiler to recognize a type in an assembly (not a module), it needs to be forced to resolve the type, which you can do, for example, by defining an instance of the type. There are other ways to resolve type names in an assembly for the compiler, for example, if you inherit from a type in an assembly, the type name will then become known to the compiler.

The csc.rsp response file, which references commonly-used .NET Framework assemblies, is used by default. Use /noconfig if you do not want the compiler to use csc.rsp.

/r is the short form of **/reference**.

To set this compiler option in the Visual Studio development environment

See Add Reference Dialog Box.

To set this compiler option programmatically

See Add Method (References Object).

Example

Compile source file `input.cs` and import metadata from `metad1.dll` and `metad2.dll` to produce `out.exe`:

```
csc /reference:metad1.dll;metad2.dll /out:out.exe input.cs
```

See Also

C# Compiler Options

/resource (Embed Resource File to Output)

```
/resource:filename[,identifier]
```

where:

filename
The .NET Framework resource file you want to embed in the output file.
identifier (optional)
The logical name for the resource; the name used to load the resource. The default is the name of the file name.

Remarks

Use /linkresource to link a resource to an assembly and not place the resource file in the output file.

Resources are public in the assembly when created with the C# compiler.

If *filename* is a .NET Framework resource file created, for example, by Resgen.exe or in the development environment, it can be accessed with members in the **System.Resources** namespace (see System.Resources.ResourceManager for more information). For all other resources, use the **GetManifestResource*** methods in the **System.Reflection.Assembly** class to access the resource at run time.

/res is the short form of **/resource**.

To set this compiler option in the Visual Studio development environment

1. Add a resource file to your project.
2. Select the file you want to embed in Solution Explorer.
3. Select **Build Action** for the file in the Properties window.
4. Set **Build Action** to **Embedded Resource**.

To set this compiler option programmatically

See BuildAction Property (Visual Basic and C# ProjectItem Object).

Example

Compile `in.cs` and attach resource file `rf.resource`:

```
csc /resource:rf.resource in.cs
```

See Also

C# Compiler Options

/target (Specify Output File Format)

The **/target** compiler option can be specified in one of four forms:

/target:exe
  Create an .exe file.
/target:library
  Create a code library.
/target:module
  Create a module.
/target:winexe
  Create a Windows program.

Unless you specify **/target:module**, **/target** causes a .NET Framework assembly manifest to be placed in an output file. See Global Attributes for information on how to specify assembly attributes.

The assembly manifest is placed in the first .exe output file in the compilation or in the first .dll, if there is no .exe output file. For example, in the following command line, the manifest will be placed in `1.exe`:

```
csc /out:1.exe t1.cs /out:2.netmodule t2.cs
```

The compiler creates only one assembly manifest per compilation. Information about all files in a compilation is placed in the assembly manifest. All output files except those created with **/target:module** can contain an assembly manifest. When producing multiple output files at the command line, only one assembly manifest can be created and it must go into the first output file specified on the command line. If the first output file is an exe (**/target:exe** or **/target:winexe**) or code library (**/target:library**), then any other output files produced in the same compilation must be modules (**/target:module**).

If you create an assembly, you can indicate that all or part of your code is CLS compliant with the CLSCompliant attribute.

```
// target_clscompliant.cs
[assembly:System.CLSCompliant(true)]   // specify assembly compliance

[System.CLSCompliant(false)]   // specify compliance for an element
class TestClass
{
   public static void Main()
   {
   }
}
```

To set this compiler option programmatically

See OutputType Property.

See Also

C# Compiler Options

/target:exe (Create a Console Application)

```
/target:exe
```

Remarks

The **/target:exe** option causes the compiler to create an executable (EXE), console application. The **/target:exe** option is in effect by default. The executable file will be created with the .exe extension.

Use /target:winexe to create a Windows program executable.

Unless otherwise specified with the /out option, the output file name takes the name of the input file that contains the Main method.

When specified at the command line, all files up to the next **/out**, **/target:winexe**, or /target:library option are used to create the .exe file. The **/target:exe** option is in effect for all files since the previous **/out** or **/target:library** option.

One and only one **Main** method is required in the source code files that are compiled into an .exe file. The /main compiler option lets you specify which class contains the **Main** method, in cases where your code has more than one class with a **Main** method.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Setting Visual C# Project Properties.
2. Click the **Common Properties** folder.
3. Click the **General** property page.
4. Modify the **Output Type** property.

To set this compiler option programmatically

See OutputType Property.

Example

Each of the following command lines will compile `in.cs`, creating `in.exe`:

```
csc /target:exe in.cs
csc in.cs
```

See Also

/target (Specify Output File Format) | C# Compiler Options

/target:library (Create a Code Library)

```
/target:library
```

Remarks

The **/target:library** option causes the compiler to create a dynamic-link library (DLL) rather than an executable file (EXE). The DLL will be created with the .dll extension.

Unless otherwise specified with the /out option, the output file name takes the name of the first input file.

When specified at the command line, all files up to the next **/out**, /target:winexe, or /target:exe option are used to create the .dll file.

When building a .dll file, a Main method is not required.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Setting Visual C# Project Properties.
2. Click the **Common Properties** folder.
3. Click the **General** property page.
4. Modify the **Output Type** property.

To set this compiler option programmatically

See OutputType Property.

Example

Compile `in.cs`, creating `in.dll`:

```
csc /target:library in.cs
```

See Also

/target (Specify Output File Format) | C# Compiler Options

/target:module (Create Module to Add to Assembly)

```
/target:module
```

Remarks

To not generate an assembly manifest, use the **/target:module** option. By default, the output file will have an extension of .netmodule.

A file that does not have an assembly manifest cannot be loaded by the .NET Framework common language runtime. However, such a file can be incorporated into the assembly manifest of an assembly by means of /addmodule.

If more than one module is created in a single compilation, internal types in one module will be available to other modules in the compilation. When code in one module references **internal** types in another module, then both modules must be incorporated into an assembly manifest, by means of **/addmodule**.

To set this compiler option in the Visual Studio development environment

Creating a module is not supported in the Visual Studio development environment.

To set this compiler option programmatically

See OutputType Property.

Example

Compile `in.cs`, creating `in.netmodule`:

```
csc /target:module in.cs
```

See Also

/target (Specify Output File Format) | C# Compiler Options

/target:winexe (Create a Windows Program)

```
/target:winexe
```

Remarks

The **/target:winexe** option causes the compiler to create an executable (EXE), Windows program. The executable file will be created with the .exe extension. A Windows program is one that provides a user interface from either the .NET Framework library or with the Win32 APIs.

Use /target:exe to create a console application.

Unless otherwise specified with the /out option, the output file name takes the name of the input file that contains the Main method.

When specified at the command line, all files until the next **/out** or /target option are used to create the Windows program.

One and only one **Main** method is required in the source code files that are compiled into an .exe file. The /main option lets you specify which class contains the **Main** method, in cases where your code has more than one class with a **Main** method.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Setting Visual C# Project Properties.
2. Click the **Common Properties** folder.
3. Click the **General** property page.
4. Modify the **Output Type** property.

To set this compiler option programmatically

See OutputType Property.

Example

Compile `in.cs` into a Windows program:

```
csc /target:winexe in.cs
```

See Also

/target (Specify Output File Format) | C# Compiler Options

/unsafe (Enable Unsafe Mode)

```
/unsafe
```

Remarks

The **/unsafe** option allows code that uses the unsafe keyword to compile.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Setting Visual C# Project Properties.
2. Click the **Configuration Properties** folder.
3. Click the **Build** property page.
4. Modify the **Allow unsafe code blocks** property.

To set this compiler option programmatically

See AllowUnsafeBlocks Property.

Example

Compile `in.cs` for unsafe mode:

```
csc /unsafe in.cs
```

See Also

C# Compiler Options

/utf8output (Display Compiler Messages with UTF-8)

```
/utf8output
```

Remarks

The **/utf8output** option displays compiler output using UTF-8 encoding.

In some international configurations, compiler output cannot correctly be displayed in the console. In these configurations, use **/utf8output** and redirect compiler output to a file.

To set this compiler option in the Visual Studio development environment

This compiler option is unavailable in Visual Studio.

To set this compiler option programmatically

This compiler option cannot be changed programmatically.

See Also

C# Compiler Options

/warn (Specify Warning Level)

```
/warn:option
```

where:

option
The minimum warning level you want displayed for the build. Valid values are 0-4:

Warning level	Meaning
0	Turns off emission of all warning messages.
1	Displays severe warning messages.
2	Displays level 1 warnings plus certain, less-severe warnings, such as warnings about hiding class members.
3	Displays level 2 warnings plus certain, less-severe warnings, such as warnings about expressions that always evaluate to **true** or **false**.
4	Displays all level 3 warnings plus informational warnings. This is the default warning level at the command line.

Remarks

The **/warn** option specifies the warning level for the compiler to display.

The Build Errors documentation describes the warnings, indicates each warning's level, and indicates potential problems (rather than actual coding errors) with statements that may not compile as you intend.

Use /warnaserror to treat all warnings as errors. Use /nowarn to disable certain warnings.

/w is the short form of **/warn**.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Setting Visual C# Project Properties.
2. Click the **Configuration Properties** folder.
3. Click the **Build** property page.
4. Modify the **Warning Level** property.

To set this compiler option programmatically

See WarningLevel Property.

Example

Compile `in.cs` and have the compiler only display level 1 warnings:

```
csc /warn:1 in.cs
```

See Also

C# Compiler Options

/warnaserror (Treat Warnings as Errors)

```
/warnaserror[+ | -]
```

Remarks

The **/warnaserror+** option treats all warnings as errors. Any messages that would ordinarily be reported as warnings are instead reported as errors, and the build process is halted (no output files are built).

By default, **/warnaserror-** is in effect, which causes warnings to not prevent the generation of an output file. **/warnaserror**, which is the same as **/warnaserror+**, causes warnings to be treated as errors.

Use /warn to specify the level of warnings that you want the compiler to display. Use /nowarn to disable certain warnings.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Setting Visual C# Project Properties.
2. Click the **Configuration Properties** folder.
3. Click the **Build** property page.
4. Modify the **Treat Warnings As Errors** property.

To set this compiler option programmatically

See TreatWarningsAsErrors Property.

Example

Compile `in.cs` and have the compiler display no warnings:

```
csc /warnaserror in.cs
```

See Also

C# Compiler Options

/win32icon (Import an .ico File)

```
/win32icon:filename
```

where:

filename
The .ico file that you want to add to your output file.

Remarks

The **/win32icon** option inserts an .ico file in the output file, which gives the output file the desired appearance in the Windows Explorer. An .ico file can be created with the Resource Compiler. The Resource Compiler is invoked when you compile a Visual C++ program; an .ico file is created from the .rc file.

See /linkresource (to reference) or /resource (to attach) a .NET Framework resource file. See /win32res to import a .res file.

To set this compiler option in the Visual Studio development environment

1. Open the project's **Property Pages** dialog box. For details, see Setting Visual C# Project Properties.
2. Click the **Common Properties** folder.
3. Click the **General** property page.
4. Modify the **Application Icon** property.

To set this compiler option programmatically

See ApplicationIcon Property.

Example

Compile `in.cs` and attach an .ico file `rf.ico` to produce `in.exe`:

```
csc /win32icon:rf.ico in.cs
```

See Also

C# Compiler Options

/win32res (Import a Win32 Resource File)

```
/win32res:filename
```

where:

filename
The resource file that you want to add to your output file.

Remarks

The **/win32res** option inserts a Win32 resource in the output file. A Win32 resource file can be created with the Resource Compiler. The Resource Compiler is invoked when you compile a Visual C++ program; a .res file is created from the .rc file.

A Win32 resource can contain version or bitmap (icon) information that would help identify your application in the Windows Explorer. If you do not specify **/win32res**, the compiler will generate version information based on the assembly version.

See /linkresource (to reference) or /resource (to attach) a .NET Framework resource file.

To set this compiler option in the Visual Studio development environment

This compiler option is unavailable in Visual Studio.

To set this compiler option programmatically

This compiler option cannot be changed programmatically.

Example

Compile `in.cs` and attach a Win32 resource file `rf.res` to produce `in.exe`:

```
csc /win32res:rf.res in.cs
```

See Also

C# Compiler Options

Compiler Errors CS0001 Through CS9999

This section is a reference to the errors generated by the C# compiler. To get help on a particular error message, either click the mouse on an error number in the Output window and press F1, or type the error number in the **Look for** box in the Index.

See Also

C# Compiler Options

Compiler Error CS0001

Internal compiler error

Try to determine if the compiler is failing because of its inability to parse unexpected syntax. Then, contact Microsoft Product Support Services.

Compiler Error CS0003

Out of memory

The compiler was unable to allocate enough virtual memory to complete compilation. Close all unnecessary applications and compile again.

You may also want to increase the pagefile size and make sure you have free disk space.

This error may also appear as the result of having mismatched versions of the .NET Framework SDK and the C# compiler or one or more corrupt files that support the C# compiler; reinstall Visual Studio.

Compiler Error CS0004

Warning treated as error

You compiled with the /warnaserror compiler option, which causes the compiler to generate warnings as errors.

Compiler Error CS0005

Compiler option '*compiler_option*' must be followed by an argument

Some compiler options require parameters. If you do not pass the arguments required by the compiler option, CS0005 is generated.

For more information, see C# Compiler Options.

Compiler Error CS0006

Metadata file '*dll_name*' could not be found

The program was compiled and explicitly passed the name of a file that contained metadata; however, the .dll was not found.

Compiler Error CS0007

Unexpected common language runtime initialization error — '*description*'

The common language runtime installation may be corrupt.

Reinstall the common language runtime portion of your operating system.

Compiler Error CS0008

Unexpected error reading metadata from file '*file*' — '*description*'

A DLL was successfully opened for retrieving metadata but is corrupted, such that data could not be read from it.

Get an uncorrupted DLL to import.

Compiler Error CS0009

Metadata file '*file*' could not be opened — '*description*'

The file specified with the /reference compiler option does not contain valid metadata.

Compiler Error CS0010

Cannot declare a namespace and a type both named '*name*'

A user-defined namespace is conflicting with an imported type name.

For example, the following sample generates CS0010:

```
// CS0010.cs
namespace System.Byte   // CS0010, rename namespace
{
   class MyClass
   {
      public static int Main()
      {
         return(0);
      }
   }
}
```

Compiler Error CS0011

Referenced class '*class1*' has base class or interface '*base1*' defined in an assembly that is not referenced. You must add a reference to assembly '*assembly*'.

A class that was imported from a file with /reference, is derived from a class or implements an interface that is not found. This could occur if a required .DLL is not also included in the compilation with /reference.

See Add Reference Dialog Box for information on how to add a reference in the development environment.

The following sequence of compilations will result in CS0011:

```
// CS0011_1.cs
// compile with: /target:library
public class B1
{
}
```

Then:

```
// CS0011_2.cs
// compile with: /r:CS0011_1.dll /target:library
public class B2: B1
{
}
```

Then:

```
// CS0011_3.CS
// compile with: /r:CS0011_2.dll
// CS0011 expected
// to fix, compile with: /r:CS0011_1.dll;CS0011_2.dll
class D1
{
   public B2 b2 = null;
   public static void Main()
   {
   }
}
```

Compiler Error CS0012

The type '*type*' is defined in an assembly that is not referenced. You must add a reference to assembly '*assembly*'.

The definition for a referenced type was not found. This could occur if a required .DLL file is not included in the compilation.

The following sequence of compilations will result in a CS0012 error:

```
// cs0012a.cs
// compile with: /t:library
public class A
{
}
```

Then:

```
// cs0012b.cs
// compile with: /t:library /r:cs0012a.dll
public class B
{
   public static A f()
   {
      return new A();
   }
}
```

Then:

```
// cs0012c.cs
// compile with: /r:cs0012b.dll
class C
{
   public static void Main()
   {
      object o = B.f();   // CS0012
   }
}
```

You could resolve this CS0012 by compiling with `/r:b.dll;a.dll`.

Compiler Error CS0013

Unexpected error writing metadata to file '*file*' — '*description*'

The .NET Framework common language runtime failed to emit metadata. This could be a number of problems:

- Disk full.
- Corrupt Visual Studio installation; reinstall Visual Studio.
- Corrupt common language runtime; reinstall the common language runtime.

Compiler Error CS0014

Required file '*file*' could not be found

A file is needed by the compiler but is not on the system.

Compiler Error CS0015

The name of type '*type*' is too long

The fully qualified name of a user-defined type must be less than or equal to 2048 characters.

Compiler Error CS0016

Could not write to output file '*file*' — '*reason*'

The compiler could not write to an output file. The file may have read-only status.

Compiler Error CS0017

Program '*output file name*' has more than one entry point defined: *function*

A program can only have one Main method.

To resolve this error, you can either delete all Main methods in your code, except one, or you can use the /main compiler option to specify which Main method you want to use.

The following sample generates CS0017:

```
// CS0017.cs
public class clx
{
   static public void Main()
   {
   }
}

public class cly
{
   public static void Main()   // CS0017, delete one Main or use /main
   {
   }
}
```

Compiler Error CS0019

Operator '*operator*' cannot be applied to operands of type '*type*' and '*type*'

A binary operator is operating on data types for which it was not designed. For example, you cannot use the || operator on strings.

The following sample generates CS0019:

```
// CS0019.cs
public class MyClass
{
   [System.Diagnostics.ConditionalAttribute("DEBUG" || "TRACE")]   // CS0019
   // try the following line instead
   // [System.Diagnostics.ConditionalAttribute("DEBUG")]
   public void TestMethod()
   {
   }

   public static void Main()
   {
   }
}
```

In this example, conditional logic must be specified outside the **ConditionalAttribute**. You can only pass one predefined symbol to the **ConditionalAttribute**.

Compiler Error CS0020

Division by constant zero

An expression uses a literal (not variable) value of zero in the denominator of a division operation. Division by zero is not defined, and therefore invalid.

The following sample generates CS0020:

```
// CS0020.cs
namespace x
{
   public class b
   {
      public static void Main()
      {
         1 / 0;   // CS0020
      }
   }
}
```

Compiler Error CS0021

Cannot apply indexing with [] to an expression of type '*type*'

An attempt was made to access a value through an index operation on a data type that does not support array-like operations.

You may get CS0021 when trying to use an indexer in a C++ assembly. In this case, to decorate the C++ class with the DefaultMember attribute so the C# compiler knows which indexer is the default. For example,

```
[DefaultMember("Item")]
__gc public class StringBreaker: public IEnumerable {
   public:
   StringBreaker::StringBreaker(String* string);

   __property int get_Count();

      // indexer for the strings...
   __property String * get_Item(int index);
   __property void set_Item(int index, String* value);

   IEnumerator* GetEnumerator();

   private:
   ArrayList* entries;
};
```

The following sample generates CS0021. It consists of a .cpp file, which compiles to a .dll, and a .cs file that uses this .dll.

```
// CPP0021.cpp
// compile with: /clr /LD
#using <mscorlib.dll>
using namespace System;
using namespace System::Reflection;

// Adding this attribute solves the problem
//[DefaultMember("Item")]
//
__gc public class MyClassMC {
   public:
   // indexer Indexer for the strings...
   __property int get_Item(int index){  return index; }
   __property void set_Item(int index, int value) {};
};
```

The following is the C# file:

```
// CS0021b.cs
// compile with: /reference:CPP0021.dll
public class MyClass
{
   public static void Main()
   {
      MyClassMC i = new MyClassMC();
      i[2] = 10;   // CS0021
   }
}
```

Compiler Error CS0022

Wrong number of indices inside [], expected '*number*'

An array-access operation specified multiple indices where only one is allowed.

The following sample generates CS0022:

```
// CS0022.cs
public class MyClass
{
   public static void Main()
   {
      int[] a = new int[10];
      a[0] = 0;     // single-dimension array
      a[0,1] = 9;   // CS0022
   }
}
```

Compiler Error CS0023

Operator '*operator*' cannot be applied to operand of type '*type*'

An attempt was made to apply an operator to a variable whose type was not designed to work with the operator.

The following sample generates CS0023:

```
// CS0023.cs
namespace x
{
   public class a
   {
      public static void Main()
      {
         string s = "hello";
         s = -s;   // CS0023, minus operator not allowed on strings
      }
   }
}
```

Compiler Error CS0025

Standard library file '*file*' could not be found

A file that is needed by the compiler was not found. Check your path to ensure that the file is on the path. Also check that the file is on your computer.

You may need to reinstall Visual Studio to resolve this error.

Compiler Error CS0026

Keyword this is not valid in a static property, static method, or static field initializer

The this keyword is not allowed in static methods; you must have an instance before you can refer to an object with **this**.

The following sample generates CS0026:

```
// CS0026.cs
public class a
{
   public static int i = 0;

   public static void Main()
   {
      this.i = this.i + 1;   // CS0026
      // try the following line instead
      // i = i + 1;
   }
}
```

Compiler Error CS0027

Keyword this is not available in the current context

The this keyword was found outside of a property, method, or constructor.

The following sample generates CS0027:

```
// CS0027.cs
using System;
class CmyClass
{
   int err1 = this.fun() + 1;   // CS0027

   public int fun()
   {
      return 10;
   }

   public void x()
   {
      int err = this.fun() + 1;   // valid use of this
      Console.WriteLine(err);
   }

   public static void Main()
   {
      CMyClass c = new CMyClass();
      c.x();
   }
}
```

Compiler Warning (level 4) CS0028

'*function declaration*' has the wrong signature to be an entry point

The method declaration for Main was invalid.

The following sample generates CS0028:

```
// CS0028.cs
// compile with: /W:4 /warnaserror
public class a
{
   public static void Main(int i)   // CS0028
   // try the following line instead
   // public static void Main()
   {
   }
}
```

Compiler Error CS0029

Cannot implicitly convert type '*type*' to '*type*'

The compiler requires an explicit conversion. For example, you may need to cast an r-value to be the same type as an l-value. Or, you must provide conversion routines to support certain operator overloads.

Conversions must occur when assigning a variable of one type to a variable of a different type. When making an assignment between variables of different types, the compiler must convert the type on the right-hand side of the assignment operator to the type on the left-hand side of the assignment operator. Take the following the code:

```
int i = 50;
long lng = 100;
i = lng;
```

`i = lng;` makes an assignment, but the data types of the variables on the left and right-hand side of the assignment operator don't match. Before making the assignment the compiler is implicitly converting the variable `lng`, which is of type long, to an int. This is implicit because no code explicitly instructed the compiler to perform this conversion. The problem with this code is that this is considered a narrowing conversion, and the compiler does not allow implicit narrowing conversions because there could be a potential loss of data.

A narrowing conversion exists when converting to a data type that occupies less storage space in memory than the data type we are converting from. For example, converting a long to an int would be considered a narrowing conversion. A long occupies 8 bytes of memory while an int occupies 4 bytes. To see how data loss can occur, consider the following sample:

```
int i = 50;
long lng = 3147483647;
i = lng;
```

The variable `lng` now contains a value that cannot be stored in the variable `i` because it is too large. If we were to convert this value to an int type we would be losing some of our data and the converted value would not be the same as the value before the conversion.

A widening conversion would be the opposite of a narrowing conversion. With widening conversions, we are converting to a data type that occupies more storage space in memory than the data type we are converting from. Here is an example of a widening conversion:

```
int i = 50;
long lng = 100;
lng = i;
```

Notice the difference between this code sample and the first. This time the variable `lng` is on the left-hand side of the assignment operator, so it is the target of our assignment. Before the assignment can be made, the compiler must implicitly convert the variable `i`, which is of type int, to type long. This is a widening conversion since we are converting from a type that occupies 4 bytes of memory (an int) to a type that occupies 8 bytes of memory (a long). Implicit widening conversions are allowed because there is no potential loss of data. Any value that can be stored in an int can also be stored in a long.

We know that implicit narrowing conversions are not allowed, so to be able to compile this code we need to explicitly convert the data type. Explicit conversions are done using casting. Casting is the term used in C# to describe converting one data type to another. To get the code to compile we would need to use the following syntax:

```
int i = 50;
long lng = 100;
i = (int) lng;   // cast to int
```

The third line of code tells the compiler to explicitly convert the variable `lng`, which is of type long, to an int before making the assignment. Remember that with a narrowing conversion, there is a potential loss of data. Narrowing conversions should be used with caution and even though the code will compile you may get unexpected results at run-time.

This discussion has only been for value types. When working with value types you work directly with the data stored in the variable. However, the .NET Framework also has reference types. When working with reference types you are working with a reference to a variable, not the actual data. Examples of reference types would be classes, interfaces and arrays. You cannot

implicitly or explicitly convert one reference type to another unless the compiler allows the specific conversion or the appropriate conversion operators are implemented.

The following sample generates CS0029:

```
// CS0029.cs
public class MyInt
{
    private int x = 0;

    // Uncomment this conversion routine to resolve CS0029
    /*
    public static implicit operator int(MyInt i)
    {
       return i.x;
    }
    */


    public static void Main()
   {
      MyInt myInt = new MyInt();
      int i = myInt; // CS0029
   }
}
```

Compiler Error CS0030

Cannot convert type '*type*' to '*type*'

You must provide conversion routines to support certain operator overloads.

The following sample generates CS0030:

```
// CS0030.cs
namespace x
{
   public class iii
   {
      /*
      public static implicit operator iii(int aa)
      {
         return null;
      }

      public static implicit operator int(iii aa)
      {
         return 0;
      }
      */

      public static iii operator ++(iii aa)
      {
         return (iii)0;   // CS0030
         // uncomment the conversion routines to resolve CS0030
      }

      public static void Main()
      {
      }
   }
}
```

Compiler Error CS0031

Constant value '*value*' cannot be converted to a '*type*'

An attempt was made to assign a value to a variable whose type cannot store the value.

The following sample generates CS0031:

```
// CS0031.cs
namespace x
{
   public class a
   {
      public static void Main()
      {
         byte i = 512;   // CS0031, 512 cannot fit in byte
      }
   }
}
```

Compiler Error CS0032

Could not open incremental build file '*file*' for writing

During an incremental build (/incremental), the compiler could not open the output file. The output file may have read-only status, which you should change to read/write status.

Compiler Error CS0033

Out of disk space or I/O error writing incremental build file '*file*'

During an incremental build (/incremental), the compiler writes to a file containing dependency information used to make rebuild information faster. This write could fail. Perhaps the file had read-only status, which you should change to read/write status.

Compiler Error CS0034

Operator '*operator*' is ambiguous on operands of type '*type1*' and '*type2*'

An operator was used on two objects and the compiler found more than one conversion. Conversions need to be unique, so you either need to make a cast or to make one of the conversions explicit.

The following sample generates CS0034:

```
// CS0034.cs
public class A
{
   // allows for the conversion of A object to int
   public static implicit operator int (A s)
   {
      return 0;
   }

   public static implicit operator string (A i)
   {
      return null;
   }
}

public class B
{
   public static implicit operator int (B s)
   // one way to resolve this CS0034 is to make one conversion explicit
   // public static explicit operator int (B s)
   {
      return 0;
   }

   public static implicit operator string (B i)
   {
      return null;
   }

   public static implicit operator B (string i)
   {
      return null;
   }

   public static implicit operator B (int i)
   {
      return null;
   }
}

public class C
{
   public static void Main ()
   {
      A a = new A();
      B b = new B();
      b = b + a;   // CS0034
      // another way to resolve this CS0034 is to make a cast
      // b = b + (int)a;
   }
}
```

Compiler Error CS0035

Operator '*operator*' is ambiguous on an operand of type '*type*'

The compiler has more than one available conversion and does not know which to choose before applying the operator.

The following sample generates CS0035:

```
// CS0035.cs
class MyClass
{
   private int i;

   public MyClass(int i)
   {
      this.i = i;
   }

   public static implicit operator double(MyClass x)
   {
      return (double) x.i;
   }

   public static implicit operator decimal(MyClass x)
   {
      return (decimal) x.i;
   }
}

class MyClass2
{
   static void Main()
   {
      MyClass x = new MyClass(7);
      object o = - x;   // CS0035
      // try a cast:
      // object o = - (double)x;
   }
}
```

Compiler Error CS0036

An out parameter cannot have the '[In]' attribute

Currently, the **In** attribute is not permitted on an out parameter.

The following sample generates CS0036:

```
// CS0036.cs

using System;
using System.Runtime.InteropServices;

public class MyClass
{
   public static void TestOut([In] out char TestChar)   // CS0036
   // try the following line instead
   // public static void TestOut(out char TestChar)
   {
      TestChar = 'b';
      Console.WriteLine(TestChar);
   }

   public static void Main()
   {
      char i;           // variable need not be initialized
      TestOut(out i);   // the arg must be passed as out
      Console.WriteLine(i);
   }
}
```

Compiler Error CS0037

Cannot convert null to '*type*' because it is a value type

The compiler cannot assign null to a value type; null can only be assigned to a reference type. struct is a value type.

The following sample generates CS0037:

```
// CS0037.cs
public struct s
{
}

class a
{
   public static void Main()
   {
      int i = null;   // CS0037
      s ss = null;    // CS0037
   }
}
```

Compiler Error CS0038

Cannot access a nonstatic member of outer type '*type1*' via nested type '*type2*'

A field in a class is not automatically available to a nested class. To be available to a nested class, the field must be static. Otherwise, you must create an instance of the outer class.

The following sample generates CS0038:

```
// CS0038.cs
class OuterClass
{
   public int count;
   // try the following line instead
   // public static int count;

   class InnerClass
   {
      void func()
      {
         // or, create an instance
         // OuterClass class_inst = new OuterClass();
         // int count2 = class_inst.count;
         int count2 = count;   // CS0038
      }
   }

   public static void Main()
   {
   }
}
```

Compiler Error CS0039

Cannot convert type '*type1*' to '*type2*'

A conversion with the as operator is only allowed by inheritance.

The following sample generates CS0039:

```
// CS0039.cs
using System;

class A
{
}

class B: A
{
}

class C: A
{
}

class M
{
   static void Main()
   {
      A a = new C();
      B b = new B();
      C c;

      c = a as C;  // valid, there is a built-in reference
                   // conversion from A to C.

      c = b as C;  // CS0039, there is no built-in reference
                   // conversion from B to C.
   }
}
```

Compiler Error CS0040

Unexpected debug information initialization error — '*reason*'

This error can occur when using the /debug compiler option and indicates that the compiler was unable to write to the .pdb file. Possible resolutions to this error include reinstalling Visual Studio, ensuring that the compiler has write access to a file or directory, or not compiling with /debug.

Compiler Error CS0041

Unexpected error writing debug information to file '*file*' — '*reason*'

This error can occur when using the /debug compiler option. If you are unable to resolve the error based on the information in *reason*, do not compile with /debug.

Compiler Error CS0042

Unexpected error creating debug information file '*file*' — '*reason*'

The compiler could not create the debug information; *reason* contains additional information on the error condition.

Compiler Error CS0043

PDB file '*file*' has an incorrect or out-of-date format. Delete it and rebuild.

Delete the .pdb file for this compilation and recompile.

Compiler Error CS0050

Inconsistent accessibility: return type '*type*' is less accessible than method '*method*'

The return type and each of the types referenced in the formal parameter list of a method must be at least as accessible as the method itself (see 3.5.4 Accessibility constraints).

The following sample generates CS0050:

```
// CS0050.cs
class MyClass
// try the following line instead
// public class MyClass
{
}

public class MyClass2
{
   public static MyClass MyMethod() // CS0050
   {
      return new MyClass();
   }

   public static void Main()
   {
   }
}
```

Compiler Error CS0051

Inconsistent accessibility: parameter type '*type*' is less accessible than method '*method*'

The return type and each of the types referenced in the formal parameter list of a method must be at least as accessible as the method itself (see 3.5.4 Accessibility constraints).

The following sample generates CS0051:

```
// CS0051.cs
class A
// try the following line instead
// public class A
{
   public static implicit operator A(int i)
   {
      return null;
   }
}

class B
// try the following line instead
// public class B
{
   public static implicit operator B(int i)
   {
      return null;
   }
}

public class C
{
   public void f (A a)
   {
   }

   public void f (B b)
   {
   }

   public static void Main ()
   {
      C c = new C();
      c.f((A)5);     // CS0051, could convert either to A or B
   }
}
```

Compiler Error CS0052

Inconsistent accessibility: field type '*type*' is less accessible than field '*field*'

A public construct must return a publicly accessible object.

The following sample generates CS0052:

```
// CS0052.cs
class MyClass
// try the following line instead
// public class MyClass
{
}

public class MyClass2
{
   public MyClass mf;   // CS0052
}

public class MyClass3
{
   public static void Main()
   {
   }
}
```

Compiler Error CS0053

Inconsistent accessibility: property type '*type*' is less accessible than property '*property*'

A public construct must return a publicly accessible object.

The following sample generates CS0053:

```
// CS0053.cs
class MyClass
// try the following line instead
// public class MyClass
{
}

public class MyClass2
{
   public MyClass MyProperty   // CS0053
   {
      get
      {
         return new MyClass();
      }
      set
      {
      }
   }
}

public class MyClass3
{
   public static void Main()
   {
   }
}
```

Compiler Error CS0054

Inconsistent accessibility: indexer return type '*type*' is less accessible than indexer '*indexer*'

A public construct must return a publicly accessible object.

The following sample generates CS0054:

```
// CS0054.cs
class MyClass
// try the following line instead
// public class MyClass
{
}

public class MyClass3
{
   public MyClass this[int i]   // CS0054
   {
      get
      {
         return new MyClass();
      }
   }
}

public class MyClass2
{
   public static void Main()
   {
   }
}
```

Compiler Error CS0055

Inconsistent accessibility: parameter type '*type*' is less accessible than indexer '*indexer*'

A public construct must return a publicly accessible object.

The following sample generates CS0055:

```
// CS0055.cs
class MyClass
// try the following line instead
// public class MyClass
{
}

public class MyClass2
{
   public int this[MyClass i]   // CS0055
   {
      get
      {
         return 0;
      }
   }
}

public class MyClass3
{
   public static void Main()
   {
   }
}
```

Compiler Error CS0056

Inconsistent accessibility: return type '*type*' is less accessible than operator '*operator*'

A public construct must return a publicly accessible object.

The following sample generates CS0056:

```
// CS0056.cs
class ii
// try the following line instead
// public class ii
{
}

public class a
{
   public static implicit operator ii(a aa)   // CS0056
   {
      return new ii();
   }

   public static void Main()
   {
   }
}
```

Compiler Error CS0057

Inconsistent accessibility: parameter type '*type*' is less accessible than operator '*operator*'

A public construct must return a publicly accessible object.

The following sample generates CS0057:

```
// CS0057.cs
class MyClass
// try the following line instead
// public class MyClass
{
}

public class MyClass2
{
   public static implicit operator MyClass2(MyClass iii)   // CS0057
   {
      return new MyClass2();
   }

   public static void Main()
   {
   }
}
```

Compiler Error CS0058

Inconsistent accessibility: parameter type '*type*' is less accessible than delegate '*delegate*'

A public construct must return a publicly accessible object.

The following sample generates CS0058:

```
// CS0058.cs
class MyClass
// try the following line instead
// public class MyClass
{
}

public delegate MyClass MyClassDel();   // CS0058

public class a
{
   public static void Main()
   {
   }
}
```

Compiler Error CS0059

Inconsistent accessibility: parameter type '*type*' is less accessible than delegate '*delegate*'

The return type and each of the types referenced in the formal parameter list of a method must be at least as accessible as the method itself (see 3.5.4 Accessibility constraints).

The following sample generates CS0059:

```
// CS0059.cs
class MyClass
// try the following line instead
// public class MyClass
{
}

public delegate void MyClassDel( MyClass iii);   // CS0059

public class a
{
   public static void Main()
   {
   }
}
```

Compiler Error CS0060

Inconsistent accessibility: base class '*class1*' is less accessible than class '*class2*'

Class accessibility should be consistent between the base class and inherited class.

The following sample generates CS0060:

```
// CS0060.cs
class MyClass
// try the following line instead
// public class MyClass
{
}

public class MyClass2 : MyClass   // CS0060
{
   public static void Main()
   {
   }
}
```

Compiler Error CS0061

Inconsistent accessibility: base interface '*interface 1*' is less accessible than interface '*interface 2*'

A public construct must return a publicly accessible object.

Interface accessibility cannot be narrowed in a derived interface.

The following sample generates CS0061:

```
// CS0061.cs
// resolve this CS0061 by either making a public and b internal
// or by making both interfaces internal
internal interface a
{
   void mf();
}

public interface b : a   // CS0061
{
   new void mf();
}

class x
{
   public static void Main()
   {
   }
}
```

Compiler Error CS0065

'*event*' : event property must have both add and remove accessors

An event that is not a field must have both access methods.

The following sample generates CS0065:

```
// CS0065.cs
using System;
public delegate void Eventhandler(object sender, int e);
public class MyClass
{
   public event EventHandler Click   // CS0065,
   {
      // uncomment the add and remove definitions
      /*
      add
      {
         Click += value;
      }
      remove
      {
         Click -= value;
      }
      */
   }

   public static void Main()
   {
   }
}
```

Compiler Error CS0066

'*event*': event must be of a delegate type

This declaration must be a delegate type.

The following sample generates CS0066:

```
// CS0066.cs
using System;

public class EventHandler
{
}

// try the following declaration for EventHandler (deleting the EventHandler class)
// public delegate void EventHandler();

public class a
{
   public event EventHandler Click;   // CS0066

   private void TestMethod()
   {
      if (Click != null)
         Click();
   }

   public static void Main()
   {
   }
}
```

Compiler Warning (level 3) CS0067

The event '*event*' is never used

An event was declared but never used in the class in which it was declared.

The following sample generates CS0067:

```
// CS0067.cs
// compile with: /W:3
using System;
delegate void MyDelegate();

class MyClass
{
   public event MyDelegate evt;   // CS0067
   // uncomment TestMethod to resolve this CS0067
/*
   private void TestMethod()
   {
      if (evt != null)
         evt();
   }
*/
   public static void Main()
   {
   }
}
```

Compiler Error CS0068

'*event*': event in interface cannot have initializer

An event in an interface cannot have an initializer.

The following sample generates CS0068:

```
// CS0068.cs

delegate void MyDelegate();

interface I
{
   event MyDelegate d = new MyDelegate(M.f);   // CS0068
   // try the following line instead
   // event MyDelegate d2;
}

class M
{
   event MyDelegate d = new MyDelegate(M.f);

   public static void f()
   {
   }

   public static void Main()
   {
   }
}
```

Compiler Error CS0069

'*event*': event in interface cannot have add or remove accessors

You cannot define an event's accessor functions in an interface.

The following sample generates CS0069:

```
// CS0069.cs
using System;
public delegate void EventHandler();

public interface a
{
   event EventHandler Click   // CS0069
   {
      add
      {
      }
      remove
      {
      }
   }
   // remove the preceding EventHandler block
   // try the following line instead
   // event EventHandler Click;
}

public class Class1
{
   public static void Main()
   {
   }
}
```

Compiler Error CS0070

The event '*event*' can only appear on the left hand side of += or -= (except when used from within the type '*type*')

Outside of the class it is defined in, an event can only add or subtract references.

The following sample generates CS0070:

```
// CS0070.cs
using System;
public delegate void EventHandler();

public class A
{
   public event EventHandler Click;

   public static void OnClick()
   {
      EventHandler eh;
      A a = new A();
      eh = a.Click;
   }

   public static void Main()
   {
   }
}

public class B
{
   public int mf ()
   {
      EventHandler eh = new EventHandler(A.OnClick);
      A a = new A();
      eh = a.Click;   // CS0070
      // try the following line instead
      // a.Click += eh;
      return 1;
   }
}
```

Compiler Error CS0071

An explicit interface implementation of an event must use property syntax

When implementing an event that was declared in an interface, you must use property syntax.

The following sample generates CS0071:

```
// CS0071.cs
public delegate void MyEvent(object sender);

interface Itest
{
   event MyEvent Clicked;
}

class Test : Itest
{
   event MyEvent ITest.Clicked()   // CS0071
   {
   }

   // try the following code instead
   /*
   event MyEvent ITest.Clicked
   {
      get
      {
         return null;
      }
      set
      {
      }
   }
   */

   public static void Main()
   {
   }
}
```

Compiler Error CS0072

'*event*' : cannot override; '*method*' is not an event

An event can only override another event.

The following sample generates CS0072:

```
// CS0072.cs
delegate void del();

class Test1
{
   public virtual event del MyEvent;
   public virtual void VMeth()
   {
   }

   public void FireAway()
   {
      if (MyEvent != null)
         MyEvent();
   }
}

class Test2 : Test1
{
   public override event del VMeth   // CS0072
   // uncomment the following lines to resolve
   // public override event del MyEvent
   {
      add
      {
         VMeth += value;
         // MyEvent += value;
      }
      remove
      {
         VMeth -= value;
         // MyEvent -= value;
      }
   }

   public static void Main()
   {
   }
}
```

Compiler Error CS0073

An add or remove accessor must have a body

An **add** or **remove** keyword in an event definition must have a body.

The following sample generates CS0073:

```
// CS0073.cs
delegate void del();

class Test
{
   public event del MyEvent
   {
      add;   // CS0073
      // try the following lines instead
      // add
      // {
      //    MyEvent += value;
      // }
      remove
      {
         MyEvent -= value;
      }

   }

   public static void Main()
   {
   }
}
```

Compiler Error CS0074

'*event*': abstract event cannot have initializer

If an event is marked as **abstract**, it cannot be initialized.

The following sample generates CS0074:

```
// CS0074.cs
delegate void D();

abstract class Test
{
   public abstract event D e = null;   // CS0074
   // try the following line instead
   // public abstract event D e;

   public static void Main()
   {
   }
}
```

Compiler Error CS0075

To cast a negative value, you must enclose the value in parentheses

Values that you want to cast must be in parentheses.

The following sample generates CS0075:

```
// CS0075.cs
using System;

enum Myint : long
{
   three = 3L,
}

class test
{
   public static void Main(String [] args )
   {
      Myint i = (Myint) -1 ;   // CS0075
      // try the following line instead
      // Myint i = (Myint) (-1) ;
      i++;
   }
}
```

Compiler Error CS0076

The enumerator name 'value__' is reserved and cannot be used

An item in an enumeration cannot have an identifier called **value__**.

The following sample generates CS0076:

```
// CS0076.cs
using System;

enum Myint : long
{
   value__ = 3L   // CS0076, cannot use the name value__
}

class MyClass
{
   public static void Main()
   {
   }
}
```

Compiler Error CS0077

The as operator must be used with a reference type ('*type*' is a value type)

The as operator was passed a value type. Because **as** can return null, it can only be passed reference types.

The following sample generates CS0077:

```
// CS0077.cs
using System;

class C
{
}

struct S
{
}

class M
{
   public static void Main()
   {
      object o1, o2;
      C c;
      S s;

      o1 = new C();
      o2 = new S();

      s = o2 as S;  // CS0077, S is not a reference type.
      // try the following line instead
      // c = o1 as C;
   }
}
```

Compiler Warning (level 4) CS0078

The 'l' suffix is easily confused with the digit '1' -- use 'L' for clarity

The compiler warns when it detects a cast to long using a lowercase l instead of an uppercase L.

The following sample generates CS0078:

```
// CS0078.cs
// compile with: /W:4
class test {
   public static void TestL(long i)
   {
   }

   public static void TestL(int i)
   {
   }

   public static void Main()
   {
      TestL(25l);   // CS0078
      // try the following line instead
      // TestL(25L);
   }
}
```

Compiler Error CS0079

The event '*event*' can only appear on the left hand side of += or -=

An event was called incorrectly.

The following sample generates CS0079:

```
// CS0079.cs
using System;

public delegate void MyEventHandler();

public class Class1
{
   private MyEventHandler _e;

   public event MyEventHandler Pow
   {
      add
      {
         _e += value;
         Console.WriteLine("in add accessor");
      }
      remove
      {
         _e -= value;
         Console.WriteLine("in remove accessor");
      }
   }

   public void Handler()
   {
   }

   public void Fire()
   {
      if (_e != null)
      {
         Pow();   // CS0079
         // try the following line instead
         // _e();
      }
   }

   public static void Main()
   {
      Class1 p = new Class1();
      p.Pow += new MyEventHandler(p.Handler);
      p._e();
      p.Pow += new MyEventHandler(p.Handler);
      p._e();
      p._e -= new MyEventHandler(p.Handler);
      if (p._e != null)
      {
         p._e();
      }
      p.Pow -= new MyEventHandler(p.Handler);
      if (p._e != null)
      {
         p._e();
      }
   }
}
```

Compiler Error CS0100

The parameter name '*parameter name*' is a duplicate

A method declaration used the same parameter name more than once. Parameter names must be unique in a method declaration.

The following sample generates CS0100:

```
// CS0100.cs
namespace x
{
   public class a
   {
      public static void f(int i, char i)   // CS0100
      // try the following line instead
      // public static void f(int i, char j)
      {
      }

      public static void Main()
      {
      }
   }
}
```

Compiler Error CS0101

The namespace '*namespace*' already contains a definition for '*type*'

A namespace has duplicate identifiers. Rename or delete one of the duplicate identifiers.

The following sample generates CS0101:

```
// CS0101.cs
namespace x
{
   public class clx
   {
      static public void Main()
      {
      }
   }

   public class clx   // CS0101
   {
   }
}
```

Compiler Error CS0102

The class '*class*' already contains a definition for '*identifier*'

A class contains multiple declarations for identifiers with the same name; delete or rename one of the duplicate identifiers.

The following sample generates CS0102:

```
// CS0102.cs
namespace x
{
   abstract public class clx
   {
      int i;

      enum splitch : int
      {
         x, x   // CS0102
      };

      public static int Main()
      {
         return 0;
      }
   }
}
```

Compiler Error CS0103

The name '*name*' does not exist in the class or namespace '*namespace*'

An attempt was made to use a name that does not exist in the class or the namespace.

The following sample generates CS0103:

```
// CS0103.cs
using System;

class MyClass
{
   public static void Main()
   {
      // MyClass test = null;
      try
      {
         MyClass conn = new MyClass();   // delete this line
         // and uncomment the following line and the line above the try
         // conn = new MyClass();
      }
      catch(Exception e)
      {
         if (conn != null)   // CS0103
            Console.WriteLine("{0}", e);
      }
   }
}
```

Compiler Error CS0104

'*reference*' is an ambiguous reference

Your program contains using directives for two namespaces and your code references a name that appears in both namespaces.

The following sample generates CS0104:

```
// CS0104.cs
using x;
using y;

namespace x
{
   public class Test
   {
   }
}

namespace y
{
   public class Test
   {
   }
}

public class a
{
   public static void Main()
   {
      Test test = new Test();   // CS0104, is Test in x or y namespace?
      // try the following line instead
      // y.Test test = new y.Test();
   }
}
```

Compiler Warning (level 3) CS0105

The using directive for '*namespace*' appeared previously in this namespace

A namespace, which should only be declared once, was declared more than once; remove all duplicate namespace declarations.

The following sample generates CS0105:

```
// CS0105.cs
// compile with: /W:3
using System;
using System;   // CS0105

public class a
{
   public static void Main()
   {
   }
}
```

Compiler Error CS0106

The modifier '*modifier*' is not valid for this item

A class or interface member was marked with an invalid access modifier. For example:

- The static and public modifiers are not permitted on interface methods.
- The **static** modifier is not permitted on a class.
- The **public** keyword is not allowed on an explicit interface declaration. In this case, remove the **public** keyword from the explicit interface declaration.
- The abstract keyword is not allowed on an explicit interface declaration because an explicit interface implementation could never be overridden.

The following sample generates CS0106:

```
// CS0106.cs
namespace MyNamespace
{
   interface I
   {
      void m();
      static public void f();   // CS0106
   }

   public class MyClass
   {
      public void I.m()   // CS0106
      {
      }

      public static void Main()
      {
      }
   }
}
```

Compiler Error CS0107

More than one protection modifier

Only one access modifier (public, private, protected, or internal) is allowed on a class member, with the exception of except **internal protected**.

The following sample generates CS0107:

```
// CS0107.cs
public class C
{
   private internal void f()   // CS0107, delete private or internal
   {
   }

   public static int Main()
   {
      return 1;
   }
}
```

Compiler Warning (level 2) CS0108

The keyword new is required on '*member1*' because it hides inherited member '*member2*'

A variable was declared with the same name as a variable in a base class. However, the new keyword was not used. This warning informs you that you should use **new**; the variable is declared as if **new** had been used in the declaration.

The following sample generates CS0108:

```
// CS0108.cs
// compile with: /W:2
using System;

namespace x
{
   public class clx
   {
      public int i = 1;
   }

   public class cly : clx
   {
      public static int i = 2;   // CS0108, use the new keyword
      // the compiler parses the previous line as if you had specified:
      // public static new int i = 2;

      public static void Main()
      {
         Console.WriteLine(i);
      }
   }
}
```

Compiler Warning (level 4) CS0109

The member '*member*' does not hide an inherited member. The new keyword is not required

A class declaration included the new keyword even though the declaration does not override an existing declaration in a base class. You can delete the **new** keyword.

The following sample generates CS0109:

```
// CS0109.cs
// compile with: /W:4
namespace x
{
   public class a
   {
      public int i;
   }

   public class b : a
   {
      public new int i;
      public new int j;   // CS0109
      public static void Main()
      {
      }
   }
}
```

Compiler Error CS0110

The evaluation of the constant value for '*const declaration*' involves a circular definition

The declaration of a const variable (`a`) cannot reference another const variable (`b`) that also references (`a`).

The following sample generates CS0110:

```
// CS0110.cs
namespace x
{
   public class a
   {
      public static void Main()
      {
      }
   }

   public class b : a
   {
      public const int i = c + 1;   // CS0110, c already references i
      public const int c = i + 1;
      // the following line would be OK
      // public const int c = 10;
   }
}
```

Compiler Error CS0111

Class '*class*' already defines a member called '*member*' with the same parameter types

A class contains two identical member declarations.

The following sample generates CS0111:

```
// CS0111.cs
namespace x
{
   abstract public class clx
   {
      public abstract void f();
   }

   public class cly : clx
   {
      public void f()
      {
      }
      override public void f() // CS0111, duplicate declaration
      {
      }
      public static int Main()
      {
         return 0;
      }
   }
}
```

Compiler Error CS0112

A static member '*function*' cannot be marked as override, virtual or abstract

Any method declaration that uses the **override**, **virtual**, or **abstract** keyword cannot also use the **static** keyword.

For more information, see Modifiers.

The following sample generates CS0112:

```
// CS0112.cs
namespace x
{
   abstract public class clx
   {
      public abstract void f();
   }

   public class cly : clx
   {
      override public static void f()   // CS0112, remove static keyword
      {
      }

      public static int Main()
      {
         return 0;
      }
   }
}
```

Compiler Error CS0113

A member '*function*' marked as override cannot be marked as new or virtual

It is mutually exclusive to mark a method with the new and override keywords.

The following sample generates CS0113:

```
// CS0113.cs
namespace x
{
   abstract public class clx
   {
      public abstract void f();
   }

   public class cly : clx
   {
      override new public void f()   // CS0113, remove new keyword
      {
      }

      public static int Main()
      {
         return 0;
      }
   }
}
```

Compiler Warning (level 2) CS0114

'*function1*' hides inherited member '*function2*'. To make the current method override that implementation, add the override keyword. Otherwise add the new keyword.

A declaration in a class conflicts with a declaration in a base class such that the base class member will be hidden.

For more information, see base.

The following sample generates CS0114:

```
// CS0114.cs
// compile with: /W:2 /warnaserror
abstract public class clx
{
   public abstract void f();
}

public class cly : clx
{
   public void f() // CS0114, hides base class member
   // try the following line instead
   // override public void f()
   {
   }

   public static void Main()
   {
   }
}
```

Compiler Error CS0115

'*function*' : no suitable method found to override

A method was marked as an override, but the compiler found no method to override.

The following sample generates CS0115:

```
// CS0115.cs
namespace x
{
   abstract public class clu
   {
      public abstract int f();
   }

   abstract public class clx
   {
      public override int f()   // CS0115
      {
         return 0;
      }

      public static void Main()
      {
      }
   }
}
```

You can resolve CS0115 in one of two ways:

- Remove the **override** keyword from the method in class `clx`.
- Use `clu` as a base class for `clx`.

Compiler Error CS0116

A namespace does not directly contain members such as fields or methods

Inside a namespace, the compiler only accepts classes, structs, unions, enums, interfaces, and delegates.

The following sample generates CS0116:

```
// CS0116.cs
namespace x
{
   int xx;   // CS0116
}
```

Compiler Error CS0117

'*type*' does not contain a definition for '*function*'

A call was made to a method that does not exist for the data type. This can also happen when a class name and its enclosing namespace name are the same and when a qualified method is called.

The following sample generates CS0117:

```
// CS0117_1.cs
namespace x
{
   public class a
   {
      public static void Main()
      {
         int i;
         i = i.get();   // CS0117, no get method on an int
      }
   }
}
```

When using indexers, it is not necessary to use the **Item** specifier:

```
// CS0117_2.cs
using System;
using System.Collections;

class Test
{
   public static void Main()
   {
      ArrayList al = new ArrayList();
      al.Add( new Test() );
      al.Add( new Test() );
      Console.WriteLine("First Element is {0}", al.Item[0]);   // CS0117
      // try the following line instead
      // Console.WriteLine("First Element is {0}", al[0]);
   }
}
```

Compiler Error CS0118

'*construct1_name*' denotes a '*construct1*' where a '*construct2*' was expected

The compiler detected what was probably the instantiation of a class; however, the data type was a namespace.

The following sample generates CS0118:

```
// CS0118.cs
namespace x
{
   class a
   {
      x ix = new x();   // CS0118, x not a class
      public static void Main()
      {
      }
   }
}
```

Compiler Error CS0119

'*construct1_name*' denotes a '*construct1*' which is not valid in the given context

The compiler detected an unexpected construct. For example:

- A class constructor is not a valid test expression in a conditional statement.
- A class name was used instead of an instance name to refer to an array element.

The following sample generates CS0119:

```
// CS0119.cs
namespace a
{
   public class z
   {
      public z()
      {
      }

      public static void Main()
      {
         (z()) ? 0 : 1;  // CS0119, cannot use constructor in a conditional statement
         // the following line is a more acceptable conditional statement
         // 1 < 2 ? 0 : 1;
      }
   }
}
```

Compiler Error CS0120

An object reference is required for the nonstatic field, method, or property '*member*'

In order to use a non-static field, method, or property, you must first create an object instance.

The following sample generates CS0120:

```
// CS0120_1.cs
public class MyClass
{
   // Non-static field
   public int i;
   // Non-static method
   public void f(){}
   // Non-static property
   int Prop
   {
      get
      {
         return 1;
      }
   }

   public static void Main()
   {
      i = 10;   // CS0120
      f();   // CS0120
      int p = Prop;   // CS0120
      // try the following lines instead
      // MyClass mc = new MyClass();
      // mc.i = 10;
      // mc.f();
      // int p = mc.Prop;
   }
}
```

CS0120 will also be generated if there is a call to a non-static method from a static method, as follows:

```
// CS0120_2.cs
// CS0120 expected
using System;

public class MyClass
{
   public static void Main()
   {
      TestCall();   // CS0120
      // To call a non-static method from Main,
      // first create an instance of the class.
      // Use the following two lines instead:
      // MyClass anInstanceofMyClass = new MyClass();
      // anInstanceofMyClass.TestCall();
   }

   public void TestCall()
   {
   }
}
```

Similarly, a static method cannot call an instance method unless you explicitly give it an instance of the class, as follows:

```
// CS0120_3.cs
using System;
```

```
public class MyClass
{
   public static void Main()
   {
      do_it("Hello There");   // CS0120
   }

   private void do_it(string sText)
   // You could also add the keyword static to the method definition:
   // private static void do_it(string sText)
   {
      Console.WriteLine(sText);
   }
}
```

Compiler Error CS0121

The call is ambiguous between the following methods or properties: '*method1*' and '*method2*'

Due to implicit conversion, the compiler was not able to call one form of an overloaded method. You can resolve this error in the following ways:

- Specify the method parameters in such a way that implicit conversion does not take place.
- Remove all overloads for the method.
- Cast to proper type before calling method.

The following sample generates CS0121:

```
// CS0121.cs
public class C
{
   void f(int i, double d)
   {
   }

   void f(double d, int i)
   {
   }

   public static void Main()
   {
      C c = new C();

      c.f(1, 1);   // CS0121
      // try the following line instead
      // c.f(1, 1.0);
      // or
      // c.f(1.0, 1);
      // or
      // c.f(1, (double)1);   // cast and specify which method to call
   }
}
```

Compiler Error CS0122

'*member*' is inaccessible due to its protection level

The access modifier for a class member prevents accessing the member.

The following sample generates CS0122:

```
// CS0122.cs
public class clx
{
   void f()   // make public to resolve CS0122
   {
   }
}

public class clz
{
   public static int Main()
   {
      clx a = new clx();
      a.f();   // CS0122, f is private
      return 0;
   }
}
```

Compiler Error CS0123

Method '*method*' does not match delegate '*delegate*'

An attempt to create a delegate failed because the correct signature was not used. Instances of a delegate must be declared with the same signature of the delegate declaration.

You can resolve this error by adjusting either your method or delegate signature.

The following sample generates CS0123:

```
// CS0123.cs
delegate void D();
// try the following line instead
// delegate void D(int i);

public class C
{
   public static void f(int i)
   {
   }

   public static void Main()
   {
      D d = new D(f);   // CS0123
   }
}
```

Compiler Error CS0126

An object of a type convertible to '*type*' required

A return statement is present, but the statement does not return a value of the expected type.

The following sample generates CS0126:

```
// CS0126.cs
public class a
{
   public int i
   {
      set
      {
      }
      get
      {
         return;   // CS0126, specify a value to return
      }
   }
}
```

Compiler Error CS0127

Since '*function*' returns void, a return keyword must not be followed by an object expression

A method with a void return type cannot return a value.

The following sample generates CS0127:

```
// CS0127.cs
namespace x
{
   public class b
   {
      public int hiddenMember2
      {
         get
         {
            return 0;
         }
         set   // CS0127, set has an implicit void return type
         {
            return 0;   // remove return statement to resolve this CS0127
         }
      }

      public static void Main()
      {
      }
   }
}
```

Compiler Error CS0128

A local variable named '*variable*' is already defined in this scope

The compiler detected declarations of two local variables with the same name.

The following sample generates CS0128:

```
// CS0128.cs
namespace x
{
   public class a
   {
      public static void Main()
      {
         char i;
         int i;   // CS0128
      }
   }
}
```

Compiler Error CS0131

The left-hand side of an assignment must be a variable, property or indexer

An assignment requires an l-value; check your syntax.

The following sample generates CS0131:

```
// CS0131.cs
public class MyClass
{
   public int i = 0;
   public void MyMethod()
   {
      i++ = 1;   // CS0131
      // try the following line instead
      // i = 1;
   }

   public static void Main()
   {
   }
}
```

Compiler Error CS0132

'*constructor*' : a static constructor must be parameterless

A static constructor cannot be declared with one or more parameters.

The following sample generates CS0132:

```
// CS0132.cs
namespace x
{
   public class clx
   {
      public clx(int i)
      {
      }
   }

   public class cly : clx
   {
      static cly(int i)   // CS0132
      {
      }
   }
}
```

Compiler Error CS0133

The expression being assigned to '*variable*' must be constant

A const variable cannot take as its value an expression that is not constant.

The following sample generates CS0133:

```
// CS0133.cs
public class A
{
   public const int i = c;   // CS0133, c is not constant
   public static int c = i;
   // try the following line instead
   // public const int i = 6;

   public static void Main()
   {
   }
}
```

Compiler Error CS0134

Cannot use qualified namespace names in nested namespace declarations

A qualified namespace is not permitted in a namespace declaration.

The following sample generates CS0134:

```
// CS0134.cs
namespace MyNameSpace
{
   namespace MyNameSpace.DottedNamespace   // CS0134, qualified namespace
   // try the following line instead
   // namespace NotRoot
   {
      public class MyClass
      {
         public static void Main()
         {
         }
      }

      namespace DottedNamespace
      {
         public class MyClass2
         {
            public static void f1()
            {
               MyNameSpace.NotRoot.MyClass.Main();   // legal use of qualified namespace
            }
         }
      }
   }
}
```

Compiler Error CS0135

'*declaration1*' conflicts with the declaration '*declaration2*'

The compiler does not allow hiding names, which commonly leads to logic errors in your code.

The following sample generates CS0135:

```
// CS0135.cs
public class MyClass2
{
   public static int i = 0;

   public static void Main()
   {
      {
         int i = 4;
         i++;
      }
      i = 0;   // CS0135
   }
}
```

Compiler Error CS0136

A local variable named '*var*' cannot be declared in this scope because it would give a different meaning to '*var*', which is already used in a 'parent or current' scope to denote something else

A variable declaration hides another declaration that would otherwise be in scope. Rename the variable that is declared on the line that generated CS0136.

The following sample generates CS0136:

```
// CS0136.cs
namespace x
{
   public class a
   {
      public static void Main()
      {
         int i = 0;
         {
            char i = 'a';   // CS0136, hides int i
         }
         i++;
      }
   }
}
```

Compiler Error CS0138

A using namespace directive can only be applied to namespaces; '*class*' is a class not a namespace

A using directive can only take the name of a namespace as a parameter.

The following sample generates CS0138:

```
// CS0138.cs
using System.Object;   // CS0138
```

Compiler Error CS0139

No enclosing loop out of which to break or continue

A break or continue statement was encountered outside of a loop.

For more information, see Jump Statements.

The following sample generates CS0139 twice:

```
// CS0139.cs
namespace x
{
   public class a
   {
      public static void Main()
      {
         continue;  // CS0139
         break;     // CS0139
      }
   }
}
```

Compiler Error CS0140

The label '*label*' is a duplicate

A label with the same name appeared twice.

The following sample generates CS0140:

```
// CS0140.cs
namespace x
{
   public class a
   {
      public static void Main()
      {
         l1: int i = 0;
         l1: int j = 0;   // CS0140, comment this line to resolve
         goto l1;
      }
   }
}
```

Compiler Error CS0143

The type '*class*' has no constructors defined

There is no appropriate constructor available.

The following sample generates CS0143:

```
// CS0143.cs
class nmc
{
   static public void Main ()
   {
      double d = new double(4.5);   // CS0143
   }
}
```

Compiler Error CS0144

Cannot create an instance of the abstract class or interface '*interface*'

You cannot create an instance of an abstract class or an interface.

The following sample generates CS0144:

```
// CS0144.cs
interface ii
{
}

public class a
{
   public static void Main()
   {
      ii xx = new ii();   // CS0144
   }
}
```

Compiler Error CS0145

A const field requires a value to be provided

You must initialize a const variable.

The following sample generates CS0145:

```
// CS0145.cs
class x
{
   const int i;   // CS0145
   // try the following line instead
   // const int i = 0;

   public static void Main()
   {
   }
}
```

Compiler Error CS0146

Circular base class definition between '*class1*' and '*class2*'

The inheritance list for a class includes a direct or indirect reference to itself. A class cannot inherit from itself.

The following sample generates CS0146:

```
// CS0146.cs
namespace x
{
   public interface a
   {
   }

   public class b : a, c
   {
      public void Main()
      {
      }
   }

   public class c : b   // CS0146
   {
   }
}
```

Compiler Error CS0148

The delegate '*delegate*' does not have a valid constructor

You imported and used a managed program (one that uses the .NET Framework common language runtime) that was created with another compiler. That compiler allowed an ill-formed delegate constructor.

Compiler Error CS0149

Method name expected

When creating a delegate, specify a method.

The following sample generates CS0149:

```
// CS0149.cs
using System;

delegate string func(int i);

class a
{
   static func dt;   // class member-field of the declared delegate type

   public static void Main()
   {
      dt = new func(17.45);   // CS0149
      // try the following line instead
      // dt = new func(func2);
      x(dt);
   }

   public static string func2(int j)
   {
      Console.WriteLine(j);
      return j.ToString();
   }

   public static void x(func myFunc)
   {
      myFunc(8);
   }
}
```

Compiler Error CS0150

A constant value is expected

A variable was found where a constant was expected.

The following sample generates CS0150:

```
// CS0150.cs
namespace x
{
   public class a
   {
      public static void Main()
      {
         int i = 0;
         int j = 0;

         switch(i)
         {
            case j:   // CS0150, j is a variable int, not a constant int
            // try the following line instead
            // case 1:
         }
      }
   }
}
```

Compiler Error CS0151

A value of an integral type expected

A variable was used in a situation where an integral data type was required. This can happen when there is no conversion or if the available implicit conversions result in an ambiguous situation.

The following sample generates CS0151:

```
// CS0151.cs
public class iii
{
   public static implicit operator int (iii aa)
   {
      return 0;
   }

   public static implicit operator long (iii aa)
   {
      return 0;
   }

   public static void Main()
   {
      iii a = new iii();

      switch (a)   // CS0151, compiler cannot choose between int and long
      // casting is one way to resolve this error
      // switch ((int)a)
      {
         case 1:
            break;
      }
   }
}
```

Compiler Error CS0152

The label '*label*' already occurs in this switch statement

A label was repeated in a switch statement.

The following sample generates CS0152:

```
// CS0152.cs
namespace x
{
   public class a
   {
      public static void Main()
      {
         int i = 0;

         switch (i)
         {
            case 1:
               i++;
               return;

            case 1:   // CS0152, two case 1 statements
               i++;
               return;
         }
      }
   }
}
```

Compiler Error CS0153

A goto case is only valid inside a switch statement

An attempt was made to use switch syntax outside of a **switch** statement.

The following sample generates CS0153:

```
// CS0153.cs
public class a
{
   public static void Main()
   {
      goto case 5;   // CS0153
   }
}
```

Compiler Error CS0154

The property or indexer '*property*' cannot be used in this context because it lacks the get accessor

An attempt to use a property failed because no get accessor method was defined in the property.

The following sample generates CS0154:

```
// CS0154.cs
public class b
{
   public int i
   {
      set
      {
      }
      // uncomment the get method to resolve this error
      /*
      get
      {
         return 0;
      }
      */
   }
}

public class a
{
   public static void Main()
   {
      b bb = new b();
      int j = bb.i;   // CS0154, no get method
   }
}
```

Compiler Error CS0155

The type caught or thrown must be derived from System.Exception

An attempt was made to pass a data type that does not derive from **System.Exception** into a catch block. Only data types that derive from **System.Exception** can be passed into a **catch** block.

For more information, see Exception Handling Statements.

The following sample generates CS0155:

```
// CS0155.cs
using System;

namespace x
{
   public class b
   // try the following line instead
   // public class b : Exception
   {
   }

   public class a
   {
      public static void Main()
      {
         try
         {
         }

         catch(b)   // CS0155, resolves if you derive b from Exception
         {
         }
      }
   }
}
```

Compiler Error CS0156

A throw statement with no arguments is not allowed outside of a catch clause

A throw statement with no parameters can only appear in a **catch** clause that takes no parameters.

For more information, see Exception Handling Statements.

The following sample generates CS0156:

```
// CS0156.cs
using System;

namespace x
{
   public class b : Exception
   {
   }

   public class a
   {
      public static void Main()
      {
         try
         {
            throw;   // CS0156
         }

         catch(b)
         {
            throw;   // this throw is valid
         }
      }
   }
}
```

Compiler Error CS0157

Control cannot leave the body of a finally clause

All of the statements in a finally clause must execute.

For more information, see Exception Handling Statements.

The following sample generates CS0157:

```
// CS0157.cs
using System;
namespace x
{
   public class b : Exception
   {
   }

   public class a
   {
      public static void Main()
      {
         try
         {
         }

         finally
         {
            return;   // CS0157, cannot leave finally clause
         }
      }
   }
}
```

Compiler Error CS0158

The label '*label*' shadows another label by the same name in a contained scope

A label in an inner scope hides a label with the same name in an outer scope.

The following sample generates CS0158:

```
// CS0158.cs
namespace x
{
   public class a
   {
      public static void Main()
      {
         goto lab1;
         lab1:
         {
            lab1:
            goto lab1;   // CS0158
         }
      }
   }
}
```

Compiler Error CS0159

No such label '*label*' within the scope of the goto statement

The label referenced by the goto statement could not be found within the scope of the **goto** statement.

The following sample generates CS0159:

```
// CS0159.cs
public class Class1
{
   public static void Main()
   {
      int i = 0;

      switch (i)
      {
         case 1:
            goto case 3;   // CS0159, case 3 label does not exist
         case 2:
            break;
      }
      goto NOWHERE;   // CS0159, NOWHERE label does not exist
   }
}
```

Compiler Error CS0160

A previous catch clause already catches all exceptions of this or a super type ('*type*')

A series of **catch** statements needs to be in decreasing order of derivation. For example, the most derived objects must appear first.

For more information, see Exception Handling Statements.

The following sample generates CS0160:

```
// CS0160.cs
using System;

public class b : Exception
{
}

public class Class1
{
   public static void Main()
   {
      try
      {
      }

      catch(Exception)   // Second-most derived; should be second catch
      {
      }

      catch(b)           // CS0160, Most derived; should be first catch
      {
      }

      catch              // Must be last catch
      {
      }
   }
}
```

Compiler Error CS0161

'*method*': not all code paths return a value

A method that returns a value must have a **return** statement in all code paths.

The following sample generates CS0161:

```
// CS0161.cs
public class Test
{
   public static int Main() // CS0161
   {
      int i = 10;
      if (i < 10)
      {
         return i;
      }
      else
      {
         // uncomment the following line to resolve
         // return 1;
      }
   }
}
```

Compiler Warning (level 2) CS0162

Unreachable code detected

The compiler detected code that will never be executed.

The following sample generates CS0162:

```
// CS0162.cs
// compile with: /W:2
public class A
{
   public static void Main()
   {
      goto lab1;
      {
         // The following statements cannot be reached:
         int i = 9;   // CS0162
         i++;
      }
      lab1:
      {
      }
   }
}
```

Compiler Error CS0163

Control cannot fall through from one case label ('*label*') to another

When a case statement contains one or more statements and is followed by another **case** statement, one of the following statements must explicitly terminate the **case**:

- **return**
- **goto**
- **break**

If you want to implement "fall through" behavior, use `goto case #`.

The following sample generates CS0163:

```
// CS0163.cs
public class MyClass
{
   public static void Main()
   {
      int i = 0;

      switch (i)   // CS0163
      {
         case 1:
            i++;
            // uncomment one of the following lines to resolve
            // return;
            // break;
            // goto case 3;

         case 2:
            i++;
            return;

         case 3:
            i = 0;
            return;
      }
   }
}
```

Compiler Warning (level 2) CS0164

This label has not been referenced

A label was declared but never used.

The following sample generates CS0164:

```
// CS0164.cs
// compile with: /W:2
public class a
{
   public int i = 0;

   public static void Main()
   {
      int i = 0;   // CS0164
      l1: i++;
      // the following lines resolve this error
      // if(i < 10)
      //    goto l1;
   }
}
```

Compiler Error CS0165

Use of unassigned local variable '*var*'

The C# compiler does not allow the use of uninitialized variables. If the compiler detects the use of a variable that might not have been initialized, it generates CS0165.

Use new to create an instance of an object or assign a value.

The following sample generates CS0165:

```
// CS0165.cs
using System;

class MyClass
{
   public int i;
}

class MyClass2
{
   public static void Main(string [] args)
   {
      int i, j;
      if (args[0] == "test")
      {
         i = 0;
      }

      /*
      // to resolve, either initialize the variables when declared
      // or provide for logic to initialize them, as follows:
      else
      {
         i = 1;
      }
      */

      j = i;   // CS0165, i might be uninitialized

      MyClass x;
      x.i = 0;   // CS0165
      // use new as follows
      // MyClass x = new MyClass();
      // x.i = 0;
   }
}
```

Compiler Error CS0167

The delegate '*delegate*' is missing the Invoke method

You imported and used a managed program (one that uses the .NET Framework common language runtime) that was created with another compiler. That compiler allowed an ill-formed delegate. Therefore, the **Invoke** method was not available.

Compiler Warning (level 3) CS0168

The variable '*var*' is declared but never used

The compiler warns when a variable is declared but not used.

The following sample generates two CS0168 warnings:

```
// CS0168.cs
// compile with: /W:3
public class clx
{
   public int i;
}

public class clz
{
   public static void Main()
   {
      int j = 0;   // CS0168, uncomment the following line
      // j++;
      clx a;       // CS0168, try the following line instead
      // clx a = new clx();
   }
}
```

Compiler Warning (level 3) CS0169

The private field '*class member*' is never used

A private variable was declared but never referenced. A common way to generate this warning is when you declare a private member of a class and do not use it.

The following sample generates CS0169:

```
// compile with: /W:3
using System;
public class ClassX
{
   int i;   // CS0169, i is not used anywhere

   public static void Main()
   {
   }
}
```

Compiler Error CS0170

Use of possibly unassigned field '*field*'

A field in a structure was used without first being initialized.

The following sample generates CS0170:

```
// CS0170.cs
public struct error
{
   public int i;
}

public class CmyClass
{
   public static void Main()
   {
      error e;
      // uncomment the next line to resolve this error
      // e.i = 0;
      System.Console.WriteLine( e.i );   // CS0170
   }
}
```

Compiler Error CS0171

Field '*field*' must be fully assigned before control leaves the constructor

A constructor in a struct must initialize all fields in the struct.

The following sample generates CS0171:

```
// CS0171.cs
struct MyStruct
{
   MyStruct(int initField)   // CS0171
   {
      // i = initField;      // uncomment this line to resolve this error
   }
   public int i;
}

class MyClass
{
   public static void Main()
   {
      MyStruct aStruct = new MyStruct();
   }
}
```

Compiler Error CS0172

Type of conditional expression can't be determined because '*type1*' and '*type2*' both implicitly convert to each other

In a conditional statement, you must be able to convert the types on either side of the `:` operator. Also, there cannot be mutual conversion routines; you only need one conversion.

The following sample generates CS0172:

```
// CS0172.cs
public class Square
{
   public class Circle
   {
      public static implicit operator Circle(Square aa)
      {
         return null;
      }

      public static implicit operator Square(Circle aa)
      // using explicit resolves this error
      // public static explicit operator Square(Circle aa)
      {
         return null;
      }
   }

   public static void Main()
   {
      Circle aa = new Circle();
      Square ii = new Square();
      object o = (1 == 1) ? aa : ii;   // CS0172
      // the following cast would resolve this error
      // (1 == 1) ? aa : (Circle)ii;
   }
}
```

Compiler Error CS0173

Type of conditional expression can't be determined because there is no implicit conversion between '*class1*' and '*class2*'

Conversions between classes are useful when you want objects of different classes to work with the same code. However, two classes that will work together cannot have mutual and redundant conversions.

To resolve CS0173, verify that there is one and only one implicit conversion between *class1* and *class2*, regardless of which direction the conversion is in and regardless of which class the conversion is in.

The following sample generates CS0173:

```
// CS0173.cs
// To resolve, mark three of the four conversion routines as explicit
// or remove three of the conversion routines.
public class iii
{
   // a conversion routine
   public static implicit operator a(iii aa)
   {
      return new a();
   }

   // a conversion routine
   public static implicit operator iii(a aa)
   {
      return new iii();
   }

   public static void Main()
   {
      a aa = new a();
      iii ii = new iii();
      a aaa = (1 == 1) ? aa : ii;   // CS0173
   }
}

public class a
{
   // a conversion routine
   public static implicit operator a(iii aa)
   {
      return new a();
   }

   // a conversion routine
   public static implicit operator iii(a aa)
   {
      return new iii();
   }
}
```

Compiler Error CS0174

A base class is required for a base reference

This error occurs only when the .NET Framework common language runtime compiles the source code for the **System.Object** class, which is the only class that has no base class.

Compiler Error CS0175

Use of keyword base is not valid in this context

The base keyword must be used to specify a particular member of the base class.

The following sample generates CS0175:

```
// CS0175.cs
using System;
class BaseClass
{
   public int TestInt = 0;
}

class MyClass : BaseClass
{
   public static void Main()
   {
      MyClass aClass = new MyClass();
      aClass.BaseTest();
   }

   public void BaseTest()
   {
      Console.WriteLine(base);   // CS0175
      // try the following line instead
      // Console.WriteLine(base.TestInt);
      base = 9;   // CS0175
      // try the following line instead
      // base.TestInt = 9;
   }
}
```

Compiler Error CS0176

Static member '*member*' cannot be accessed with an instance reference; qualify it with a type name instead

Only a class name can be used to qualify a static variable; an instance name cannot be a qualifier.

The following sample generates CS0176:

```
// CS0176.cs
public class b
{
   public static int ii;
}

public class a
{
   public static void Main()
   {
      b bb = new b();
      int i = bb.ii;   // CS0176
      // try the following line instead
      // int i = b.ii;
   }
}
```

Compiler Error CS0177

The out parameter '*parameter*' must be assigned to before control leaves the current method

A parameter marked with the out keyword was not assigned a value in the method body.

The following sample generates CS0177:

```
// CS0177.cs
public class iii
{
   public void f(out int i)   // CS0177
   {
   // uncomment the following line to resolve this error
   //   i = 0;
   }

   public static void Main()
   {
   }
}
```

Compiler Error CS0178

Incorrectly structured array initializer

An array initialization was ill-formed. When specifying the array dimensions, you can specify the following:

- A number in brackets
- Empty brackets
- A comma enclosed in brackets

The following sample generates CS0178:

```
// CS0178.cs
class x
{
   public static void Main()
   {
      a = new int[5][,][][5;   // CS0178, no comma or right bracket
   }
}
```

Compiler Error CS0179

'*member*' cannot be extern and declare a body

When a class member is marked extern, it means that the member's definition is located in another file. Therefore, a class member marked as **extern** cannot be defined in the class. Either remove the **extern** keyword or delete the definition.

The following sample generates CS0179:

```
// CS0179.cs
public class MyClass
{
   public extern int ExternMethod(int aa)   // CS0179
   {
      return 0;
   }
   // try the following line instead
   // public extern int ExternMethod(int aa);

   public static void Main()
   {
   }
}
```

Compiler Error CS0180

'*member*' cannot be both extern and abstract

The abstract and extern keywords are mutually exclusive. The **extern** keyword means that the member is defined outside the file, and **abstract** means that the implementation is provided in a derived class.

The following sample generates CS0180:

```
// CS0180.cs
namespace x
{
   public class iii
   {
      public extern abstract int aaa(int aa);   // CS0180

      public static void Main()
      {
      }
   }
}
```

Compiler Error CS0182

An attribute argument must be a constant expression, typeof expression or array creation expression

An argument to an attribute was not specified correctly.

The following sample generates CS0182:

```
// CS0182.cs
public class C
{
   static string s = "Test";

   [System.Diagnostics.ConditionalAttribute(s)]   // CS0182
   // try the following line instead
   // [System.Diagnostics.ConditionalAttribute("Test")]
   void nonConstantArgumentToConditional()
   {
   }

   public static void Main()
   {
   }
}
```

Compiler Warning (level 1) CS0183

The given expression is always of the provided ('*type*') type

If a conditional statement always evaluates to **true**, then you do not need a conditional statement.

The following sample generates CS0183:

```
// CS0183.cs
// compile with: /W:1
using System;
public class Test
{
   public static void Main()
   {
      Object obj3 = null;
      String str2 = "CS0183";
      obj3 = str2;

      if ( obj3 is String )
         Console.WriteLine( "obj3 is String" );

      if ( str2 is Object )   // CS0183
         Console.WriteLine( "str2 is Object" );
   }
}
```

Compiler Warning (level 1) CS0184

The given expression is never of the provided ('*type*') type

The expression can never be **true** because the variable you are testing is neither declared as ***type*** nor derived from ***type***.

The following sample generates CS0184:

```
// CS0184.cs
// compile with: /W:1
class MyClass
{
   public static void Main()
   {
      int i = 0;
      if (i is string)   // CS0184
         i++;
   }
}
```

Compiler Error CS0185

'*type*' is not a reference type as required by the lock statement

The lock statement can only evaluate reference types.

The following sample generates CS0185:

```
// CS0185.cs
public class MainClass
{
   public static void Main ()
   {
      lock (1)   // CS0185
      // try the following lines instead
      // MainClass x = new MainClass();
      // lock(x)
      {
      }
   }
}
```

See Also

Reference Types

Compiler Error CS0186

Use of null is not valid in this context

The following sample generates CS0186:

```
// CS0186.cs
using System;

class MyClass
{
   static void Main()
   {
      // Each of the following lines generates CS0186:
      if ((object)null is string) {}   // CS0186
      foreach (int i in null) {}   // CS0186
      string s = (object)null as string;   // CS0186
   }
}
```

Compiler Error CS0187

No such operator '*operator*' defined for type '*type*'

An operator was used on a type, but was not defined to do so.

The following sample generates CS0187:

```
// CS0187.cs
class MyClass
{
   public static void Main()
   {
      string someString = "Hello";
      someString++;   // CS0187
   }
}
```

Compiler Error CS0188

The this object cannot be used before all of its fields are assigned to

All fields in a struct have to be assigned by a constructor before the constructor can call a method in the struct.

The following sample generates CS0188:

```
// CS0188.cs
struct s
{
   public int a;

   void f()
   {
   }

   s (int i)
   {
      // a = i;
      f();  // CS0188
   }
}

class MyClass
{
   public static void Main()
   {
   }
}
```

Compiler Error CS0191

A readonly field cannot be assigned to (except in a constructor or a variable initializer)

A readonly field can only take an assignment in a constructor or at declaration.

The following sample generates CS0191:

```
// CS0191.cs
class MyClass
{
   public readonly int TestInt = 6;  // OK to assign to readonly field in declaration

   MyClass()
   {
      TestInt = 11;                  // OK to assign to readonly field in constructor
   }

   public void TestReadOnly()
   {
      TestInt = 19;                  // CS0191
   }

   public static void Main()
   {
   }
}
```

CS0191 also results if the **readonly** field is static and the constructor is not marked **static**.

Compiler Error CS0192

A readonly field cannot be passed ref or out (except in a constructor)

A field (variable) marked with the readonly keyword cannot be passed either to a ref or out parameter except inside a constructor.

The following sample generates CS0192:

```
// CS0192.cs
class MyClass
{
   public readonly int TestInt = 6;

   static void TestMethod(ref int testInt)
   {
      testInt = 0;
   }

   MyClass()
   {
      TestMethod(ref TestInt);   // OK
   }

   public void PassReadOnlyRef()
   {
      TestMethod(ref TestInt);   // CS0192
   }

   public static void Main()
   {
   }
}
```

CS0192 also results if the **readonly** field is static and the constructor is not marked **static**.

Compiler Error CS0193

The * or -> operator must be applied to a pointer

The ***** or **->** operator was used with a nonpointer type.

The following sample generates CS0193:

```
// CS0193.cs
using System;

public struct Age
{
   public int AgeYears;
   public int AgeMonths;
   public int AgeDays;
}

public class MyClass
{
   public static void SetAge(ref Age anAge, int years, int months, int days)
   {
      anAge->Months = 3;   // CS0193, anAge is not a pointer
      // try the following line instead
      // anAge.AgeMonths = 3;
   }

   public static void Main()
   {
      Age MyAge = new Age();
      Console.WriteLine(MyAge.AgeMonths);
      SetAge(ref MyAge, 22, 4, 15);
      Console.WriteLine(MyAge.AgeMonths);
   }
}
```

Compiler Error CS0196

A pointer must be indexed by only one value

A pointer cannot have multiple indices.

The following sample generates CS0196:

```
// CS0196.cs
public class MainClass
{
   public static void Main ()
   {
      int *i = null;
      int j = 0;
      j = i[1,2];   // CS0196
      // try the following line instead
      // j = i[1];
   }
}
```

Compiler Error CS0197

Cannot pass '*argument*' as ref or out, because ' *argument* ' is a marshal-by-reference class

Any class that derives (directly or indirectly) from **System.MarshalByRefObject** is a "marshal-by-reference" class. Such a class can be marshaled by reference across process and machine boundaries. Thus, instances of this class could be proxies to remote objects. You cannot pass a field of a proxy object as ref or out. So, you cannot pass fields of such a class as **ref** or **out**, unless the instance is this, which can never be a proxy object.

The following sample generates CS0197:

```
// CS0197a.cs
using System;
class X : MarshalByRefObject
{
   public int i;
}

class M
{
   public int i;
   static void AddSeventeen(ref int i)
   {
      i += 17;
   }

   static void Main()
   {
      X x = new X();
      x.i = 12;
      AddSeventeen(ref x.i);   // CS0197
      // try the following lines instead
      // M m = new M();
      // m.i = 12;
      // AddSeventeen(ref m.i);
   }
}
```

The following sample generates CS0197 when you try to pass the structure **System.Drawing.Point** as a reference:

```
// CS0197b.cs
public class A : System.MarshalByRefObject
{
   public System.Drawing.Point point;
}

public class B
{
   public void Notify()
   {
      A a = new A();
      if ( !a.point.IsEmpty )   // CS0197
      // try the following line instead
      // if(!((System.Drawing.Point)((object)(a.point))).IsEmpty)
         {
         }
   }
   static void Main()
   {
   }
}
```

Compiler Error CS0198

A static readonly field cannot be assigned to (except in a static constructor or a variable initializer)

A readonly variable must have the same static usage as the constructor in which you want to initialize it.

The following sample generates CS0198:

```
// CS0198.cs
class MyClass
{
   public static readonly int TestInt = 6;

   MyClass()
   {
      TestInt = 11;   // CS0198, constructor is not static and readonly field is
   }

   public static void Main()
   {
   }
}
```

Compiler Error CS0199

A static readonly field cannot be passed ref or out (except in a static constructor)

A readonly variable must have the same static usage as the constructor in which you want to pass it as a ref or out parameter.

The following sample generates CS0199:

```
// CS0199.cs
class MyClass
{
   public static readonly int TestInt = 6;

   static void TestMethod(ref int testInt)
   {
      testInt = 0;
   }

   MyClass()
   {
      TestMethod(ref TestInt);   // CS0199, TestInt is static
   }

   public static void Main()
   {
   }
}
```

Compiler Error CS0200

Property or indexer '*property*' cannot be assigned to — it is read only

An attempt was made to assign a value to a property, but the property does not have a set accessor. Resolve the error by adding a set accessor.

The following sample generates CS0200:

```
// CS0200.cs
public class MainClass
{
   // private int _mi;
   int I
   {
      get
      {
         return 1;
      }

      // uncomment the set accessor and declaration for _mi
      /*
      set
      {
         _mi = value;
      }
      */
   }

   public static void Main ()
   {
      MainClass II = new MainClass();
      II.I = 9;   // CS0200
   }
}
```

Compiler Error CS0201

Only assignment, call, increment, decrement, and new object expressions can be used as a statement

The compiler generates an error when it encounters a meaningless statement.

The following sample generates CS0201:

```
// CS0201.cs
using System;

public class MainClass
{
   public static void Main ()
   {
      2 * 3;   // CS0201
   }
}
```

Compiler Error CS0202

The call to GetEnumerator must return a class or a struct, not '*type*'

A call to **GetEnumerator** cannot return a pointer; it must return an instance.

The following sample generates CS0202:

```
// CS0202.cs
// compile with: /unsafe

public class C1
{
   public int Current
   {
      get
      {
         return 0;
      }
   }

   public bool MoveNext ()
   {
      return false;
   }

   public static implicit operator C1 (int c1)
   {
      return 0;
   }
}

public class C2
{
   public int Current
   {
      get
      {
         return 0;
      }
   }

   public bool MoveNext ()
   {
      return false;
   }

   public C1* GetEnumerator ()
   // try the following line instead
   // public C1 GetEnumerator ()
   {
      return null;
   }
}

public class MainClass
{
   unsafe public static void Main ()
   {
      C2 c2 = new C2();

      foreach (C1 x in c2)   // CS0202
      {
         System.Console.WriteLine(x.Current);
      }
   }
}
```

Compiler Error CS0204

Only 65535 locals are allowed

The limit of 65535 local variables has been exceeded in a method.

Compiler Error CS0205

Cannot call an abstract base member: '*method*'

You cannot call an abstract method, even from within the implementation of the method.

The following sample generates CS0205:

```
// CS0205.cs
abstract public class MyClass
{
   abstract public void mf();
}

public class MyClass2 : MyClass
{
   public override void mf()
   {
      base.mf();   // CS0205, delete this line
   }

   public static void Main()
   {
   }
}
```

Compiler Error CS0206

A property or indexer may not be passed as an out or ref parameter

A property is not available to be passed as a ref or out parameter.

The following sample generates CS0206:

```
// CS0206.cs
public class MyClass
{
   public static int iii
   {
      get
      {
         return 0;
      }
      set
      {
      }
   }

   public static void MyMeth(ref int i)
   // public static void MyMeth(int i)
   {
   }

   public static void Main()
   {
      MyMeth(ref iii);   // CS0206
      // try the following line instead
      // MyMeth(iii);   // CS0206
   }
}
```

Compiler Error CS0208

Cannot take the address or size of a variable of a managed type ('S')

Even when used with the unsafe keyword, taking the address of a managed object is not allowed. For more information on managed objects, see A.2 Pointer types.

The following sample generates CS0208:

```
// CS0208.cs
// compile with: /unsafe

class S
{
   public int a = 98;
}

public class a
{
   unsafe public static int Main()
   {
      S s = new S();  // S is managed
      S * s2 = &s;    // CS0208
      return 1;
   }
}
```

Compiler Error CS0209

The type of locals declared in a fixed statement must be a pointer type

The variable that you declare in a fixed statement must be a pointer.

The following sample generates CS0209:

```
// CS0209.cs
// compile with: /unsafe

class Point
{
   public int x, y;
}

public class Class
{
   unsafe public static void Main()
   {
      Point pt = new Point();

      fixed (int i)    // CS0209
      {
      }
      // try the following lines instead
      /*
      fixed (int* p = &pt.x)
      {
      }
      fixed (int* q = &pt.y)
      {
      }
      */
   }
}
```

Compiler Error CS0210

You must provide an initializer in a fixed or using statement declaration

You must declare and initialize the variable in a fixed statement.

The following sample generates CS0210:

```
// CS0210a.cs
// compile with: /unsafe

class Point
{
   public int x, y;
}

public class Class
{
   unsafe public static void Main()
   {
      Point pt = new Point();

      fixed (int i)    // CS0210
      {
      }
      // try the following lines instead
      /*
      fixed (int* p = &pt.x)
      {
      }
      fixed (int* q = &pt.y)
      {
      }
      */
   }
}
```

The following sample also generates CS0210 because the using statement has no initializer.

```
// CS0210b.cs

using System.IO;
class Test
{
   static void Main()
   {
      using (StreamWriter w) // CS0210
      // Try this line instead:
      // using (StreamWriter w = new StreamWriter("TestFile.txt"))
      {
         w.WriteLine("Hello there");
      }
   }
}
```

Compiler Error CS0211

Cannot take the address of the given expression

You can take the address of fields, local variables, and indirection of pointers, but you cannot take, for example, the address of the sum of two local variables.

The following sample generates CS0211:

```
// CS0211.cs
// compile with: /unsafe

public class a
{
   unsafe public void mf()
   {
      int a = 0, b = 0;
      int *i = &(a + b);   // CS0211, the addition of two local variables
      // try the following line instead
      // int *i = &a;
   }

   public static void Main()
   {
   }
}
```

Compiler Error CS0212

You can only take the address of unfixed expression inside of a fixed statement initializer

The following sample generates CS0212 and shows how to take the address of an unfixed expression:

```
// CS0212a.cs
// compile with: /unsafe

public class A {
    public int iField = 5;

    unsafe public void mf() {
        A a = new A();
        int* ptr = &a.iField;   // CS0212

        // Try the following code instead:
        /*
        fixed (int* ptr = &a.iField)
        {
        }
        */
    }
    static void Main()
    {
    }
}
```

The following sample also generates CS0212 and shows how to resolve the error:

```
// CS0212b.cs
// compile with: /unsafe
using System;

public class MyClass
{
   unsafe public void mf()
   {
      // Null-terminated ASCII characters in an sbyte array
      sbyte[] sbArr1 = new sbyte[] { 0x41, 0x42, 0x43, 0x00 };
      sbyte* pAsciiUpper = &sbArr1[0];   // CS0212
      // To resolve this error, delete the previous line and
      // uncomment the following code:
      // fixed (sbyte* pAsciiUpper = sbArr1)
      // {
      //    String szAsciiUpper = new String(pAsciiUpper);
      // }
   }

   public static void Main()
   {
   }
}
```

Compiler Error CS0213

You do not need to use the fixed statement to take the address of an already fixed expression

A local variable in an unsafe method or a parameter is already fixed (on the stack), so you cannot take the address of either of these two variables in a fixed expression.

The following sample generates CS0213:

```
// CS0213.cs
// compile with: /unsafe
public class MyClass
{
   unsafe public static void Main()
   {
      int i = 45;
      fixed (int *j = &i)   // CS0213
      {
      }

      // try the following line instead
      // int* j = &i;
   }
}
```

Compiler Error CS0214

Pointers may only be used in an unsafe context

Pointers can only be used with the unsafe keyword.

The following sample generates CS0214:

```
// CS0214.cs
public struct S
{
   public int a;
}

public class a
{
   public static int Main()
   // try the following line instead
   // unsafe public static int Main()
   {
      S s = new S();
      S * s2 = &s;    // CS0214
      s2->a = 3;      // CS0214

      return 1;
   }
}
```

Compiler Error CS0215

The return type of operator True or False must be bool

User-defined true and false operators must have a return type of bool.

The following sample generates CS0215:

```
// CS0215.cs
class MyClass
{
   public static int operator true (MyClass MyInt)   // CS0215
   // try the following line instead
   // public static bool operator true (MyClass MyInt)
   {
      return true;
   }

   public static int operator false (MyClass MyInt)   // CS0215
   // try the following line instead
   // public static bool operator false (MyClass MyInt)
   {
      return true;
   }

   public static void Main()
   {
   }
}
```

Compiler Error CS0216

The operator '*operator*' requires a matching operator '*missing_operator*' to also be defined

A user-defined true operator requires a user-defined false operator, and vice versa.

The following sample generates CS0216:

```
// CS0216.cs
class MyClass
{
   public static bool operator true (MyClass MyInt)   // CS0216
   {
      return true;
   }

   // to resolve, uncomment the following operator definition
   /*
   public static bool operator false (MyClass MyInt)
   {
      return true;
   }
   */

   public static void Main()
   {
   }
}
```

Compiler Error CS0217

In order to be applicable as a short circuit operator a user-defined logical operator ('*operator*') must have the same return type as the type of its 2 parameters.

If you define an operator for a user-defined type, and then try to use the operator as a short-circuit operator, the user-defined operator must have parameters and return values of the same type. For more information on short-circuit operators, see && Operator and || Operator.

The following sample generates CS0217:

```
// CS0217.cs
using System;

public class MyClass
{
   public static bool operator true (MyClass f)
   {
      return false;
   }

   public static bool operator false (MyClass f)
   {
      return false;
   }

   public static implicit operator int(MyClass x)
   {
      return 0;
   }

   public static int operator & (MyClass f1, MyClass f2)   // CS0217
   // try the following line instead
   // public static MyClass operator & (MyClass f1, MyClass f2)
   {
      return new MyClass();
   }

   public static void Main()
   {
      MyClass f = new MyClass();
      int i = f && f;
   }
}
```

Compiler Error CS0218

The type ('*type*') must contain declarations of operator true and operator false

If you define an operator for a user-defined type, and then try to use the operator as a short-circuit operator, the user-defined operator must have operator true and operator false defined. For more information on short-circuit operators, see && Operator and || Operator.

The following sample generates CS0218:

```
// CS0218.cs
using System;
public class MyClass
{
   // uncomment these operator declarations to resolve this CS0218
   /*
   public static bool operator true (MyClass f)
   {
      return false;
   }

   public static bool operator false (MyClass f)
   {
      return false;
   }
   */

   public static implicit operator int(MyClass x)
   {
      return 0;
   }

   public static MyClass operator & (MyClass f1, MyClass f2)
   {
      return new MyClass();
   }

   public static void Main()
   {
      MyClass f = new MyClass();
      int i = f && f;   // CS0218, requires operators true and false
   }
}
```

Compiler Warning (level 3) CS0219

The variable '*variable*' is assigned but never used

The variable '*variable*' is assigned but its value is never used

The compiler issues a level-three warning, when you declare and assign a variable, but do not use it.

The following sample generates CS0219:

```
// CS0219.cs
// compile with: /W:3
public class MyClass
{
   public static void Main()
   {
      int a = 0;   // CS0219
   }
}
```

Compiler Error CS0220

The operation overflows at compile time in checked mode

An operation was detected by checked, which is the default, and a data loss resulted. Either correct the inputs to the assignment or use unchecked to resolve this error.

The following sample generates CS0220:

```
// CS0220.cs
using System;

class TestClass
{
   const int x = 1000000;
   const int y = 1000000;

   public int MethodCh()
   {
      int z = (x * y);   // CS0220
      return z;
   }

   public int MethodUnCh()
   {
      unchecked
      {
         int z = (x * y);
         return z;
      }
   }

   public static void Main()
   {
      TestClass myObject = new TestClass();
      Console.WriteLine("Checked  : {0}", myObject.MethodCh());
      Console.WriteLine("Unchecked: {0}", myObject.MethodUnCh());
   }
}
```

Compiler Error CS0221

Constant value '*value*' cannot be converted to a '*type*' (use 'unchecked' syntax to override)

An assignment operation that would result in a data loss was detected by checked, which is on by default. Either correct the assignment or use unchecked to resolve this error.

The following sample generates CS0221:

```
// CS0221.cs
public class MyClass
{
   public static void Main()
   {
      // unchecked
      // {
         int a = (int)0xFFFFFFFF;   // CS0221
         a++;
      // }
   }
}
```

Compiler Error CS0225

The params parameter must be a single dimensional array

When using the params keyword, you must specify a single-dimensional array of the data type.

The following sample generates CS0225:

```
// CS0225.cs
public class MyClass
{
   public static void TestParams(params int a)   // CS0225
   // try the following line instead
   // public static void TestParams(params int[] a)
   {
   }

   public static void Main()
   {
      TestParams(1);
   }
}
```

Compiler Error CS0227

Unsafe code may only appear if compiling with /unsafe

If source code contains the unsafe keyword, then the /unsafe compiler option must also be used.

The following sample, when compiled without **/unsafe**, will generate CS0227:

```
// CS0227.cs
public class a
{
   unsafe public static void Main()   // CS0227
   {
   }
}
```

Compiler Error CS0228

'*type*' does not contain a definition for '*member*', or it is not accessible

You may have replaced the system version of **System.Hashtable**, **System.String**, or **System.Array**, and the replacement does not contain *member*.

Otherwise, you may need to reinstall Visual Studio.

Compiler Error CS0229

Ambiguity between '*member1*' and '*member2*'

Members of different interfaces have the same name. If you want to keep the same names, you must qualify the names.

The following sample generates CS0229:

```
// CS0229.cs
interface IList
{
   int Count
   {
      get;
      set;
   }
}

interface Icounter
{
   void Count(int i);
}

interface IListCounter : IList , Icounter
{
}

class main
{
   void Test(IListCounter x)
   {
      x.Count = 1;  // CS0229
      x.Count(1);   // CS0229
      // try the following lines instead
      // ((IList)x).Count = 1;
      // ((ICounter)x).Count(1);
   }

   public static void Main()
   {
   }
}
```

Compiler Error CS0230

Type and identifier are both required in a foreach statement

A foreach statement was poorly formed.

The following sample generates CS0230:

```
// CS0230.cs
using System;

class MyClass
{
   public static void Main()
   {
      int[] myarray = new int[3] {1,2,3};

      foreach (int in myarray)   // CS0230
      // try the following line instead
      // foreach (int x in myarray)
      {
         Console.WriteLine(x);
      }
   }
}
```

Compiler Error CS0231

A params parameter must be the last parameter in a formal parameter list.

The following sample generates CS0231:

```
// CS0231.cs
class Test
{
   public void TestMethod(params int[] p, int i) {} // CS0231
   // To resolve the error, use the following line instead:
   // public void TestMethod(int i, params int[] p) {}

   static void Main()
   {
   }
}
```

Compiler Error CS0233

sizeof can only be used in an unsafe context (consider using System.Runtime.InteropServices.Marshal.SizeOf)

The sizeof operator can only be used in an unsafe block.

The following sample generates CS0233:

```
// CS0233.cs
using System;
using System.Runtime.InteropServices;

[StructLayout(LayoutKind.Sequential)]
public struct S
{
   public int a;
}

public class MyClass
{
   public static void Main()
   {
      S myS = new S();
      Console.WriteLine(sizeof(myS));   // CS0233
      // try the following line instead
      // Console.WriteLine(Marshal.SizeOf(myS));
   }
}
```

Compiler Error CS0234

The type or namespace name '*name*' does not exist in the class or namespace '*scope*' (are you missing an assembly reference?)

A type was expected. Possible reasons include:

- An assembly that contains the definition of a type was not referenced in the compilation; use /reference (Import Metadata) to specify the assembly.
- You passed a variable name to the typeof operator.

See Add Reference Dialog Box for information on how to add a reference in the development environment.

The following sample generates CS0234:

```
// CS0234.cs
public class C
{
   public static void Main()
   {
      System.DateTime x = new System.DateTim();   // CS0234
      // try the following line instead
      // System.DateTime x = new System.DateTime();
   }
}
```

Compiler Error CS0236

A field initializer cannot reference the nonstatic field, method, or property '*field*'

Instance fields cannot be used to initialize other instance fields outside of a method.

The following sample generates CS0236:

```
// CS0236.cs
public class MyClass
{
   public int i = 5;
   public int j = i;  // CS0236
   public int k;      // initialize in constructor

   MyClass()
   {
      k = i;
   }

   public static void Main()
   {
   }
}
```

Compiler Error CS0238

'*member*' cannot be sealed because it is not an override

sealed was used on a member that was not also marked with override.

The following sample generates CS0238:

```
// CS0238.cs
abstract class cly
{
   public abstract void f();
}

class clx : cly
{
   public static void Main()
   {
   }

   public sealed void f() // CS0238
   // Try the following definition instead:
   // public override sealed void f()
   {
   }
}
```

Compiler Error CS0239

'*member*' : cannot override inherited member '*inherited member*' because it is sealed

A member cannot override a sealed inherited member.

The following sample generates CS0239:

```
// CS0239.cs
abstract class cly
{
   public abstract void f();
}

class clx : cly
{
   public static void Main()
   {
   }

   public override sealed void f()
   {
   }
}

class clz : clx
{
   public override void f()   // CS0239
   // Try the following definition instead:
   // public new void f()
   {
   }
}
```

Compiler Error CS0241

Default parameter specifiers are not permitted

Method parameters cannot have default values. Use method overloads if you want to achieve the same effect.

The following sample generates CS0241:

```
// CS0241.cs
public class TestClass
{
   public void TestMethod(int i = 9)   // CS0241
   // try the following line instead
   // public void TestMethod(int i)
   {
   }

   public static void Main()
   {
      TestClass x = new TestClass();
      x.TestMethod(9);
   }
}
```

Compiler Error CS0242

The operation in question is undefined on void pointers

Incrementing a void pointer is not allowed.

The following sample generates CS0242:

```
// CS0242.cs
// compile with: /unsafe
class TestClass
{
   public unsafe void Test()
   {
      void * p = null;
      p++;   // CS0242, incrementing a void pointer not allowed
   }

   public static void Main()
   {
   }
}
```

Compiler Error CS0243

Conditional not valid on '*method*' because it is an override method

The Conditional attribute is not allowed on a method that is marked with the override keyword.

The compiler never binds to override methods; it only binds to the base method and the common language runtime calls the override, as appropriate.

The following sample generates CS0243:

```
// CS0243.cs
public class MyClass
{
   public virtual void M()
   {
   }
}

public class MyClass2 : MyClass
{
   [System.Diagnostics.ConditionalAttribute("MySymbol")]   // CS0243
   // remove Conditional attribute or remove override keyword
   public override void M()
   {
   }

   public static void Main()
   {
   }
}
```

Compiler Error CS0244

"is" or "as" are not valid on pointer types

The is and as keywords are not valid for use on pointer types.

The following sample generates CS0244:

```
// CS0244.cs
// compile with: /unsafe
using System;

class UnsafeTest
{
   unsafe static void SquarePtrParam (int* p)
   {
      bool b = p is object;   // CS0244, p is pointer
   }

   unsafe public static void Main()
   {
      int i = 5;
      SquarePtrParam (&i);
   }
}
```

Compiler Error CS0245

Destructors and object.Finalize cannot be called directly. Consider calling IDisposable.Dispose if available.

For more information, see Programming Essentials for Garbage Collection.

The following sample generates CS0245:

```
// CS0245.cs
using System;
using System.Collections;

class C // : IDisposable
{
   /*
   public void Dispose()
   {
      // cleanup code goes here
   }
   */

   void m()
   {
      this.Finalize();   // CS0245
      // this.Dispose();
   }

   public static void Main()
   {
   }
}
```

Compiler Error CS0246

The type or namespace name '*type/namespace*' could not be found (are you missing a using directive or an assembly reference?)

A type was not found. You may have forgotten to reference (/reference) the assembly that contains the type or you may not have correctly qualified its use with the using directive.

There are several reasons for this error:

1. The name of the type or namespace you are trying to use may be misspelled (including the correct case). Without the correct name the compiler is unable to find the definition for the type or namespace you have referred to in your code. This occurs most often because C# is case-sensitive and the correct casing has not been used when referring to the type. For example, `Dataset ds;` will generate CS0246; notice the s in Dataset is not capitalized.
2. If the error is for a namespace name, you may not have referenced (/reference) the assembly containing the namespace. For example, your code might contain `using Accessibility;`. However, if your project doesn't reference the assembly Accessibility.dll then you will get CS0246. See the Add Reference Dialog Box for information on how to reference an assembly in the development environment.

   See Add Reference Dialog Box for information on how to add a reference in the development environment.

3. If the error is for a type name, you may not have the proper using directive, or you have not fully qualified the name of the type. Consider the following line of code: `DataSet ds;`. To be able to use the DataSet type you would need to do two things. First, you need a reference to the assembly that contains the definition for the DataSet type. Second, you need a using directive for the namespace where DataSet is located. For example, because DataSet is located in the **System.Data** namespace, you would need the following statement at the beginning of your code: `using System.Data;`.

   The second step is not required. However, if you omitted this step then it would require that you fully qualify the DataSet type when referring to it. Fully qualifying it means that you use the namespace and type each time you refer to it in your code. So, if you decided to skip the second step you would need to change your declaration code above to: `System.Data.DataSet ds;`.

The following sample generates CS0246:

```
// CS0246.cs
// using System.Diagnostics;

public class MyClass
{
   [Conditional("A")]   // CS0246, uncomment using directive to resolve
   public void Test()
   {
   }

   public static void Main()
   {
   }
}
```

Compiler Error CS0247

Cannot use a negative size with stackalloc

A negative number was passed to a stackalloc statement.

The following sample generates CS0247:

```
// CS0247.cs
// compile with: /unsafe
public class MyClass
{
   unsafe public static void Main()
   {
      int *p = stackalloc int [-30];   // CS0247
   }
}
```

Compiler Error CS0248

Cannot create an array with a negative size

An array size was specified with a negative number.

The following sample generates CS0248:

```
// CS0248.cs
class MyClass
{
   public static void Main()
   {
      int[] myarray = new int[-3] {1,2,3};   // CS0248, pass a nonnegative number
   }
}
```

Compiler Error CS0249

Do not override object.Finalize. Instead, provide a destructor.

Use destructor syntax to specify instructions to execute when your object is destroyed.

See Using C# and Managed Extensions for C++ Destructor Syntax for more information.

The following sample generates CS0249:

```
// CS0249.cs
class C
{
   protected override void Finalize()   // CS0249
   // try the following line instead
   // ~C()
   {
   }

   public static void Main()
   {
   }
}
```

Compiler Error CS0250

Do not directly call your base class Finalize method. It is called automatically from your destructor.

A program cannot attempt to force cleanup of base class resources.

See Finalize Methods and Destructors for more information.

The following sample generates CS0250

```
// CS0250.cs
class B
{
}

class C : B
{
   ~C()
   {
      base.Finalize();   // CS0250
   }

   public static void Main()
   {
   }
}
```

Compiler Warning (level 2) CS0251

Indexing an array with a negative index (array indices always start at zero)

Do not use a negative number to index into an array.

The following sample generates CS0251:

```
// CS0251.cs
// compile with: /W:2
class MyClass
{
   public static void Main()
   {
      int[] myarray = new int[] {1,2,3};
      try
      {
         myarray[-1]++;   // CS0251
         // try the following line instead
         // myarray[1]++;
      }
      catch (System.IndexOutOfRangeException e)
      {
         System.Console.WriteLine("{0}", e);
      }
   }
}
```

Compiler Warning (level 2) CS0252

Possible unintended reference comparison; to get a value comparison, cast the left hand side to type '*type*'

The compiler is doing a reference comparison. If you want to compare the value of strings, cast the left side of the expression to *type*.

The following sample generates CS0252:

```
// CS0252.cs
// compile with: /W:2
using System;

class MyClass
{
   public static void Main()
   {
      string s = "11";
      object o = s + s;

      bool b = o == s;   // CS0252
      // try the following line instead
      // bool b = (string)o == s;
   }
}
```

Compiler Warning (level 2) CS0253

Possible unintended reference comparison; to get a value comparison, cast the right hand side to type '*type*'

The compiler is doing a reference comparison. If you want to compare the value of strings, cast the right side of the expression to *type*.

The following sample generates CS0253:

```
// CS0253.cs
// compile with: /W:2
using System;
class MyClass
{
   public static void Main()
   {
      string s = "11";
      object o = s + s;

      bool c = s == o;   // CS0253
      // try the following line instead
      // bool c = s == (string)o;
   }
}
```

Compiler Error CS0254

The right hand side of a fixed statement assignment may not be a cast expression

The right side of a fixed expression may not use a cast.

The following sample generates CS0254:

```
// CS0254.cs
// compile with: /unsafe
class Point
{
   public int x, y;
}

class FixedTest
{
   unsafe static void SquarePtrParam (int* p)
   {
      *p *= *p;
   }

   unsafe public static void Main()
   {
      Point pt = new Point();
      pt.x = 5;
      pt.y = 6;

      fixed (int* p = (int)&pt.x)   // CS0254
      // try the following line instead
      // fixed (int* p = &pt.x)
      {
         SquarePtrParam (p);
      }
   }
}
```

Compiler Error CS0255

stackalloc may not be used in a catch or finally block

The stackalloc keyword may not be used in a catch or finally block.

The following sample generates CS0255:

```
// CS0255.cs
// compile with: /unsafe
using System;

public class TestTryFinally
{
   public static unsafe void Test()
   {
      int i = 123;
      string s = "Some string";
      object o = s;

      try
      {
         // Conversion is not valid; o contains a string not an int
         i = (int) o;
      }

      finally
      {
         Console.Write("i = {0}", i);
         int* fib = stackalloc int[100];   // CS0255
      }
   }

   public static void Main()
   {
   }
}
```

Compiler Error CS0500

'*class member*' cannot declare a body because it is marked abstract

An abstract method cannot contain its implementation.

The following sample generates CS0500:

```
// CS0500.cs
namespace x
{
   abstract public class clx
   {
      abstract public void f(){}   // CS0500
      // try the following line instead
      // abstract public void f();
   }

   public class cly
   {
      public static int Main()
      {
         return 0;
      }
   }
}
```

Compiler Error CS0501

'*member function*' must declare a body because it is not marked abstract or extern

Nonabstract methods must have implementations. For example, the following sample generates CS0501:

```
// CS0501.cs
namespace x
{
   public class clx
   {
      public void f();   // CS0501, needs implementation
   }
}
```

Compiler Error CS0502

The class '*class*' is abstract and sealed

A class cannot be both abstract and sealed.

The following sample generates CS0502:

```
// CS0502.cs
namespace x
{
   abstract sealed public class clx   // CS0502
   {
      abstract public void f();
   }
}
```

Compiler Error CS0503

The abstract method '*method*' cannot be marked virtual

It is redundant to mark a member method as both abstract and virtual because **abstract** implies **virtual**.

The following sample generates CS0503:

```
// CS0503.cs
namespace x
{
   abstract public class clx
   {
      abstract virtual public void f();   // CS0503
   }
}
```

Compiler Error CS0504

The constant '*variable*' cannot be marked static

If a variable is const, it is also static. If you want a **const** and **static** variable, just declare that variable as **const**; if all you want is a **static** variable, just mark it **static**.

The following sample generates CS0504:

```
// CS0504.cs
namespace x
{
   abstract public class clx
   {
      static const int i = 0;   // CS0504, cannot be both static and const
      abstract public void f();
   }
}
```

Compiler Error CS0505

'*member1*' : cannot override; '*member2*' is not a function

A class declaration attempted to override a non-method in a base class. Overrides must match the member type. If a method with the same name as a method in a base class is desired, use new (and not override) on the method declaration in the base class.

The following sample generates CS0505:

```
// CS0505.cs
namespace x
{
   public class clx
   {
      public int i;

      int z()
      {
         return 0;
      }
   }

   public class cly : clx
   {
      public override int i() // CS0505, clx::i is a variable
      {
         return 0;
      }

      public static void Main()
      {
      }
   }
}
```

Compiler Error CS0506

'*function1*' : cannot override inherited member '*function2*' because it is not marked "virtual", "abstract", or "override"

A method was overridden that was not explicitly marked as **virtual**, **abstract**, or **override**.

The following sample generates CS0506:

```
// CS0506.cs
namespace MyNameSpace
{
   abstract public class ClassX
   {
      public int i = 0;

      public int f()
      {
         return 0;
      }
      // Try the following definition for f() instead:
      // abstract public int f();
   }

   public class ClassY : ClassX
   {
      public override int f()   // CS0506
      {
         return 0;
      }

      public static int Main()
      {
         return 0;
      }
   }
}
```

Compiler Error CS0507

'*function1*' : cannot change access modifiers when overriding '*access*' inherited member '*function2*'

An attempt was made to change the access specification in a method override.

The following sample generates CS0507:

```
// CS0507.cs
abstract public class clx
{
   public int i = 0;

   virtual protected int f()
   {
      return 0;
   }
}

public class cly : clx
{
   public override int f()   // CS0507
   {
      return 0;
   }

   public static int Main()
   {
      return 0;
   }
}
```

Compiler Error CS0508

'*function1*' cannot change return type when overriding inherited member '*function2*'

An attempt was made to change the return type in a method override.

The following sample generates CS0508:

```
// CS0508.cs
namespace x
{
   abstract public class clx
   {
      public int i = 0;
      abstract public int f();   // return type is int
   }

   public class cly : clx
   {
      public override double f()
      {
         return 0.0;             // CS0508
      }

      public static int Main()
      {
         return 0;
      }
   }
}
```

Compiler Error CS0509

'*class1*' : cannot inherit from sealed class '*class2*'

A sealed class cannot act as a base class. Structs are sealed by default.

The following sample generates CS0509:

```
// CS0509.cs
namespace x
{
   sealed public class clx   // class is sealed
   {
      public int i = 0;
   }

   public class cly : clx    // CS0509
   {
      public static int Main()
      {
         return 0;
      }
   }
}
```

Compiler Error CS0513

'*function*' is abstract but it is contained in nonabstract class '*class*'

A method cannot be an abstract member of a nonabstract class.

The following sample generates CS0513:

```
// CS0513.cs
namespace x
{
   public class clx
   {
      abstract public void f();   // CS0513, abstract member of nonabstract class
   }
}
```

Compiler Error CS0514

'*constructor*' : static constructor cannot have an explicit this or base constructor call

There is no object for a static class. Therefore, the base class cannot be called. Objects must be known at compile time.

The following sample generates CS0514:

```
// CS0514.cs
namespace x
{
   public class clx
   {
      public clx(int i)
      {
      }
   }

   public class cly : clx
   {
      static cly() : base(0)  // CS0514, remove base or static
      {
      }
   }
}
```

See Also

base

Compiler Error CS0515

'*function*' : access modifiers are not allowed on static constructors

A constructor cannot have an access modifier.

The following sample generates CS0515:

```
// CS0515.cs
namespace x
{
   public class clx
   {
      public static int Main()
      {
         return 0;
      }
   }

   public class clz
   {
      public static clz()   // CS0515, remove public keyword
      {
      }
   }
}
```

Compiler Error CS0516

Constructor '*constructor*' can not call itself

A program cannot recursively call constructors.

The following sample generates CS0516:

```
// CS0516.cs
namespace x
{
   public class clx
   {
      public clx() : this()   // CS0516, delete "this()"
      {
      }

      public static void Main()
      {
      }
   }
}
```

Compiler Error CS0517

'*class*' has no base class and cannot call a base constructor

CS0517 can occur only when the .NET Framework common language runtime compiles the source code for the object class, which is the only class that has no base class.

Compiler Error CS0518

The predefined type '*type*' is not defined or imported

The main cause for this problem is that the project is not importing mscorlib.dll, which defines the entire System namespace. This can be caused by one of the following:

- The /nostdlib option from the command line compiler has been specified. The /nostdlib option prevents the import of mscorlib.dll. Use this option if you want to define or create a user-specific System namespace.
- An incorrect mscorlib.dll is referenced.
- A corrupt Visual Studio .NET or .NET Framework common language runtime installation exists.
- Residual components from an earlier installation that are incompatible with the latest installation remain.

To resolve this problem, take one of the following actions:

- Do not specify the /nostdlib option from the command line compiler.
- Make sure that the project refers to the correct mscorlib.dll.
- Reinstall the .NET Framework common language runtime (if the previous solutions do not solve the problem).

Compiler Error CS0519

'*name*' conflicts with a predefined namespace

A user-defined identifier is the same as an identifier in a predefined namespace of the common language runtime.

The following sample generates CS0519:

```
// CS0519.cs
class System   // CS0519, rename class
{
   public static int Main()
   {
      return(0);
   }
}
```

Compiler Error CS0520

'*name*' conflicts with a predefined type

One or more files needed by the compiler were not found. Reinstall the .NET Framework common language runtime.

Compiler Error CS0522

'*constructor*' : structs cannot call base class constructors

A struct cannot call a base class constructor; remove the call to the base class constructor.

The following sample generates CS0522:

```
// CS0522.cs
public class clx
{
   public clx(int i)
   {
   }

   public static void Main()
   {
   }
}

public struct cly
{
   public cly(int i):base(0)   // CS0522
   // try the following line instead
   // public cly(int i)
   {
   }
}
```

Compiler Error CS0523

Struct member '*struct2 field*' of type '*struct1*' causes a cycle in the structure layout

The definitions of two structs include recursive references. Change the struct definitions such that each does not define itself on the other.

The following sample generates CS0523:

```
// CS0523.cs
struct RecursiveLayoutStruct1
{
   public RecursiveLayoutStruct2 field;
}

struct RecursiveLayoutStruct2
{
   public RecursiveLayoutStruct1 field;   // CS0523
}

class MyClass
{
   public static void Main()
   {
   }
}
```

Compiler Error CS0524

'*type*' : interfaces cannot declare types

An interface cannot contain a user-defined type.

The following sample generates CS0524:

```
// CS0524.cs
namespace x
{
   public interface clx
   {
      public class cly   // CS0524, delete user-defined type
      {
      }
   }
}
```

Compiler Error CS0525

Interfaces cannot contain fields

An interface can contain methods and properties but not fields.

The following sample generates CS0525:

```
// CS0525.cs
namespace x
{
   public interface clx
   {
      public int i;   // CS0525
   }
}
```

Compiler Error CS0526

Interfaces cannot contain constructors

Constructors cannot be defined for interfaces. A method is considered a constructor if it has the same name as the class and no return type.

The following sample generates CS0526:

```
// CS0526.cs
namespace x
{
   public interface clx
   {
      public clx()   // CS0526
      {
      }
   }

   public class cly
   {
      public static void Main()
      {
      }
   }
}
```

Compiler Error CS0527

'*type*' : type in interface list is not an interface

It is possible for a struct or interface to inherit from another interface but not from any other type.

The following sample generates CS0527:

```
// CS0527.cs
public struct clx : int   // CS0527, int not an interface
{
   public static void Main()
   {
   }
}
```

Compiler Error CS0528

'*interface*' is already listed in interface list

An interface-inheritance list includes a duplicate. An interface can only be specified once in the inheritance list.

The following sample generates CS0528:

```
// CS0528.cs
namespace x
{
   public interface a
   {
   }

   public class b : a, a   // CS0528
   {
      public void Main()
      {
      }
   }
}
```

Compiler Error CS0529

Inherited interface '*interface1*' causes a cycle in the interface hierarchy of '*interface2*'

The inheritance list for an interface includes a direct or indirect reference to itself. An interface cannot inherit from itself.

The following sample generates CS0529:

```
// CS0529.cs
namespace x
{
   public interface a
   {
   }

   public interface b : a, c
   {
   }

   public interface c : b   // CS0529, b inherits from c
   {
   }
}
```

Compiler Error CS0531

'*member*' : interface members cannot have a definition

Methods that are declared in an interface must be implemented in a class that inherits from it and not in the interface itself.

The following sample generates CS0531:

```
// CS0531.cs
namespace x
{
   public interface clx
   {
      int xclx()   // CS0531, cannot define xclx
      // Try the following declaration instead:
      // int xclx();
      {
         return 0;
      }
   }

   public class cly
   {
      public static void Main()
      {
      }
   }
}
```

Compiler Error CS0533

'*derived-class member*' hides inherited abstract member '*base-class member*'

A base class method is hidden. Check the syntax of your declaration to see if it is correct.

For more information, see base.

The following sample generates CS0533:

```
// CS0533.cs
namespace x
{
   abstract public class a
   {
      abstract public void f();
   }

   abstract public class b : a
   {
      new abstract public void f();   // CS0533
      // try the following lines instead
      // override public void f()
      // {
      // }

      public static void Main()
      {
      }
   }
}
```

Compiler Error CS0534

'*function1*' does not implement inherited abstract member '*function2*'

A class is required to implement all the abstract members in the base class, unless the class is also abstract.

The following sample generates CS0534:

```
// CS0534.cs
namespace x
{
   abstract public class clx
   {
      public abstract void f();
   }

   public class cly : clx   // CS0534, no override for clx::f
   {
      // uncomment the following sample override to resolve CS0534
      // override public void f()
      // {
      // }

      public static int Main()
      {
         return 0;
      }
   }
}
```

Compiler Error CS0535

'*class*' does not implement interface member '*member*'

A class derived from an interface, but the class did not implement one or more of the interface's members. A class must implement all members of interfaces from which it derives.

The following sample generates CS0535:

```
// CS0535.cs
namespace x
{
   public interface a
   {
      void f();
   }

   public class b : a   // CS0535, a::f is not implemented
   {
      public static void Main()
      {
      }
   }
}
```

Compiler Error CS0536

'*class*' does not implement interface member '*interface member*'. '*class member*' is static, not public, or has the wrong return type

'*class*' does not implement interface member '*member1*'. '*member2*' is either static, not public, or has the wrong return type.

The compiler did not detect an interface member implementation. A declaration may be present that almost implements the interface member. Check for the following syntax errors in the declaration for the interface member:

- **public** keyword is omitted.
- Return type does not match.
- **static** keyword is present.

The following sample generates CS0536:

```
// CS0536.cs
public interface a
{
   void f();
}

public class b : a
{
   public static int f()   // CS0536
   // try the following line instead
   // public void f()
   {
   }

   public static void Main()
   {
   }
}
```

Compiler Error CS0537

The class System.Object cannot have a base class or implement an interface.

It is recommended not to declare a user-defined class called **Object** in a user-defined namespace called **System**.

The following sample generates CS0537. To resolve this error, remove the base class defined on your declaration of `System.Object` (`MarshalByRefObject`).

```
// CS0537.cs
using System;
namespace System
{
   // To resolve the error, remove the base class MarshalByRefObject.
   public class Object : MarshalByRefObject // CS0537
   {
      static void Main()
      {
      }
   }
}
```

Compiler Error CS0538

'*name*' in explicit interface declaration is not an interface

An attempt was made to explicitly declare an interface, but an interface was not specified.

The following sample generates CS0538:

```
// CS0538.cs
interface MyIFace
{
   void F();
}

public class MyClass
{
   public void G()
   {
   }
}

class C: MyIFace
{
   void MyIFace.F()
   {
   }

   void MyClass.G()   // CS0538, MyClass not an interface
   {
   }
}
```

Compiler Error CS0539

'*member*' in explicit interface declaration is not a member of interface

An attempt was made to explicitly declare an interface member that does not exist. You should either delete the declaration or change it so that it refers to a valid interface member.

The following sample generates CS0539:

```
// CS0539.cs
namespace x
{
   interface I
   {
      void m();
   }

   public class clx : I
   {
      void I.x()   // CS0539
      {
      }

      public static int Main()
      {
         return 0;
      }
   }
}
```

Compiler Error CS0540

'*interface member*' : containing class does not implement interface '*interface*'

You attempted to implement an interface member in a class that does not derive from the interface. You should either delete the implementation of the interface member or add the interface to the base-class list of the class.

The following sample generates CS0540:

```
// CS0540.cs
namespace x
{
   interface I
   {
      void m();
   }

   public class clx
   {
      void I.m()   // CS0540
      {
      }

      public static int Main()
      {
         return 0;
      }
   }
}
```

Compiler Error CS0541

'*declaration*' : explicit interface declaration can only be declared in a class or struct

An explicit interface declaration was found outside a class or struct.

The following sample generates CS0541:

```
// CS0541.cs
namespace x
{
   interface IFace
   {
      void F();
   }

   interface IFace2 : IFace
   {
      void IFace.F();   // CS0541
   }
}
```

Compiler Error CS0542

'*user-defined type*' : member names cannot be the same as their enclosing type

A name was used more than once in the same construct. This error might be caused by inadvertently putting a return type on a constructor.

The following sample generates CS0542:

```
// CS0542.cs
class F
{
   // Remove void from F() to resolve the problem.
   void F()   // CS0542, same name as the class
   {
   }
}

class MyClass
{
   public static void Main()
   {
   }
}
```

Compiler Error CS0543

'*enumeration*' : the enumerator value is too large to fit in its type

A value that was assigned to an element in an enumeration is outside the range of the data type.

The following sample generates CS0543:

```
// CS0543.cs
namespace x
{
   enum I : byte
   {a = 255, b, c}   // CS0543
   public class clx
   {
      public static int Main()
      {
         return 0;
      }
   }
}
```

Compiler Error CS0544

'*property override*' : cannot override; '*non-property*' is not a property

An attempt was made to override a nonproperty data type as a property, which is not allowed.

The following sample generates CS0544:

```
// CS0544.cs
namespace x
{
   public class a
   {
      public int i;
   }

   public class b : a
   {
      public override int i   // CS0544
      // the following line resolves CS0544
      // public new int i
      {
         get
         {
            return 0;
         }
      }

      public static void Main()
      {
      }
   }
}
```

Compiler Error CS0545

'*function*' : cannot override because '*property*' does not have an overridable get accessor

An attempt was made to define an override for a property accessor when the base class has no such definition to override. You can resolve this error by:

- Adding a **set** accessor in the base class.
- Removing the **set** accessor from the derived class.
- Hiding the base class property by adding the new keyword in derived class's property.
- Making the base class property virtual.

The following sample generates CS0545:

```
// CS0545.cs
namespace x
{
   public class a
   {
      public virtual int i
      {
         set
         {
         }
         // uncomment this accessor to resolve CS0545
         // get
         // {
         //   return 0;
         // }
      }
   }

   public class b : a
   {
      public override int i   // CS0545
      {
         get
         {
            return 0;
         }
      }

      public static void Main()
      {
      }
   }
}
```

See Also

Property Declaration | Accessors

Compiler Error CS0546

'*accessor*' : cannot override because '*property*' does not have an overridable set accessor

An attempt to override one of the accessor methods for a property failed because the accessor cannot be overridden. You can resolve this error by:

- Adding a **set** accessor in the base class.
- Removing the **set** accessor from the derived class.
- Hiding the base class property by adding the new keyword in derived class's property.
- Making the base class property virtual.

For more information, see Property Declaration and Accessors.

The following sample generates CS0546:

```
// CS0546.cs
namespace x
{
   public class a
   {
      public virtual int i
      {
         get
         {
            return 0;
         }
         // uncomment the following accessor to resolve this CS0546
         // set
         // {
         // }
      }
   }

   public class b : a
   {
      public override int i
      {
         set   // CS0546, no set
         {
         }
      }

      public static void Main()
      {
      }
   }
}
```

Compiler Error CS0547

'*property*' : property or indexer cannot have void type

void is invalid as a return value for a property.

For more information, see Properties.

The following sample generates CS0547:

```
// CS0547.cs
public class a
{
   public void i   // CS0547
   // Try the following declaration instead:
   // public int i
   {
      get
      {
         return 0;
      }
   }
}

public class b : a
{
   public static void Main()
   {
   }
}
```

Compiler Error CS0548

'*property*' : property or indexer must have at least one accessor

A property must have at least one accessor (get or set) method.

For more information, see Property Declaration and Accessors.

The following sample generates CS0548:

```
// CS0548.cs
public class b
{
   public int MyProp   // CS0548
   {
      // remove comments to resolve CS0548
      /*
      get
      {
         return 0;
      }
      set
      {
         return 0;
      }
      */
   }

   public static void Main()
   {
   }
}
```

Compiler Error CS0549

'*function*' is a new virtual member in sealed class '*class*'

A sealed class cannot be used as a base class, and a virtual method has to be implemented in a derived class, which is a contradiction.

The following sample generates CS0549:

```
// CS0549.cs
sealed public class MyClass
{
   virtual public void TestMethod()   // CS0549
   // to resolve, remove virtual from method or sealed from class
   {
   }

   public static void Main()
   {
   }
}
```

Compiler Error CS0550

'*accessor*' adds an accessor not found in interface member '*property*'

The implementation of a property in a derived class contains an accessor that was not specified in the base interface.

For more information, see Property Declaration and Accessors.

The following sample generates CS0550:

```
// CS0550.cs
namespace x
{
   interface ii
   {
      int i
      {
         get;
         // add the following accessor to resolve this CS0550
         // set;
      }
   }

   public class a : ii
   {
      int ii.i
      {
         get
         {
            return 0;
         }
         set // CS0550, no set in interface
         {
         }
      }

      public static void Main()
      {
      }
   }
}
```

Compiler Error CS0551

Explicit interface implementation '*implementation*' is missing accessor '*accessor*'

A class that explicitly implements an interface's property must implement all the accessors that the interface defines.

For more information, see Property Declaration and Accessors.

The following sample generates CS0551:

```
// CS0551.cs
interface ii
{
   int i
   {
      get;
      set;
   }
}

public class a : ii
{
   int ii.i   // CS0551
   {
      set
      {
      }
      // uncomment the following accessor to resolve
      /*
      get
      {
         return 0;
      }
      */
   }

   public static void Main()
   {
   }
}
```

Compiler Error CS0552

'*conversion routine*' : user defined conversion to/from interface

You cannot create a user-defined conversion to or from an interface. If you need the conversion routine, resolve this error by making the interface a class or derive a class from the interface.

The following sample generates CS0552:

```
// CS0552.cs
public interface ii
{
}

public class a
{
   // delete the routine to resolve CS0552
   public static implicit operator ii(a aa) // CS0552
   {
      return new ii();
   }

   public static void Main()
   {
   }
}
```

Compiler Error CS0553

'*conversion routine*' : user defined conversion to/from base class

User-defined conversions to values of a base class are not allowed; you do not need such an operator.

The following sample generates CS0553:

```
// CS0553.cs
namespace x
{
   public class ii
   {
   }

   public class a : ii
   {
      // delete the conversion routine to resolve CS0553
      public static implicit operator ii(a aa) // CS0553
      {
         return new ii();
      }

      public static void Main()
      {
      }
   }
}
```

Compiler Error CS0554

'*conversion routine*' : user defined conversion to/from derived class

User-defined conversions to values of a derived class are not allowed; you do not need such an operator.

See chapter 6 in the C# language specification for more information on user-defined conversions.

The following sample generates CS0554:

```
// CS0554.cs
namespace x
{
   public class ii
   {
      // delete the conversion routine to resolve CS0554
      public static implicit operator ii(a aa) // CS0554
      {
         return new ii();
      }
   }

   public class a : ii
   {
      public static void Main()
      {
      }
   }
}
```

Compiler Error CS0555

User-defined operator cannot take an object of the enclosing type and convert to an object of the enclosing type

User-defined conversions to values of the enclosing class are not allowed; you do not need such an operator.

The fo3llowing sample generates CS0555:

```
// CS0555.cs
public class MyClass
{
   // delete the following operator to resolve this CS0555
   public static implicit operator MyClass(MyClass aa)   // CS0555
   {
      return new MyClass();
   }

   public static void Main()
   {
   }
}
```

Compiler Error CS0556

User-defined conversion must convert to or from the enclosing type

A user-defined conversion routine must convert to or from the class that contains the routine.

The following sample generates CS0556:

```
// CS0556.cs
namespace x
{
   public class ii
   {
      public class iii
      {
         public static implicit operator int(byte aa)   // CS0556
         // try the following line instead
         // public static implicit operator int(iii aa)
         {
            return 0;
         }
      }

      public static void Main()
      {
      }
   }
}
```

Compiler Error CS0557

Duplicate user-defined conversion in class '*class*'

Duplicate conversion routines are not allowed in a class.

The following sample generates CS0557:

```
// CS0557.cs
namespace x
{
   public class ii
   {
      public class iii
      {
         public static implicit operator int(iii aa)
         {
            return 0;
         }

         public static explicit operator int(iii aa)   // CS0557, delete duplicate
         {
            return 0;
         }
      }

      public static void Main()
      {
      }
   }
}
```

Compiler Error CS0558

User-defined operator '*operator*' must be declared static and public

Both the **static** and **public** access modifiers must be specified on user-defined operators.

The following sample generates CS0558:

```
// CS0558.cs
namespace x
{
   public class ii
   {
      public class iii
      {
         static implicit operator int(iii aa)   // CS0558, add public
         {
            return 0;
         }
      }

      public static void Main()
      {
      }
   }
}
```

Compiler Error CS0559

The parameter and return type for ++ or — operator must be the containing type

The method declaration for an operator overload must follow certain guidelines.

The following sample generates CS0559:

```
// CS0559.cs
public class iii
{
   public static implicit operator int(iii x)
   {
      return 0;
   }

   public static implicit operator iii(int x)
   {
      return null;
   }

   public static int operator ++(int aa)   // CS0559
   // try the following line instead
   // public static iii operator ++(iii aa)
   {
      return (iii)0;
   }

   public static void Main()
   {
   }
}
```

Compiler Error CS0560

Accessor '*accessor1*' : cannot override '*accessor2*' because it is hidden by '*function*'

A property definition is hidden in an indirect base class.

For more information, see Property Declaration and Accessors.

The following sample generates CS0560:

```
// CS0560.cs
namespace NC
{
   class B
   {
      virtual public int Prop
      {
         get
         {
            return 0;
         }
         set
         {
         }
      }
   }

   abstract class D : B
   {
      public new int set_Prop(int i)
      {
         return 0;
      }

      // to resolve this CS0560, delete the set_Prop method and
      // uncomment the definition of Prop
      /*
      public override int Prop
      {
         set
         {
            return;
         }
         get
         {
            return 0;
         }
      }
      */
   }

   class DD : D
   {
      // override blocked by non-accessors
      public override int Prop
      {
         get
         {
            return 0;
         }
         set   // CS0560
         {
            return;
         }
      }

      public static void Main()
```

```
        {
        }
    }
}
```

Compiler Error CS0561

'*function*' : cannot override '*accessor*' because it is a special compiler-generated method

A property declaration created a hidden virtual method (accessor) named **Get*property_name***. That accessor name conflicts with a method name. You can only override this accessor method through a property override not a method override.

The following sample generates CS0561:

```
// CS0561.cs
class B
{
   public virtual int Prop
   {
      get
      {
         return 0;
      }
   }
}

class C : B
{
   public override int get_Prop()   // CS0561
   {
      return 0;
   }

   /* Try the following instead:
   public override int Prop
   {
      get
      {
         return 0;
      }
   }
   */

   public static void Main()
   {
   }
}
```

Compiler Error CS0562

The parameter of a unary operator must be the containing type

The method declaration for an operator overload must follow certain guidelines. For more information, see Overloadable Operators.

The following sample generates CS0562:

```
// CS0562.cs
namespace x
{
   public class iii
   {
      public static implicit operator int(iii x)
      {
         return 0;
      }

      public static implicit operator iii(int x)
      {
         return null;
      }

      public static iii operator +(int aa)   // CS0562
      // try the following line instead
      // public static iii operator +(iii aa)
      {
         return (iii)0;
      }

      public static void Main()
      {
      }
   }
}
```

See Also

Operator Overloading Tutorial

Compiler Error CS0563

One of the parameters of a binary operator must be the containing type

The method declaration for an operator overload must follow certain guidelines. For more information, see Overloadable Operators.

The following sample generates CS0563:

```
// CS0563.cs
public class iii
{
   public static implicit operator int(iii x)
   {
      return 0;
   }
   public static implicit operator iii(int x)
   {
      return null;
   }
   public static int operator +(int aa, int bb)   // CS0563
   // Use the following line instead:
   // public static int operator +(int aa, iii bb)
   {
      return 0;
   }
   public static void Main()
   {
   }
}
```

See Also

Operator Overloading Tutorial

Compiler Error CS0564

Overloaded shift operator must have the type of the first operand be the containing type, and the type of the second operand must be int.

You attempted to overload a shift operator (<< or >>) with incorrectly typed operands. The first operand must be the type and the second operand must be of the type **int**.

The following sample generates CS0564:

```
// CS0564.cs
using System;
class C
{
   public static int operator << (C c1, C c2) // CS0564
// To correct, change second operand to int, like so:
// public static int operator << (C c1, int c2)
   {
      return 0;
   }
   static void Main()
   {
   }
}
```

Compiler Error CS0567

Interfaces cannot contain operators

Operators are not permitted in interface definitions.

The following sample generates CS0567:

```
// CS0567.cs
interface IA
{
   int operator +(int aa, int bb);   // CS0567
}

class Sample
{
   public static void Main()
   {
   }
}
```

Compiler Error CS0568

Structs cannot contain explicit parameterless constructors

Each struct already has a default constructor that initializes the object to zero. Therefore, the constructors that you can create for a struct must take one or more parameters.

The following sample generates CS0568:

```
// CS0568.cs
public struct ClassY
{
   public int field1;
   public ClassY(){}   // CS0568, cannot have no param constructor
   // Try following instead:
   // public ClassY(int i)
   // {
   //    field1 = i;
   // }
}

public class ClassX
{
   public static void Main()
   {
   }
}
```

Compiler Error CS0569

'*method2*' : cannot override '*method1*' because it is not supported by the language

This error occurs when you derive from a base class that was written in another language and when the compiler does not support the method that you are attempting to override.

Compiler Error CS0570

'*class*' references a type not unsupported by the language

This error occurs when using imported metadata that was generated by another compiler. Your code attempted to use a class member that the compiler cannot process.

The following samples generate CS0570:

The following C++ program uses an attribute that is not usable by C#:

```
// CPP0570.cpp
// compile with: /clr /LD
#using <mscorlib.dll>
using namespace System;
using namespace System::Runtime::CompilerServices;

namespace CS0570_Server {
   // This class is marked with RequiredAttributeAttribute.
   // It may not be consumed by other languages.
   [RequiredAttributeAttribute(__typeof(Int32))]
   public __gc class Scenario1 {
   public:
      int intVar;
   };

   public __gc class CS0570Class {
   public:
      Scenario1* sc1_field;
      __property virtual Scenario1* get_sc1_prop() {
         return sc1_field;
      }
      Scenario1* sc1_method() {return 0;}
   };
};
```

Next, compile the following C# program:

```
// CS0570.cs
// compile with: /r:CPP0570.dll
using System;
using CS0570_Server;

public class C {
   public static int Main() {
      CS0570Class r = new CS0570Class();
      r.sc1_field = null;      // CS0570
      object o = r.sc1_prop;   // CS0570
      r.sc1_method();         // CS0570
      int retval = 0x03;
      if (0 == retval)
         Console.WriteLine("PASSED");
      else
         Console.WriteLine("FAILED, retval=="+retval.ToString());
      return retval;
   }
}
```

Compiler Error CS0571

'*function*' : cannot explicitly call operator or accessor

Certain operators have internal names. For example, **op_Increment** is the internal name of the ++ operator. You should not use or explicitly call such method names.

The following sample generates CS0571:

```
// CS0571.cs
public class MyClass
{
   public static MyClass operator ++ (MyClass c)
   {
      return null;
   }

   public static int prop
   {
      get
      {
         return 1;
      }
      set
      {
      }
   }

   public static void Main()
   {
      op_Increment(null);   // CS0571
      // use the increment operator as follows
      // MyClass x = new MyClass();
      // x++;

      set_prop(1);      // CS0571
      // try the following line instead
      // prop = 1;
   }
}
```

Compiler Error CS0572

'*type*' : cannot reference a type through an expression; try '*path_to_type*' instead

An attempt was made to access a member of a class through an identifier, which is not permitted in this situation.

The following sample generates CS0572:

```
// CS0572.cs
using System;
class C
{
   public class Inner
   {
      public static int v = 9;
   }
}

class D : C
{
   public static void Main()
   {
      C cValue = new C();
      Console.WriteLine(cValue.Inner.v);   // CS0572
      // try the following line instead
      // Console.WriteLine(C.Inner.v);
   }
}
```

Compiler Error CS0573

'*field declaration*' : cannot have instance field initializers in structs

You cannot initialize an instance field of a struct. Fields of value types will be initialized to their default values, and reference-type fields will be initialized to **null**. For more information, see Default Values Table.

The following sample generates CS0573:

```
// CS0573.cs
namespace x
{
   public class clx
   {
      public static void Main()
      {
      }
   }

   public struct cly
   {
      clx a = new clx();   // CS0573
      // clx a;            // OK
      int i = 7;           // CS0573
      // int i;            // OK
   }
}
```

See Also

11.3.4 Default values

Compiler Error CS0574

Name of destructor must match name of class

The name of a destructor must be the class name preceded by a tilde (~).

The following sample generates CS0574:

```
// CS0574.cs
namespace x
{
   public class iii
   {
      ~iiii()   // CS0574
      {
      }

      public static void Main()
      {
      }
   }
}
```

Compiler Error CS0575

Only class types can contain destructors

A struct cannot contain a destructor.

The following sample generates CS0575:

```
// CS0575.cs
namespace x
{
   public struct iii
   {
      ~iii()   // CS0575
      {
      }

      public static void Main()
      {
      }
   }
}
```

Compiler Error CS0576

Namespace '' already contains a definition for 'System'

An attempt was made to use the **System** namespace twice.

The following sample generates CS0576:

```
// CS0576.cs
using System.Collections;
using System = System.Collections;   // CS0576, delete this line

public class z
{
   public static void Main()
   {
   }
}
```

Compiler Error CS0577

Conditional not valid on '*function*' because it is a constructor, destructor, operator, or explicit interface implementation

Conditional cannot be applied to the specified methods.

The following sample generates CS0577:

```
// CS0577.cs
interface I
{
   void m();
}

public class MyClass : I
{
   [System.Diagnostics.Conditional("a")]   // CS0577
   // cannot use the above attribute on an explicit interface definition
   void I.m()
   {
   }

   public static void Main()
   {
   }
}
```

Compiler Error CS0578

Conditional not valid on '*function*' because its return type is not void

ConditionalAttribute cannot be applied to a method that has a return type other than **void**. The reason for this is that any other return type for a method may be needed by another part of your program.

The following sample generates CS0578:

```
// CS0578.cs
public class MyClass
{
   [System.Diagnostics.ConditionalAttribute("a")]
   public int TestMethod()   // CS0578
   {
      return 0;
   }

   public static void Main()
   {
   }
}
```

To resolve this error, you must either delete **ConditionalAttribute**, or you must change the return value of the method to **void**.

Compiler Error CS0579

Duplicate '*attribute*' attribute

It is not possible to specify the same attribute more than once unless the attribute specifies **AllowMultiple=true** in its AttributeUsage.

```
// CS0579.cs
using System;
public class MyAttribute : Attribute
{
}

[AttributeUsage(AttributeTargets.All,AllowMultiple=true)]
public class MyAttribute2 : Attribute
{
}

public class z
{
   [MyAttribute, MyAttribute]     // CS0579
   public void zz()
   {
   }

   [MyAttribute2, MyAttribute2]   // OK
   public void zzz()
   {
   }

   public static void Main()
   {
   }
}
```

Compiler Error CS0580

Too many unnamed arguments to attribute '*attribute*'

The number of unnamed arguments is not correct for the attribute. See the definition of the attribute and pass the correct arguments. For more information, see Attributes.

The following sample generates CS0580:

```
// CS0580.cs
using System;
using System.Runtime.CompilerServices;

public class bug
{
   [IndexerName("abc", "def")]   // CS0580
   // name attribute takes one unnamed string parameter
   // try the following line instead
   // [IndexerName("abc")]
   public int this [int index]
   {
      get
      {
         return 0;
      }
   }

   public static void Main()
   {
   }
}
```

Compiler Error CS0581

Named arguments not allowed on attribute '*attribute*'

Some attributes do not take named arguments.

The following sample generates CS0581:

```
// CS0581.cs
using System.Runtime.CompilerServices;

public class MyClass
{
   [IndexerName("xxx", namedarg=3)]   // CS0581
   // try the following line instead
   // [IndexerName("xxx")]
   public int this[int index]
   {
      get
      {
         return 0;
      }
   }

   public static void Main()
   {
   }
}
```

Compiler Error CS0582

Conditional not valid on interface members

ConditionalAttribute is not valid on an interface member.

The following sample generates CS0582:

```
// CS0582.cs
interface MyIFace
{
   [System.Diagnostics.ConditionalAttribute("DEBUG")]   // CS0582, remove conditional attribu
te
   void zz();
}

public class MyClass
{
   public static void Main()
   {
   }
}
```

Compiler Error CS0583

Internal Compiler Error: likely culprit is '*culprit*'.

Please create a bug report file with the '/bugreport' option, and submit the report to your assigned problem reporting contact.

Use the /bugreport compiler option to capture the problem.

Compiler Error CS0584

Internal Compiler Error: stage '*stage*' symbol '*symbol*'

Try to determine if the compiler is failing due to its inability to parse unexpected syntax. Then, contact Microsoft Product Support Services.

Compiler Error CS0585

Internal Compiler Error: stage '*stage*'

Try to determine if the compiler is failing due to its inability to parse unexpected syntax. Then, contact Microsoft Product Support Services.

Compiler Error CS0586

Internal Compiler Error: stage '*stage*'

Try to determine if the compiler is failing due to its inability to parse unexpected syntax. Then, contact Microsoft Product Support Services.

Compiler Error CS0587

Internal Compiler Error: stage '*stage*'

Try to determine if the compiler is failing due to its inability to parse unexpected syntax. Then, contact Microsoft Product Support Services.

Compiler Error CS0588

Internal Compiler Error: stage 'LEX'

Try to determine if the compiler is failing due to its inability to parse unexpected syntax. Then, contact Microsoft Product Support Services.

Compiler Error CS0589

Internal Compiler Error: stage 'PARSE'

Try to determine if the compiler is failing due to its inability to parse unexpected syntax. Then, contact Microsoft Product Support Services.

Compiler Error CS0590

User-defined operators cannot return void

The purpose of a user-defined operator is to return an object.

The following sample generates CS0590:

```
// CS0590.cs
namespace x
{
   public class a
   {
      public static void operator+(a A1, a A2)   // CS0590
      {
      }

      // try the following user-defined operator
      /*
      public static a operator+(a A1, a A2)
      {
         return A2;
      }
      */

      public static int Main()
      {
         return 1;
      }
   }
}
```

Compiler Error CS0591

Invalid value for argument to '*attribute*' attribute

An attribute was passed either an invalid argument or two mutually exclusive arguments.

The following sample generates CS0591:

```
// CS0591.cs
using System;

[AttributeUsage(0)]   // CS0591
class I: Attribute
{
}

public class a
{
   public static void Main()
   {
   }
}
```

Compiler Error CS0592

Attribute '*attribute*' is not valid on this declaration type. It is valid on '*type*' declarations only.

An attribute was applied to a declaration for which it was not intended.

The following sample generates CS0592:

```
// CS0592.cs
using System;

[AttributeUsage(AttributeTargets.Interface)]
public class MyAttribute : Attribute
{
}

[MyAttribute]
public class A  // CS0592, MyAttribute is not valid for a class
{
   public static void Main()
   {
   }
}
```

Compiler Error CS0594

Floating-point constant is outside the range of type '*type*'

A value was assigned to a floating-point variable that is too large for the variables of this data type. See Integral Types Table for information about the range of values allowed in data types.

The following sample generates CS0594:

```
// CS0594.cs
namespace MyNamespace
{
   public class MyClass
   {
      public static void Main()
      {
         float f = 6.77777777777E400;   // CS0594, value too large
      }
   }
}
```

Compiler Error CS0596

The Guid attribute must be specified with the ComImport attribute

The **Guid** attribute must be present when using the **ComImport** attribute.

The following sample generates CS0596:

```
// CS0596.cs
using System.Runtime.InteropServices;

namespace x
{
   [ComImport]   // CS0596
   // try the following line to resolve this CS0596
   // [ComImport, Guid("00000000-0000-0000-0000-000000000001")]
   public class a
   {
   }

   public class b
   {
      public static void Main()
      {
      }
   }
}
```

Compiler Error CS0597

Too few unnamed arguments for attribute '*attribute*'

An attribute did not have the required number of arguments.

Compiler Error CS0599

Invalid value for named attribute argument '*argument*'

An invalid argument was passed to an attribute.

Compiler Error CS0601

The DllImport attribute must be specified on a method marked 'static' and 'extern'.

The **DllImport** attribute was used on a method that did not have the correct access keywords.

The following sample generates CS0601:

```
// CS0601.cs
using System.Runtime.InteropServices;
using System.Text;

public class C
{
   [DllImport("KERNEL32.DLL")]
   extern int GetCurDirectory(int bufSize, StringBuilder buf);   // CS0601
   // try the following line instead
   // static extern int GetCurDirectory(int bufSize, StringBuilder buf);
}

public class MainClass
{
   public static void Main ()
   {
   }
}
```

Compiler Warning (level 1) CS0602

The feature '*old_feature*' is deprecated. Please use '*new_feature*' instead

A language feature used in your code (*old_feature*) is still supported, but that support may be removed in a future release. Instead, you should use the recommended syntax (*new_feature*).

Compiler Error CS0609

Cannot set the name attribute on an indexer marked override

The name attribute (**IndexerNameAttribute**) cannot be applied to an indexed property that is an override. For more information, see Indexers.

The following sample generates CS0609:

```
// CS0609.cs
using System;
using System.Runtime.CompilerServices;

public class idx
{
   public virtual int this[int iPropIndex]
   {
      get
      {
         return 0;
      }
      set
      {
      }
   }
}

public class MonthDays : idx
{
   [IndexerName("MonthInfoIndexer")]   // CS0609, delete to resolve this CS0609
   public override int this[int iPropIndex]
   {
      get
      {
         return 0;
      }
      set
      {
      }
   }
}

public class test
{
   public static void Main( string[] args )
   {
   }
}
```

Compiler Error CS0610

Field or property cannot be of type '*type*'

There are some types that cannot be used as fields or properties. These types include **System.ArgIterator** and **System.TypedReference**.

The following sample generates CS0610 as a result of using **System.TypedReference** as a field:

```
// CS0610.cs
public class MainClass
{
   System.TypedReference i;   // CS0610
   public static void Main ()
   {
   }

   public static void Test(System.TypedReference i)   // OK
   {
   }
}
```

Compiler Error CS0611

Array elements cannot be of type '*type*'

There are some types that cannot be used as the type of an array. These types include **System.TypedReference** and **System.ArgIterator**.

The following sample generates CS0611 as a result of using **System.TypedReference** as an array element:

```
// CS0611.cs
public class a
{
   public static void Main()
   {
      System.TypedReference[] ao = new System.TypedReference [10];   // CS0611
      // try the following line instead
      // int[] ao = new int[10];
   }
}
```

Compiler Warning (level 1) CS0612

'*member*' is obsolete

The class designer marked a member with the Obsolete attribute. This means that the member might not be supported in a future version of the class.

The following sample shows how accessing an obsolete member generates CS0612:

```
// CS0612.cs
// compile with: /W:1
using System;

class MyClass
{
   [Obsolete]
   public static void ObsoleteMethod()
   {
   }

   [Obsolete]
   public static int ObsoleteField;
}

class MainClass
{
   static public void Main()
   {
      MyClass.ObsoleteMethod();    // CS0612 here: method is deprecated
      MyClass.ObsoleteField = 0;   // CS0612 here: field is deprecated
   }
}
```

Compiler Error CS0616

'*class*': is not an attribute class

An attempt was made to use a non-attribute class in an attribute block. All the attribute types need to be inherited from **System.Attribute**.

The following sample generates CS0616:

```
// CS0616.cs
[CMyClass(i=5)]   // CS0616
public class CMyClass
{
   public static void Main()
   {
   }
};
```

The following sample shows how you might define an attribute:

```
// CreateAttrib.cs
using System;

[AttributeUsage(AttributeTargets.Class|AttributeTargets.Interface)]
public class MyAttr : Attribute
{
   public int Name = 0;
   public int Count = 0;

   public MyAttr (int iCount, int sName)
   {
      Count = iCount;
      Name = sName;
   }

   public static void Main()
   {
   }
}

[MyAttr(5, 50)]
class Class1
{
}

[MyAttr(6, 60)]
interface Interface1
{
}
```

Compiler Error CS0617

'*reference*' is not a valid named attribute argument. Named attribute arguments must be fields which are not readonly, static or const, or read-write properties which are not static.

An attempt was made to access a private member of an attribute class.

The following sample generates CS0617:

```
// CS0617.cs
using System;

[AttributeUsage(AttributeTargets.Struct|AttributeTargets.Class|AttributeTargets.Interface)]
public class MyClass : Attribute
{
   public int Name;

   public MyClass (int sName)
   {
      Name = sName;
      Bad = -1;
   }

   public readonly int Bad;
   // try the following line instead
   // public int Bad;
}

[MyClass(5, Bad=0)] class Class1   // CS0617
{
}

public class MyClass2
{
   public static void Main ()
   {
   }
}
```

Compiler Warning (level 2) CS0618

'*member*' is obsolete: '*text*'

A class member was marked with the Obsolete attribute, such that a warning will be issued when the class member is referenced.

The following sample generates CS0618:

```
// CS0618.cs
// compile with: /W:2
using System;

public class C
{
   [Obsolete("Use newMethod instead", false)]   // warn if referenced
   public static void m2()
   {
   }

   public static void newMethod()
   {
   }
}

class MyClass
{
   public static void Main()
   {
      C.m2();  // CS0618
   }
}
```

Compiler Error CS0619

'*member*' is obsolete: '*text*'

A class member was marked with the Obsolete attribute, such that an error will be issued when the class member is referenced.

The following sample generates CS0619:

```
// CS0619.cs
using System;

public class C
{
   [Obsolete("Use newMethod instead", true)]   // generates an error on use
   public static void m()
   {
   }

   // this is the method you should be using
   public static void newMethod()
   {
   }
}

class MyClass
{
   public static void Main()
   {
      C.m();   // CS0619
   }
}
```

Compiler Error CS0620

Indexers can't have void type

The return type of an indexer cannot be **void**. An indexer must return a value.

The following sample generates CS0620:

```
// CS0620.cs
class MyClass
{
   public static void Main()
   {
      MyClass test = new MyClass();
      System.Console.WriteLine(test[2]);
   }

   void this [int intI]   // CS0620, return type cannot be void
   {
      get
      {
         // will need to return some value
      }
   }
}
```

Compiler Error CS0621

'*member*' : virtual or abstract members cannot be private

Private **virtual** or **abstract** members are not allowed.

The following sample generates CS0621:

```
// CS0621.cs
abstract class MyClass
{
   private virtual void DoNothing1()   // CS0621
   {
   }

   private abstract void DoNothing2();   // CS0621

   public static void Main()
   {
   }
}
```

Compiler Error CS0622

Can only use array initializer expressions to assign to array types. Try using a new expression instead.

Syntax that is appropriate to initialize an array was used in the declaration of a non-array.

The following sample generates CS0622:

```
// CS0622.cs
using System;

public class Test
{
   public static void Main ()
   {
      Test t = { new Test() };   // CS0622
      // Try the following instead:
      // Test[] t = { new Test() };
   }
}
```

Compiler Error CS0623

Array initializers can only be used in a variable or field initializer. Try using a new expression instead.

An array was initialized incorrectly.

The following sample generates CS0623:

```
// CS0623.cs
class MyClass
{
   public int[] x = { 2, 3, {4}};   // CS0623

   // The following declarations show how to initialize an array:
   // public int[] x = { 2, 3, 4};   // single-dimensional array
   // int[]x = new int[] {1,2,3};   // dynamically created array
   // int[,]x = { {1,2}, {3,4}, {5,6} };   // multidimensional array
   // object[]z = { null, "test", new int[] {2,3} };   // array of objects
   // int[][]x = { new int[]{2,3}, new int[]{4,5} };   // array of ints
   // private static readonly char[] array = { ',', ' ' };

   public static void Main()
   {
   }
}
```

Compiler Error CS0625

'*field*' : Instance field types marked with StructLayout(LayoutKind.Explicit) must have a FieldOffset attribute.

When a struct is marked with an explicit **StructLayout** attribute, all fields in the struct must have the FieldOffset attribute. For more information, see StructLayoutAttribute Class.

The following sample generates CS0625:

```
// CS0625.cs
using System;
using System.Runtime.InteropServices;

[StructLayout(LayoutKind.Explicit)]
struct Worksheet
{
   // uncomment the next line to resolve
   // [FieldOffset(5)]
   public int i;   // CS0625, not static; an instance field
}

public class MainClass
{
   public static void Main ()
   {
   }
}
```

Compiler Warning (level 1) CS0626

Method, operator, or accessor '*method*' is marked external and has no attributes on it. Consider adding a DllImport attribute to specify the external implementation

A method marked **extern** should also be marked with an attribute, for example, the DllImport attribute.

The attribute specifies where the method is implemented. At run time, the program will need this information.

The following sample generates CS0626:

```
// CS0626.cs
// compile with: /warnaserror
using System.Runtime.InteropServices;

public class MyClass
{
   static extern public void mf(); // CS0626
   // try the following line
   // [DllImport("mydll.dll")] static extern public void mf();

   public static void Main()
   {
   }
}
```

Compiler Warning (level 4) CS0628

'*member*' : new protected member declared in sealed class

A sealed class cannot introduce a protected member because no other class will be able to inherit from the **sealed** class and use the **protected** member.

The following sample generates CS0628:

```
// CS0628.cs
// compile with: /W:4
sealed class C
{
   protected int i;   // CS0628
}

class MyClass
{
   public static void Main()
   {
   }
}
```

Compiler Error CS0629

Conditional member '*member*' cannot implement interface member '*base class member*'

The Conditional attribute cannot be used on the implementation of an interface.

The following sample generates CS0629:

```
// CS0629.cs
interface MyInterface
{
   void MyMethod();
}

public class MyClass : MyInterface
{
   [System.Diagnostics.Conditional("debug")]
   public void MyMethod()    // CS0629, remove the Conditional attribute
   {
   }

   public static void Main()
   {
   }
}
```

Compiler Error CS0631

Indexers can't have ref or out parameters

The declaration for an indexer property cannot include the use of ref or out parameters.

The following sample generates CS0631:

```
// CS0631.cs
public class MyClass
{
   public int this[ref int i]   // CS0631
   // try the following line instead
   // public int this[int i]
   {
      get
      {
         return 0;
      }
   }
}

public class MyClass2
{
   public static void Main()
   {
   }
}
```

Compiler Error CS0633

The argument to the IndexerName attribute must be a valid identifier

Any argument that you pass to the **IndexerName** attribute must be a valid identifier.

The following sample generates CS0633:

```
// CS0633.cs
using System;
using System.Runtime.CompilerServices;

public class MonthDays
{
   [IndexerName("bad indexer name")]   // CS0633
   // try the following line instead
   // [IndexerName("badindexername")]
   public int this[int iPropIndex]
   {
      get
      {
         return 0;
      }
      set
      {
      }
   }
}

public class test
{
   public static void Main( string[] args )
   {
   }
}
```

Compiler Error CS0634

'*arg*' : argument only valid for marshal of type System.Interop.UnmanagedType.CustomMarshaller

One of the arguments passed to the **Marshal** attribute can only be used when the marshal format is **System.Interop.UnmanagedType.CustomMarshaller**.

Compiler Error CS0635

'*attribute*' : System.Interop.UnmanagedType.CustomMarshaller requires named arguments ComType and Marshal

The **ComType** and **Marshal** arguments must be specified when the marshal format is **System.Interop.UnmanagedType.CustomMarshaller**.

Compiler Error CS0636

The FieldOffset attribute can only be placed on members of types marked with the StructLayout(LayoutKind.Explicit)

You must use the **StructLayout(LayoutKind.Explicit)** attribute on the struct declaration, if it contains any members marked with the **FieldOffset** attribute. For more information, see FieldOffset.

The following sample generates CS0636:

```
// CS0636.cs
using System;
using System.Runtime.InteropServices;

// To resolve the error, uncomment the following line:
// [StructLayout(LayoutKind.Explicit)]
struct Worksheet
{
   [FieldOffset(4)]public int i;   // CS0636
}

public class MainClass
{
   public static void Main ()
   {
   }
}
```

Compiler Error CS0637

The FieldOffset attribute not allowed on static or const fields

The FieldOffset attribute cannot be used on fields marked with static or const.

The following sample generates CS0637:

```
// CS0637.cs
using System;
using System.Runtime.InteropServices;

[StructLayout(LayoutKind.Explicit)]
public class MainClass
{
   [FieldOffset(3)]   // CS0637
   public static int i;
   public static void Main ()
   {
   }
}
```

Compiler Error CS0641

'*attribute*' : attribute is only valid on classes derived from System.Attribute

An attribute was used that can only be used on a class that derives from **System.Attribute**.

The following sample generates CS0641:

```
// CS0641.cs
using System;

[AttributeUsage(AttributeTargets.All)]
public class NonAttrClass   // CS0641
// try the following line instead
// public class NonAttrClass : Attribute
{
}

class MyClass
{
   public static void Main()
   {
   }
}
```

Compiler Warning (level 3) CS0642

Possible mistaken null statement

A semicolon after a conditional statement may cause your code to execute differently than intended.

The following sample generates CS0642:

```
// CS0642.cs
// compile with: /W:3
using System;

class MyClass
{
   public static void Main()
   {
      int i;

      for (i = 0; i < 10; i += 1);   // CS0642, semicolon intentional?
      {
         Console.WriteLine (i);
      }
   }
}
```

Compiler Error CS0643

'*arg*' duplicate named attribute argument

A parameter, ***arg***, on a user-defined attribute was specified twice. For more information, see Introduction to Attributes.

The following sample generates CS0643:

```
// CS0643.cs
using System;
using System.Runtime.InteropServices;

[AttributeUsage(AttributeTargets.Class)]
public class MyAttribute : Attribute
{
   public MyAttribute()
   {
   }

   public int x;
}

[MyAttribute(x = 5, x = 6)]   // CS0643, error setting x twice
// try the following line instead
// [MyAttribute(x = 5)]
class MyClass
{
}

public class MainClass
{
   public static void Main ()
   {
   }
}
```

Compiler Error CS0644

'*class1*' cannot inherit from special class '*class2*'

Classes cannot explicitly inherit from any of the following base classes:

- **System.Enum**
- **System.ValueType**
- **System.Delegate**
- **System.Array**

These are used as implicit base classes by the compiler. For example, **System.ValueType** is the implicit base class of structs.

The following sample generates CS0644:

```
// CS0644.cs
class MyClass : System.ValueType   // CS0644
{
}
```

Compiler Error CS0645

Identifier too long

A class name or other identifier can be no longer than 512 characters.

Compiler Error CS0646

Cannot specify the DefaultMember attribute on type containing an indexer

If a class or other type specifies **System.Reflection.DefaultMemberAttribute**, it cannot contain an indexer. For more information, see Properties.

The following sample generates CS0646:

```
// CS0646.cs
// to resolve, delete System.Reflection.DefaultMemberAttribute or
// remove the indexer
[System.Reflection.DefaultMemberAttribute("x")]   // CS0646
class MyClass
{
   public int this[int index]                     // an indexer
   // try the following line instead
   // public int prop
   {
      get
      {
         return 0;
      }
   }

   public int x;
   public static void Main()
   {
   }
}
```

Compiler Error CS0647

Error emitting '*attribute*' attribute -- '*reason*'

The following sample generates CS0647:

```
// CS0647.cs
using System.Runtime.InteropServices;

[Guid("z")]   // CS0647, incorrect uuid format.
// try the following line instead
// [Guid("00000000-0000-0000-0000-000000000001")]
public class MyClass
{
   public static void Main()
   {
   }
}
```

Compiler Error CS0648

'*type*' is a type not supported by the language

Metadata that was generated from another language, possibly using the Managed Extensions for C++, contained a type that was not marked as managed. Information about this type is included in the metadata, but the type is not available to programs written in C#.

Compiler Warning (level 4) CS0649

Field '*field*' is never assigned to, and will always have its default value '*value*'

The compiler detected an uninitialized private or internal field declaration that is never assigned a value.

The following sample generates CS0649:

```
// CS0649.cs
// compile with: /W:4
using System.Collections;

class MyClass
{
   Hashtable table;  // CS0649
   // You may have intended to initialize the variable to null
   // Hashtable table = null;

   // Or you may have meant to create an object here
   // Hashtable table = new Hashtable();

   public void Func(object o, string p)
   {
      // Or here
      // table = new Hashtable();
      table[p] = o;
   }

   public static void Main()
   {
   }
}
```

Compiler Error CS0650

Syntax error, bad array declarator. To declare a managed array the rank specifier precedes the variable's identifier

An array was declared incorrectly.

The following sample generates CS0650:

```
// CS0650.cs
public class MyClass
{
   public static void Main()
   {
      int myarray[2];   // CS0650
      // try the following lines instead
      // int[] myarray = new int[2] {1,2};
      // myarray[0] = 0;
   }
}
```

Compiler Warning (level 2) CS0652

Comparison to integral constant is useless; the constant is outside the range of type '*type*'

The compiler detected a comparison between a constant and a variable where the constant is out of the range of the variable.

The following sample generates CS0652:

```
// CS0652.cs
// compile with: /W:2
public class Class1
{
   private static byte i = 0;
   public static void Main()
   {
      short j = 256;
      if (i == 256)   // CS0652, 256 is out of range for byte
         i = 0;
   }
}
```

Compiler Error CS0653

Cannot apply attribute class '*class*' because it is abstract

An abstract custom attribute class cannot be used as an attribute.

The following sample generates CS0653:

```
// CS0653.cs
using System;

public abstract class MyAttribute : Attribute
{
}

public class My2Attribute : MyAttribute
{
}

[My]   // CS0653
// try the following line instead
// [My2]
class MyClass
{
   public static void Main()
   {
   }
}
```

Compiler Error CS0654

Method '*method*' referenced without parentheses

A method was called incorrectly.

The following sample generates CS0654:

```
// CS0654.cs
using System;

class MyClass
{
   public static int ReturnZero(int i)
   {
      return 0;
   }

   public static void Main()
   {
      int i;
      i = ReturnZero;   // CS0654
      // try the following line instead
      i = ReturnZero(4);
   }
}
```

Compiler Error CS0655

'*parameter*' is not a valid named attribute argument because it is not a valid attribute parameter type.

See the Attributes Tutorial for a discussion of valid parameter types for an attribute.

The following sample generates CS0655:

```
// CS0655.cs
using System;

class MyAttribute : Attribute
{
   // decimal is not valid attribute parameter type
   public decimal d = 0;
   public int e = 0;
}

[My(d = 0)]   // CS0655
// try the following line instead
// [My(e = 0)]
class C
{
   public static void Main()
   {
   }
}
```

Compiler Error CS0656

Missing compiler required member '*object.member*'

One of the following problems exists:

- Your installation of the common language runtime is corrupt.
- You have a reference to an assembly that defines a type that is also found in the common language runtime. However, your assembly's type is not defined the way the C# compiler expects.

Check your references to ensure that you are using the correct version of the common language runtime.

Compiler Error CS0657

'*attribute modifier*' is not a valid attribute location for this declaration. Valid attribute locations for this declaration are '*location*'

The compiler found an attribute modifier in an invalid location. See Attribute Targets for more information.

The following sample generates CS0657:

```
// CS0657.cs
using System;
public class TestAttribute : Attribute{}
[return: Test]   // CS0657, return not valid on a class
class Class1
{
   public static void Main()
   {
   }
}
```

Compiler Error CS0658

'*attribute modifier*' is not a recognized attribute location. All attributes in this block will be ignored.

An invalid attribute modifier was specified. See Attribute Targets for more information.

The following sample generates CS0658:

```
// CS0658.cs
using System;
public class TestAttribute : Attribute{}
[badAttributeLocation: Test]   // CS0658, badAttributeLocation is invalid
class ClassTest
{
   public static void Main()
   {
   }
}
```

Compiler Warning (level 3) CS0659

'*class*' overrides Object.Equals(object o) but does not override Object.GetHashCode()

The compiler detected an override of the **Equals** function but no override for **GetHashCode**. An override of **Equals** implies that you also want to override **GetHashCode**.

The following sample generates CS0659:

```
// CS0659.cs
// compile with: /W:3
class Test   // CS0659
{
   public override bool Equals(object o)
   {
      return true;
   }

   public static void Main()
   {
   }
}
```

Compiler Warning (level 3) CS0660

'*class*' defines operator == or operator != but does not override Object.Equals(object o)

The compiler detected the user-defined equality or inequality operator, but no override for the **Equals** function. A user-defined equality or inequality operator implies that you also want to override the **Equals** function.

The following sample generates CS0660:

```
// CS0660.cs
// compile with: /W:3 /warnaserror
class Test   // CS0660
{
   public static bool operator == (object o, Test t)
   {
      return true;
   }

   // uncomment the Equals function to resolve
   // public override bool Equals(object o)
   // {
   //    return true;
   // }

   public override int GetHashCode()
   {
      return 0;
   }

   public static void Main()
   {
   }
}
```

Compiler Warning (level 3) CS0661

'class' defines operator == or operator != but does not override Object.GetHashCode()

The compiler detected the user-defined equality or inequality operator, but no override for the **GetHashCode** function. A user-defined equality or inequality operator implies that you also want to override the **GetHashCode** function.

The following sample generates CS0661:

```
// CS0661.cs
// compile with: /W:3
class Test   // CS0661
{
   public static bool operator == (object o, Test t)
   {
      return true;
   }

   public static bool operator != (object o, Test t)
   {
      return true;
   }

   public override bool Equals(object o)
   {
      return true;
   }

   // uncomment the GetHashCode function to resolve
   // public override int GetHashCode()
   // {
   //    return 0;
   // }

   public static void Main()
   {
   }
}
```

Compiler Error CS0662

'*method*' cannot specify only Out attribute on a ref parameter. Use both In and Out attributes, or neither.

An interface method has a parameter that uses ref with just the Out attribute. A **ref** parameter that uses the **Out** attribute must also use the In attribute.

The following sample generates CS0662:

```
// CS0662.cs
using System.Runtime.InteropServices;

interface I
{
   void method([Out] ref int i);   // CS0662
   // try one of the following lines instead
   // void method(ref int i);
   // void method([Out, In]ref int i);
}

class test
{
   public static void Main()
   {
   }
}
```

Compiler Error CS0663

'*method*' cannot define overloaded methods which differ only on ref and out

Methods that differ only on their use of ref and out on a parameter are not allowed.

The following sample generates CS0663:

```
// CS0663.cs
class TestClass
{
   public static void Main()
   {
   }

   public void Test(ref int i)
   {
   }

   public void Test(out int i)   // CS0663
   {
   }
}
```

Compiler Error CS0664

Literal of type double cannot be implicitly converted to type '*type*'; use an '*suffix*' suffix to create a literal of this type

An assignment could not be completed; use a suffix to correct the instruction. The documentation for each type identifies the corresponding suffix for the type.

The following sample generates CS0664:

```
// CS0664.cs
class M
{
   static void Main()
   {
      decimal m = 1.0;   // CS0664
      // try the following line instead
      // decimal m = 1.0M;
      System.Console.WriteLine(m);
   }
}
```

Compiler Warning (level 3) CS0665

Assignment in conditional expression is always constant; did you mean to use == instead of = ?

A conditional expression used the = operator and not the == operator.

The following sample generates CS0665:

```
// CS0665.cs
// compile with: /W:3
class Test
{
   public static void Main()
   {
      bool i = false;

      if (i = true)   // CS0665
      // try the following line instead
      // if (i == true)
      {
      }

      System.Console.WriteLine(i);
   }
}
```

Compiler Error CS0666

'*member*' : new protected member declared in struct

A struct cannot be **abstract** and is always implicitly **sealed**. For more information, see 11.3.2 Inheritance.

The following sample generates CS0666:

```
// CS0666.cs
class M
{
   static void Main()
   {
   }
}

struct S
{
   protected int x;   // CS0666
}
```

Compiler Error CS0667

The feature 'MarshalAs attribute on method' is deprecated. Please use 'return : location override with MarshalAs attribute' instead'

The following sample generates CS0667:

```
// CS0667.cs
using System.Runtime.InteropServices;

public interface MyInterface
{
   [MarshalAs(UnmanagedType.LPWStr )] string Test(); // CS0667
   // try the following line instead
   // [return: MarshalAs(UnmanagedType.LPWStr)] string Test();

}

class MyTestClass
{
   static void Main()
   {
   }
}
```

See Also

COM Interop Tutorials | Platform Invoke Tutorial

Compiler Error CS0669

A class with the ComImport attribute cannot have a user-defined constructor

The COM interop layer in the common language runtime supplies the constructor for ComImport classes. Consequently, a COM object can be used as a managed object in the runtime.

The following sample generates CS0669:

```
// CS0669.cs
using System.Runtime.InteropServices;
[ComImport, Guid("00000000-0000-0000-0000-000000000001")]
class TestClass
{
   TestClass()   // CS0669, delete constructor to resolve
   {
   }

   public static void Main()
   {
   }
}
```

Compiler Error CS0668

Two indexers have different names; the IndexerName attribute must be used with the same name on every indexer within a type

The values passed to the **IndexerName** attribute must be the same for all indexers in a type. For more information on the **IndexerName** attribute, see IndexerNameAttribute Class.

The following sample generates CS0668:

```
// CS0668.cs
using System;
using System.Runtime.CompilerServices;

class IndexerClass
{
   [IndexerName("IName1")]
   public int this [int index]   // indexer declaration
   {
      get
      {
         return index;
      }
      set
      {
      }
   }

   [IndexerName("IName2")]
   public int this [string s]    // CS0668, change IName2 to IName1
   {
      get
      {
         return int.Parse(s);
      }
      set
      {
      }
   }

   void Main()
   {
   }
}
```

Compiler Error CS0670

Field cannot have void type

A field was declared to be of type void.

The following sample generates CS0670:

```
// CS0670.cs
class C
{
   void f;   // CS0670
   // try the following line instead
   // public int f;

   public static void Main()
   {
      C myc = new C();
      myc.f = 0;
   }
}
```

Compiler Warning (level 1) CS0672

Member '*member1*' overrides obsolete member '*member2*. Add the Obsolete attribute to '*member1*'

The compiler found an **override** to a method marked as **obsolete**. However, the overriding method was not itself marked as obsolete. The overriding method will still generate CS0612, if called.

Review your method declarations and explicitly indicate whether a method (and all of its overrides) should be marked **obsolete**.

The following sample generates CS0672:

```
// CS0672.cs
// compile with: /W:1
class MyClass
{
   [System.Obsolete]
   public virtual void ObsoleteMethod()
   {
   }
}

class MyClass2 : MyClass
{
   public override void ObsoleteMethod()   // CS0672
   {
   }
}

class MainClass
{
   static public void Main()
   {
   }
}
```

Compiler Error CS0673

System.Void cannot be used from C# -- use typeof(void) to get the void type object.

System.Void cannot be used in C#.

The following sample generates CS0673:

```
// CS0673.cs
class MyClass
{
   public static void Main()
   {
      System.Type t = typeof(System.Void);   // CS0673
      // try the following line instead
      // System.Type t = typeof(void);
   }
}
```

Compiler Error CS0674

Do not use 'System.ParamArrayAttribute'. Use the 'params' keyword instead.

The C# compiler does not allow the use of System.ParamArrayAttribute; use params instead.

The following sample generates CS0674:

```
// CS0674.cs
using System;
public class MyClass
{

   public static void UseParams([ParamArray] int[] list)   // CS0674
   // try the following line instead
   // public static void UseParams(params int[] list)
   {
      for ( int i = 0 ; i < list.Length ; i++ )
         Console.WriteLine(list[i]);
      Console.WriteLine();
   }

   public static void Main()
   {
      UseParams(1, 2, 3);
   }
}
```

Compiler Warning (level 3) CS0675

Bitwise-or operator used on a sign-extended operand; consider casting to a smaller unsigned type first

The compiler implicitly widened and sign-extended a variable, and then used the resulting value in a bitwise OR operation. This can result in unexpected behavior.

The following sample generates CS0675:

```
// CS0675.cs
// compile with: /W:3
using System;

public class sign
{
   public static void Main()
   {
      int hi = 1;
      int lo = 1;
      long value = (((long)hi) << 32) | lo;              // CS0675
      // try the following line instead
      // long value = (((long)hi) << 32) | ((uint)lo);   // correct
   }
}
```

Compiler Error CS0676

Cannot pass volatile field '*variable*' as ref or out, or take its address

A variable that is marked with the **volatile** keyword cannot be passed to a ref or out parameter.

The following sample generates CS0676:

```
// CS0676.cs
using System;

class TestClass
{
   private volatile int i;

   public void TestVolatile(ref int ii)
   {
   }

   public static void Main()
   {
      TestClass x = new TestClass();
      x.TestVolatile(ref x.i);   // CS0676 don't pass volatile value as ref
   }
}
```

Compiler Error CS0677

'*variable*': a volatile field can not be of the type '*type*'

Fields declared with the **volatile** keyword must be one of the following types:

- Any reference type
- Any pointer type (in an **unsafe** context)
- The types **sbyte**, **byte**, **short**, **ushort**, **int**, **uint**, **char**, **float**, **bool**
- Enum types based on any of the above types

The following sample generates CS0677:

```
// CS0677.cs
class TestClass
{
   private volatile long i;   // CS0677

   public static void Main()
   {
   }
}
```

Compiler Error CS0678

'*variable*': a field can not be both volatile and readonly

Use of the volatile and readonly keywords is mutually exclusive.

The following sample generates CS0678:

```
// CS0678.cs
using System;

class TestClass
{
   private readonly volatile int i;   // CS0678
   // try the following line instead
   // private volatile int i;

   public static void Main()
   {
   }
}
```

Compiler Error CS0682

'type1' cannot implement 'type2' because it is not supported by the language

This error occurs when you try to implement an interface written in another language and the compiler does not support said interface.

Compiler Error CS0683

"'*explicitmethod*' explicit method implementation cannot implement '*method*' because it is an accessor

The following sample generates CS0683:

```
// CS0683.cs
interface IExample
{
   int Test { get; }
}

class CExample : IExample
{
   int IExample.get_Test() { return 0; } // CS0683
   int IExample.Test { get{ return 0; } } // correct
}
```

Compiler Warning (level 1) CS0684

'*interface*' interface marked with 'CoClassAttribute' not marked with 'ComImportAttribute'

If you specify **CoClassAttribute** on an interface, you must also specify **ComImportAttribute**.

The following sample generates CS0684:

```
// CS0684.cs
// compile with: /W:1
using System;
using System.Runtime.InteropServices;

[CoClass(typeof(C))] // CS0684
// try the following line instead
// [CoClass(typeof(C)), ComImport]
interface I
{
}

class C
{
   static void Main() {}
}
```

Compiler Warning (level 1) CS0688

'*member1*' (a method, property accessor, or event accessor) has a link demand, but overrides or implements '*member2*' (a method, property accessor, or event accessor) which does not have a link demand. A security hole may exist.

The following sample generates CS0688. To resolve the warning, remove the security attribute from the overriding method.

```
// CS0688.cs
// compile with: /W:1
using System;
using System.Security.Permissions;

class Base
{
   public virtual void DoScaryFileStuff()
   {
   }
}

class Derived: Base
{
   // To resolve the error, remove the following security attribute:
   [FileIOPermission(SecurityAction.LinkDemand)] // CS0688
   public override void DoScaryFileStuff()
   {
   }
   static void Main()
   {
   }
}
```

Alternatively, you could add the security attribute to the base overridden method as follows:

```
// CS0688b.cs
using System;
using System.Security.Permissions;

class Base
{
   [FileIOPermission(SecurityAction.LinkDemand, All=@"C:\\")]
   public virtual void DoScaryFileStuff()
   {
   }
}

class Derived: Base
{
   [FileIOPermission(SecurityAction.LinkDemand, All=@"C:\\")]

   public override void DoScaryFileStuff()
   {
   }
   static void Main()
   {
   }
}
```

Compiler Error CS0690

Input file '*file*' is not a valid PE because it contains invalid metadata

You are able to open the metadata file, but due to some localization problem, the metadata was corrupted. This is similar to error CS0009, except that you are able to open the metadata file.

Compiler Error CS1001

Identifier expected

You did not supply an identifier. For example, when declaring an enum, you must specify members.

The following sample generates CS1001:

```
// CS1001.cs
public class clx
{
   enum splitch : int
   {
      'a', 'b' // CS1001, 'a' is not a valid int identifier
   };

   public static void Main()
   {
   }
}
```

Compiler Error CS1002

; expected

The compiler detected a missing semicolon.

The following sample generates CS1002:

```
// CS1002.cs
namespace x
{
   abstract public class clx
   {
      int i   // CS1002, missing semicolon

      public static int Main()
      {
         return 0;
      }
   }
}
```

Compiler Error CS1003

Syntax error, '*char*' expected

The compiler will generate this error for any one of several error conditions. Review your code to find the syntax error.

The following sample generates CS1003:

```
// CS1003.cs
public class b
{
   public static void Main()
   {
      int[] a;
      a[];   // CS1003
   }
}
```

Compiler Error CS1004

Duplicate '*modifier*' modifier

A duplicate modifier, such as an **access** modifier, was detected.

The following sample generates CS1004:

```
// CS1004.cs
public class clx
{
   public public static void Main()   // CS1004, two public keywords
   {
   }
}
```

Compiler Error CS1005

Indirection to managed type is not valid

It is not possible to use unsafe references (pointer syntax) on an object or string.

The following sample generates CS1005:

```
// CS1005.cs
namespace x
{
   public class C
   {
      public object* po;   // CS1005
      // try the following line instead
      // public object po;
   }

   public class a
   {
      public static int Main()
      {
         return 1;
      }
   }
}
```

Compiler Error CS1007

Property accessor already defined

When you declare a property, you must also declare its accessor methods. However, a property cannot have more than one **get** accessor method or more than one **set** accessor method.

The following sample generates CS1007:

```
// CS1007.cs
public class clx
{
   public int MyProperty
   {
      get
      {
         return 0;
      }
      get   // CS1007, this is the second get method
      {
         return 0;
      }
   }

   public static void Main() {}
}
```

Compiler Error CS1008

Type byte, sbyte, short, ushort, int, uint, long, or ulong expected

Certain data types, such as enums, can only be declared to hold data of specified types.

The following sample generates CS1008:

```
// CS1008.cs
abstract public class clx
{
   enum splitch : char   // CS1008, char not valid type for enums
   {
      x, y, z
   }

   public static void Main()
   {
   }
}
```

Compiler Error CS1009

Unrecognized escape sequence

An unexpected character followed a backslash (\) in a string. The compiler expects one of the valid escape characters; see Character Escapes for more information.

The following sample generates CS1009:

```
// CS1009-a.cs
class MyClass
{
   static void Main()
   {
      string a = "\m";   // CS1009
      // try the following line instead
      // string a = "\t";
   }
}
```

A common case for this error is using the backslash character in a file name, for example:

```
string filename = "c:\myFolder\myFile.txt";
```

To resolve this error, use "\\" or the @-quoted string literal, as shown in the following example:

```
// CS1009-b.cs
class MyClass
{
   static void Main()
   {
      string filename = "c:\myFolder\myFile.txt";   // CS1009
      // try the one of the following lines instead
      // string filename = "c:\\myFolder\\myFile.txt";
      // string filename = @"c:\myFolder\myFile.txt";
   }
}
```

Compiler Error CS1010

Newline in constant

A string was not properly delimited.

The following sample generates CS1010:

```
// CS1010.cs
class Sample
{
   static void Main()
   {
      string a = "Hello World;   // CS1010, add end quote
   }
}
```

Compiler Error CS1011

Empty character literal

A char was declared and initialized to a null. The initialization of a **char** must specify a character.

The following sample generates CS1011:

```
// CS1011.cs
class Sample
{
   public char CharField = '';   // CS1011
}
```

Compiler Error CS1012

Too many characters in character literal

An attempt was made to initialize a char constant with more than one character.

The following sample generates CS1012:

```
// CS1012.cs
class Sample
{
   static void Main()
   {
      char a = 'xx';   // CS1012
   }
}
```

Compiler Error CS1013

Invalid number

The compiler detected a malformed number.

The following sample generates CS1013:

```
// CS1013.cs
class Sample
{
   static void Main()
   {
       int i = 0x;   // CS1013
   }
}
```

See Also

Integral Types Table

Compiler Error CS1014

A get or set accessor expected

A method declaration was found in a property declaration. Only **get** and **set** methods are allowed to be declared in a property.

The following sample generates CS1014:

```
// CS1014.cs
class Sample
{
   public int TestProperty
   {
      get
      {
         return 0;
      }
      int z;   // CS1014, not get or set
   }

   static void Main() {}
}
```

See Also

Property Declaration | Accessors

Compiler Error CS1015

An object, string, or class type expected

An attempt was made to pass a predefined data type into a catch block. Only data types that derive from **System.Exception** can be passed into a **catch** block.

The following sample generates CS1015:

```
// CS1015.cs
class Sample
{
   static void Main()
   {
      try
      {
      }
      catch(int)   // CS1015, int is not derived from System.Exception
      {
      }
   }
}
```

See Also

Exception Handling Statements

Compiler Error CS1016

Named attribute argument expected

Unnamed attribute arguments must appear before the named arguments.

The following sample generates CS1016:

```
// CS1016.cs
using System;

[AttributeUsage(AttributeTargets.Class)]
public class HelpAttribute : Attribute
{
   public HelpAttribute(string url)   // url is a positional parameter
   {
      m_url = url;
   }

   public string Topic = null;      // Topic is a named parameter
   private string m_url = null;
}

[HelpAttribute(Topic="Samples", "http://intranet/inhouse")]   // CS1016
// try the following line instead
//[HelpAttribute("http://intranet/inhouse", Topic="Samples")]
public class MainClass
{
   public static void Main ()
   {
   }
}
```

Compiler Error CS1017

Try statement already has an empty catch block

A **catch** block that does not take any parameters must be the last in a series of **catch** blocks.

The following sample generates CS1017:

```
// CS1017.cs
using System;

namespace x
{
   public class b : Exception
   {
   }

   public class a
   {
      public static void Main()
      {
         try
         {
         }

         catch   // CS1017, must be last catch
         {
         }

         catch(b)
         {
            throw;
         }
      }
   }
}
```

See Also

Exception Handling Statements

Compiler Error CS1018

Keyword this or base expected

The compiler encountered an incomplete constructor declaration.

The following sample generates CS1018 and suggests several ways to resolve the error:

```
// CS1018.cs
namespace x
{
   public class C
   {
   }

   public class a : C
   {
      public a(int i)
      {
      }

      public a () :   // CS1018
      // possible resolutions:
      // public a () resolves by removing the colon
      // public a () : base() calls C's default constructor
      // public a () : this(1) calls the assignment constructor of class a
      {
      }

      public static int Main()
      {
         return 1;
      }
   }
}
```

Compiler Error CS1019

Overloadable unary operator expected

You have a unary operator that returns a value of another class. If you want this conversion, you need to do an **implicit** or **explicit** cast.

The following sample generates CS1019:

```
// CS1019.cs
public class ii
{
   int i
   {
      get
      {
         return 0;
      }
   }
}

public class a
{
   public static a operator ii(a aa)   // CS1019
   // try the following line instead
   //public static a operator ++(a aa)
   {
      return new a();
   }

   public static void Main()
   {
   }
}
```

Compiler Error CS1020

Overloadable binary operator expected

An attempt was made to define an operator overload, but the operator was not a binary operator, which takes two parameters.

The following sample generates CS1020:

```
// CS1020.cs
public class iii
{
   public static int operator ++(iii aa, int bb)   // CS1020, change ++ to +
   {
      return 0;
   }

   public static void Main()
   {
   }
}
```

Compiler Error CS1021

Integral constant is too large

A value assigned to an integral type is larger than the largest possible unsigned integer value.

The following sample generates CS1021:

```
// CS1021.cs
enum F : int
{
   a=(int)2590000000000000000000   // CS1021
}

public class clx
{
   public static void Main()
   {
   }
}
```

Compiler Error CS1022

Type or namespace definition, or end-of-file expected

A source-code file does not have a matching set of braces.

The following sample generates CS1022:

```
// CS1022.cs
namespace x
{
}
}   // CS1022
```

Compiler Error CS1023

Embedded statement cannot be a declaration or labeled statement

An embedded statement, such as the statements following an **if** statement, can contain neither declarations nor labeled statements.

The following sample generates CS1023 twice:

```
// CS1023.cs
public class a
{
   public static void Main()
   {
      if (1)
         int i;      // CS1023, declaration is not valid here

      if (1)
         xx : i++;   // CS1023, labeled statement is not valid here
   }
}
```

Compiler Error CS1024

Preprocessor directive expected

A line began with the pound symbol (#), but the subsequent string was not a valid preprocessor directive.

The following sample generates CS1024:

```
// CS1024.cs
#import System   // CS1024
```

Compiler Error CS1025

Single-line comment or end-of-line expected

A line with a preprocessor directive cannot have a multiline comment.

The following sample generates CS1025:

```
#if true /* hello
*/   // CS1025
#endif   // this is a good comment
```

CS1025 could also occur if you attempt some invalid preprocessor directive, as follows:

```
// CS1025.cs
#define a

class Sample
{
   static void Main()
   {
      #if a 1   // CS1025, invalid syntax
         System.Console.WriteLine("Hello, World!");
      #endif
   }
}
```

Compiler Error CS1026

) expected

An incomplete statement was found.

A common cause of this error is placing a statement, rather than an expression, within an inline expression in an ASP.NET page. For example, the following is incorrect:

```
<%=new TimeSpan(DateTime.Now.Ticks - new DateTime(2001, 1, 1).Ticks).Days;%>
```

The following is correct:

```
<%=new TimeSpan(DateTime.Now.Ticks - new DateTime(2001, 1, 1).Ticks).Days %>
```

It is interpreted as follows:

```
<% Response.Write(new TimeSpan(DateTime.Now.Ticks - new DateTime(2001, 1, 1).Ticks).Days); %>
```

The following sample generates CS1026:

```
// CS1026.cs
#if (a == b   // CS1026, add closing )
#endif

class x
{
   public static void Main()
   {
   }
}
```

Compiler Error CS1027

#endif directive expected

A matching **#endif** preprocessor directive was not found for a specified **#if** directive. Or, the compiler may have found a **#endregion** directive when there was no matching **#region** directive inside a **#if** block.

The following sample generates CS1027:

```
// CS1027.cs
#if true   // CS1027, uncomment next line to resolve
// #endif

namespace x
{
   public class clx
   {
      public static void Main()
      {
      }
   }
}
```

Compiler Error CS1028

Unexpected preprocessor directive

A preprocessor directive was found but not expected.

For example, a **#endif** was found with no preceding **#if**.

The following sample generates CS1028:

```
// CS1028.cs
#endif   // CS1028, no matching #if
namespace x
{
   public class clx
   {
      public static void Main()
      {
      }
   }
}
```

Compiler Error CS1029

#error: '*text*'

Displays the text of an error defined with the #error directive.

The following sample shows how to create a user-defined error:

```
// CS1029.cs
class Sample
{
   static void Main()
   {
      #error Let's give an error here   // CS1029
   }
}
```

Compiler Warning (level 1) CS1030

#warning: '*text*'

Displays the text of a warning defined with the #warning directive.

The following sample shows how to create a user-defined warning:

```
// CS1030.cs
class Sample
{
   static void Main()
   {
      #warning Let's give a warning here   // CS1030
   }
}
```

Compiler Error CS1031

Type expected

A type specifier is expected.

The following sample generates CS1031:

```
// CS1031.cs
namespace x
{
   public class ii
   {
   }

   public class a
   {
      public static operator +(a aa)   // CS1031
      // try the following line instead
      // public static ii operator +(a aa)
      {
         return new ii();
      }

      public static void Main()
      {
         e = new base;   // CS1031, not a type
         e = new this;   // CS1031, not a type
         e = new ();      // CS1031, not a type
      }
   }
}
```

Compiler Error CS1032

Cannot define/undefine preprocessor symbols after first token in file

The **#define** and **#undef** preprocessor directives must be used at the beginning of a program, before any other keywords, such as those used in the namespace declaration.

The following sample generates CS1032:

```
// CS1032.cs
namespace x
{
   public class clx
   {
      #define a   // CS1032, put before namespace
      public static void Main()
      {
      }
   }
}
```

Compiler Error CS1033

Compiler limit exceeded: File cannot exceed '*number*' lines

A source-code file exceeded the maximum allowable number of lines that the compiler can process. To resolve this error, create two or more source-code files from the original file. The maximum number of lines is 2,097,151 lines.

Compiler Error CS1034

Compiler limit exceeded: Line cannot exceed '*number*' characters

The limit for the number of characters allowed on a line is 2046.

Compiler Error CS1035

End-of-file found, '*/' expected

An opening comment delimiter was not matched with a closing delimiter.

The following sample generates CS1035:

```
// CS1035.cs
public class a
{
   public static void Main()
   {
   }
}
/*   // CS1035, needs closing comment
```

Compiler Error CS1036

(or . expected

The XML in a doc comment was badly formed.

The following sample generates CS1036:

```
// CS1036.cs
// compile with: /doc:x.xml
using System;

/// <seealso cref="System.String*"/>   // CS1036
public class Test
{
   public static void Main()
   {
   }
}
```

Compiler Error CS1037

Overloadable operator expected

When specifying a comment with /doc, the compiler encountered an invalid link.

The following sample generates CS1037:

```
// CS1037.cs
// compile with: /doc:x.xml

/// <summary>help text</summary>
public class A
{
   /// <summary>help text</summary>
   public static A operator / (A a1, A a2)
   {
      return null;
   }

   /// <seealso cref="A.operator@"/>   // CS1037
   // try the following line instead
   // /// <seealso cref="A.operator/"/>
   public static A Divide(A a1, A a2)
   {
      return a1 / a2;
   }

   /// <summary>help text</summary>
   public static void Main()
   {
   }
}
```

Compiler Error CS1038

#endregion directive expected

A #region directive did not have a matching #endregion directive.

The following sample generates CS1038:

```
// CS1038.cs
#region testing

public class clx
{
   public static void Main()
   {
   }
}
// CS1038
// uncomment the next line to resolve
// #endregion
```

Compiler Error CS1039

Unterminated string literal

The compiler detected an ill-formed string literal.

See 2.4.4.5 String literals for a discussion of the various ways to declare string literals.

The following sample generates CS1039:

```
// CS1039.cs
public class MyClass
{
   public static void Main()
   {
      string b = @"hello, world;   // CS1039
   }
}
```

To resolve the error, add the terminating quotation mark.

Compiler Error CS1040

Preprocessor directives must appear as the first non-whitespace character on a line

A preprocessor directive was found on a line and was not the first token on the line. A directive must be the first token on the line.

The following sample generates CS1040:

```
// CS1040.cs
/* Define a symbol, X */ #define X   // CS1040

// try the following two lines instead
// /* Define a symbol, X */
// #define X

public class MyClass
{
   public static void Main()
   {
   }
}
```

Compiler Error CS1041

Identifier expected, '*keyword*' is a keyword

A reserved word for the C# language was found where an identifier was expected. Replace the keyword with a user-specified identifier.

The following sample generates CS1041:

```
// CS1041a.cs
class MyClass
{
   public void f(int long)   // CS1041
   // Try the following instead:
   // public void f(int i)
   {
   }

   public static void Main()
   {
   }
}
```

When you are importing from another programming language that does not have the same set of reserved words, you can modify the reserved identifier with the @ prefix, as demonstrated in the following sample. For more information on using the @ prefix, see 2.4.2 Identifiers.

An identifier with an `@` prefix is called a verbatim identifier.

```
// CS1041b.cs
class MyClass
{
   public void f(int long)   // CS1041
   // Try the following instead:
   // public void f(int @long)
   {
   }

   public static void Main()
   {
   }
}
```

Compiler Error CS1043

{ or ; expected

A property accessor was declared incorrectly.

The following sample generates CS1043:

```
// CS1043.cs
public class MyClass
{
   public int DoSomething
   {
      get return 1;   // CS1043
      // try the following line instead
      // get { return 1;}
   }

   public static void Main()
   {
   }
}
```

Compiler Error CS1044

Cannot use more than one type in a for, using, fixed, or declaration statement

The compiler found an invalid statement.

The following sample generates CS1044:

```
// CS1044.cs
using System;

public class MyClass : IDisposable
{
   public void Dispose()
   {
      Console.WriteLine("Res1.Dispose()");
   }

   public static void Main()
   {
      using (MyClass mc1 = new MyClass(),
             MyClass mc2 = new MyClass())   // CS1044, remove an instantiation
      {
      }
   }
}
```

Compiler Error CS1055

An add or remove accessor expected

If your event is not declared as a field, it must define both **add** and **remove** accessor functions.

The following sample generates CS1055:

```
// CS1055.cs
delegate void del();
class Test
{
   public event del MyEvent
   {
      int i;   // CS1055
      // uncomment accessors and delete previous line to resolve
      // add
      // {
      //    MyEvent += value;
      // }
      // remove
      // {
      //    MyEvent -= value;
      // }
   }

   public static void Main()
   {
   }
}
```

Compiler Error CS1501

No overload for method '*method*' takes '*number*' arguments

A call was made to a class method, but there is no form of the method that takes the necessary number of arguments.

CS1501 could occur if you are calling a method on a class in a referenced assembly and if that method has default values on one or more of its parameters. C# does not let you create methods with a default value on a parameter, but another language that targets the runtime might. If a parameter (in a method in a referenced assembly) has a default value, you must still call the method and explicitly pass all parameters.

CS1501 could occur when you allocate memory for a delegate; you must also specify the name of the method that the delegate represents.

The following sample generates CS1501:

```
// CS1501.cs
public class a
{
   // declare the following constructor to resolve this CS1501
   /*
   public a(int i)
   {
   }
   */

   public static void Main()
   {
      a aa = new a(2);   // CS1501
   }
}
```

The following sample generates CS1501:

```
// CS1501a.cs
using System;

delegate string func(int i);   // declare the delegate

class a
{
   static func dt;   // class member-field of the declared delegate type

   public static void Main()
   {
      dt = new func();   // CS1501
      // try the following line instead
      // dt = new func(z);   // this works
      x(dt);
   }

   public static string z(int j)
   {
      Console.WriteLine(j);
      return j.ToString();
   }

   public static void x(func hello)
   {
      hello(8);
   }
}
```

Compiler Error CS1502

The best overloaded method match for '*declaration*' has some invalid arguments

This error occurs when the argument types being passed to the method do not match the parameter types of that method. If the called method is overloaded, then none of the overloaded versions has a signature that matches the argument types being passed.

To resolve this problem, do one of the following:

- Double-check the types of the arguments being passed. Make sure that they match the arguments of the method being called.
- If appropriate, convert any mismatched parameters using the System.Convert Class.
- If appropriate, cast any mismatched parameters to match the type that the method is expecting.
- If appropriate, define another overloaded version of the method to match the parameter types that are being sent.

The following sample generates CS1502:

```
// CS1502.cs
namespace x
{
   public class a
   {
      public a(char i)
      // try the following constructor instead
      // public a(int i)
      {
      }

      public static void Main()
      {
         a aa = new a(2222);   // CS1502
      }
   }
}
```

Compiler Error CS1503

Argument '*arg*': cannot convert from '*type1*' to '*type2*'

The type of one argument in a method does not match the type that was passed when the class was instantiated.

See CS1502 for a discussion on how to resolve this error.

Compiler Error CS1504

Source file '*file*' could not be opened ('*reason*')

A source file could not be opened or read by the compiler. The file may be locked by another application, or there could be some other operating system problem. The message includes the operating system's reason for why the compiler could not open or read the file.

Compiler Error CS1507

Cannot link resource file '*file*' when building a module

/resource or /linkresource was used in the same compilation with /target:module, which is not allowed.

Compiler Error CS1508

Resource identifier '*identifier*' has already been used in this assembly

In a compilation, the same identifier (***identifier***) was passed to two or more /resource or /linkresource compiler options.

For example, the following options would generate CS1508:

```
/resource:anyfile.bmp,DuplicatIdent /linkresource:a.bmp,DuplicatIdent
```

Compiler Error CS1509

Referenced file '*file*' is not an assembly; use '/addmodule' option instead

An output file (output file 1), produced in a compilation that used /target:module (does not have an assembly manifest), was specified to /reference. So, rather than appending an assembly to the assembly for the current program, the metadata information in output file 1 will be added to the assembly for the current program.

Compiler Error CS1510

A ref or out argument must be an lvalue

Only a variable can be passed as a ref parameter in a method call. A **ref** value is similar to passing a pointer.

The following sample generates CS1510:

```
// CS1510.cs
public class C
{
   public static int i = 0;

   public static void mf(ref int i)
   {
      i++;
   }

   public static void Main ()
   {
      mf (ref 1);   // CS1510, can't pass a number as a ref parameter
      // try the following to resolve the error
      // mf (ref i);
   }
}
```

Compiler Error CS1511

Keyword base is not available in a static method

The base keyword was used in a static method.

The following sample generates CS1511:

```
// CS1511.cs
public class A
{
   public int j = 0;
}

class C : A
{
   public void func()
   {
      base.j = 3;   // base allowed here
   }

   public static int Main ()
   {
      base.j = 3;   // CS1511
      return 1;
   }
}
```

Compiler Error CS1512

Keyword base is not available in the current context

The base keyword was used outside of a method, property, or constructor.

The following sample generates CS1512:

```
// CS1512.cs
using System;

class Base {}

class CMyClass : Base
{
   private String xx = base.ToString();   // CS1512
   // try putting this initialization in the constructor instead
   // public CMyClass()
   // {
   //    xx = base.ToString();
   // }

   public static void Main()
   {
      CMyClass z = new CMyClass();
   }
}
```

Compiler Error CS1513

} expected

The compiler expected a closing curly brace (`}`) that was not found.

The following sample generates CS1513:

```
// CS1513
namespace y   // CS1513, no close curly brace
{
   class x
   {
      public static void Main()
      {
      }
   }
```

Compiler Error CS1514

{ expected

The compiler expected an opening curly brace (`{`) that was not found.

The following sample generates CS1514:

```
// CS1514.cs
namespace x
// CS1514, no open curly brace
```

Compiler Error CS1515

'in' is expected

In a foreach, in statement, the "in" part is missing.

The following sample generates CS1515:

```
using System;

class Driver
{
   static void Main()
   {
      int[] arr = new int[] {1, 2, 3};

      // try the following line instead
      // foreach (int x in arr)
      foreach (int x arr)      // CS1515, "in" is missing
      {
         Console.WriteLine(x);
      }
   }
}
```

Compiler Error CS1517

Invalid preprocessor expression

The compiler encountered an invalid preprocessor expression.

For more information, see Preprocessor Directives.

The following sample shows some valid and invalid preprocessor expressions:

```
// CS1517.cs
#if symbol      // OK
#endif
#if !symbol     // OK
#endif
#if (symbol)    // OK
#endif
#if true        // OK
#endif
#if false       // OK
#endif
#if 1           // CS1517
#endif
#if ~symbol     // CS1517
#endif
#if *           // CS1517
#endif

class x
{
   public static void Main()
   {
   }
}
```

Compiler Error CS1518

Expected class, delegate, enum, interface, struct, or union

A declaration was found that is not supported in a namespace. Inside a namespace, the compiler accepts only classes, structs, enums, interfaces, namespaces, and delegates.

The following sample generates CS1518:

```
// CS1518.cs
namespace x
{
   sealed class c1 {};      // OK
   namespace f2 {};         // OK
   sealed f3 {};            // CS1518
}
```

Compiler Error CS1519

Invalid token '*token*' in class, struct, or interface member declaration

Any class, struct, or interface member declaration that contains invalid modifiers before the type will generate this error.

The following sample generates CS1519:

```
// CS1519.cs
public class IMyInterface
{
   checked void f4();   // CS1519
   lock void f5();      // CS1519
   namespace;           // CS1519
   int i;               // OK
}
```

Compiler Error CS1520

Class, struct, or interface method must have a return type

A method that is declared in a class, struct, or interface must have an explicit return type.

The following sample generates CS1520:

```
// CS1520a.cs
public class x
{
   f7()   // CS1520, needs return type
   // try the following definition
   // void f7()
   {
   }

   public static void Main()
   {
   }
}
```

Alternatively, this error might be encountered when the case of a constructor's name differs from that of the class or struct declaration, as in the following sample:

```
// CS1520b.cs
public class Class1
{
   public class1()   // CS1520, incorrect case
   {
   }
   static void Main()
   {
   }
}
```

Compiler Error CS1521

Invalid base type

A base class specification was ill formed.

The following sample generates CS1521:

```
// CS1521.cs
class CMyClass
{
   public static void Main()
   {
   }
}

class CMyClass1 : CMyClass     // OK
{
}

class CMyClass2 : CMyClass[]   // CS1521
{
}

class CMyClass3 : CMyClass*    // CS1521
{
}
```

Compiler Warning (level 1) CS1522

Empty switch block

The compiler detected a switch block with no **case** or **default** statement. A **switch** block must have one or more **case** or **default** statements.

The following sample generates CS1522:

```
// CS1522.cs
// compile with: /W:1
using System;
class x
{
   public static void Main()
   {
      int i = 6;

      switch(i)   // CS1522
      {
         // add something to the switch block, for example:
         /*
         case (5):
            Console.WriteLine("5");
            return;
         default:
            Console.WriteLine("not 5");
            return;
         */
      }
   }
}
```

Compiler Error CS1523

The keyword case or default must precede code in switch block

Code was found in a switch statement that was part of neither a **case** nor a **default** block.

The following sample generates CS1523:

```
// CS1523.cs
class x
{
   public static void Main()
   {
      int i = 0;

      switch (i)
      {
         i = 6;   // CS1523, code not under a label
         // try the following
         /*
         default:
            i = 6;
         */
      }
   }
}
```

Compiler Error CS1524

Expected catch or finally

A **try** block must be followed by a **catch** or **finally** block.

The following sample generates CS1524:

```
// CS1524.cs
class x
{
   public static void Main()
   {
      try
      {
         // Code here
      }
      catch
      {
      }
      try
      {
         // Code here
      }
      finally
      {
      }
      try
      {
         // Code here
      }
   }     // CS1524, missing catch or finally
}
```

See Also

Exception Handling Statements

Compiler Error CS1525

Invalid expression term '*character*'

The compiler detected an invalid character in an expression.

The following sample generates CS1525:

```
// CS1525.cs
class x
{
   public static void Main()
   {
      int i = 0;
      i = i +   // OK - identifier
      'c' +     // OK - character
      (5) +     // OK - parenthesis
      [ +       // CS1525, operator not a valid expression element
      throw +   // CS1525, keyword not allowed in expression
      void;     // CS1525, void not allowed in expression
   }
}
```

An empty label can also generate CS1525, as in the following sample:

```
// CS1525b.cs
using System;
public class MyClass
{
   public static void Main()
   {
      goto FoundIt;
      FoundIt:      // CS1525
      // Uncomment the following line to resolve:
      // System.Console.Write("Hello");
   }
}
```

Compiler Error CS1526

'new' expression requires () or [] after type

The new operator, used to dynamically allocate memory for an object, was not specified correctly.

The following sample shows how to use **new** to allocate space for an array and an object.

```
// CS1526.cs
public class y
{
   public static int i = 0;
   public int myi = 0;
}

public class z
{
   public static void Main()
   {
      y py = new y;   // CS1526
      y[] aoys = new y[10];   // Array of Ys

      for (int i = 0; i < aoys.Length; i++)
         aoys[i] = new y();   // an object of type y
   }
}
```

Compiler Error CS1527

Namespace elements cannot be explicitly declared as private, protected, or protected internal

Type declarations in a namespace can have either public or internal access. If no accessibility is specified, **internal** is the default.

The following sample generates CS1527:

```
// CS1527.cs
namespace Sample
{
   private class C1 {};             // CS1527
   protected class C2 {};           // CS1527
   protected internal class C3 {};  // CS1527
}
```

Compiler Error CS1528

Expected ; or = (cannot specify constructor arguments in declaration)

A reference to a class was formed as if an object to the class was being created. For example, there was an attempt to pass a variable to a constructor. Use the new operator to create an object of a class.

The following sample generates CS1528:

```
// CS1528.cs
using System;

public class B
{
   public B(int i)
   {
      _i = i;
   }

   public void PrintB()
   {
      Console.WriteLine(_i);
   }

   private int _i;
}

public class mine
{
   public static void Main()
   {
      B b(3);   // CS1528, reference is not an object
      // try one of the following
      // B b;
      // or
      // B bb = new B(3);
      // bb.PrintB();
   }
}
```

Compiler Error CS1529

A using clause must precede all other namespace elements

Statements with a using clause must appear first in a namespace.

The following sample generates CS1529:

```
// CS1529.cs
namespace x
{
   namespace subspace
   {
      using Microsoft;

      class SomeClass
      {
      };

      using Microsoft;        // CS1529, place before class definition
   }

   using System.Reflection;   // CS1529, place before namespace 'subspace'
}

using System;                 // CS1529, place at the beginning of the file
```

Compiler Error CS1530

Keyword new not allowed on namespace elements

It is not necessary to specify the new keyword on any construct that is in a namespace.

The following sample generates CS1530:

```
// CS1530.cs
namespace a
{
   new class i   // CS1530
   {
   }

   // try the following instead
   class ii
   {
      public static void Main()
      {
      }
   }
}
```

Compiler Error CS1533

Invoke cannot be called directly on a delegate

The **Invoke** method cannot be called on a delegate. For example:

```
// CS1533.cs
using System;

public class A
{
   public static void Main()
   {
   }
}

public delegate void B(A a);

internal class C
{
   public void Baz(A a, B b)
   {
      b.Invoke(a);   // CS1533
      // try the following line instead
      // b(a);
   }
}
```

Compiler Error CS1534

Overloaded binary operator '*operator*' only takes two parameters

The definition of a binary overloadable operator must take two parameters.

The following sample generates CS1534:

```
// CS1534.cs
class MyClass
{
   public static MyClass operator - (MyClass MC1, MyClass MC2, MyClass MC3)   // CS1534
   // try the following line instead
   // public static MyClass operator - (MyClass MC1, MyClass MC2)
   {
      return new MyClass();
   }

   public static int Main()
   {
      return 1;
   }
}
```

Compiler Error CS1535

Overloaded unary operator '*operator*' only takes one parameter

The definition of a unary overloadable operator must take one parameter.

The following sample generates CS1535:

```
// CS1535.cs
class MyClass
{
   // uncomment the method parameter to resolve CS1535
   public static MyClass operator ++ (/*MyClass MC1*/)   // CS1535
   {
      return new MyClass();
   }

   public static int Main()
   {
      return 1;
   }
}
```

Compiler Error CS1536

Invalid parameter type void

It is not necessary or valid to specify a void parameter other than a void pointer.

The following sample generates CS1536:

```
// CS1536.cs
class a
{
   public static int x( void )   // CS1536
   // try the following line instead
   // public static int x()
   {
      return 0;
   }

   public static void Main()
   {
   }
}
```

Compiler Error CS1537

The using alias '*alias*' appeared previously in this namespace

You defined a symbol twice as an alias for a namespace. A symbol can only be defined once.

The following sample generates CS1537:

```
// CS1537.cs
namespace x
{
   using System;
   using Object = System.Object;
   using Object = System.Object;   // CS1537, delete this line to resolve
   using System = System;
}
```

Compiler Error CS1540

Cannot access protected member '*member*' via a qualifier of type '*type1*'; the qualifier must be of type '*type2*' (or derived from it)

Although a derived class can access protected members of its base class, it cannot do so through an instance of the base class.

The following sample generates CS1540:

```
// CS1540.cs
public class Base
{
   protected void func()
   {
   }
}

public class Derived : Base
{
   public static void test(Base anotherInstance)
   // the method declaration could be changed as follows
   // public static void test(Derived anotherInstance)
   {
      anotherInstance.func();   // CS1540
   }
}

public class Tester : Derived
{
   public static void Main()
   {
      Base pBase = new Base();
      // the allocation could be changed as follows
      // Derived pBase = new Derived();
      test(pBase);
   }
}
```

Compiler Error CS1541

Invalid include option: '*symbol*' — cannot include directories

The compiler detected an attempt to specify a directory rather than a specific file. For example, when you use the /reference compiler option, you must specify a file; it is not possible to specify a directory.

For example, the following parameter passed to the **/reference** compiler option would generate CS1541:

```
/reference:c:\
```

Compiler Error CS1542

'*dll*' cannot be added to this assembly because it already is an assembly; use '/R' option instead

The file that was referenced with the /addmodule compiler option was not built with /target:module; use /reference to reference the file in this compilation.

Compiler Error CS1545

Property, indexer, or event '*property*' is not supported by the language; try directly calling accessor methods '*set accessor*' or '*get accessor*'

Your code is consuming an object that has a default indexer and tried to use the indexed syntax. To resolve this error, call the property's get or set accessor method.

The following sample generates CS1545:

```
// CPP1545.cpp
// compile with: /clr /LD
#using <mscorlib.dll>
using namespace System;

public __gc class Employee {
public:
   Employee(String * s, int d) {
      _name = s;
      _dept = d;
   };

   __property String * get_name() {
      return _name;
   }

   __property int get_dept() {
      return _dept;
   }

private:
   String * _name;
   int _dept;
};

public __gc class Manager {
public:
   __property Employee * get_Report(String * s) {
      for (pEmp = Reports ; pEmp && (pEmp -> emp -> name != s);
         pEmp = pEmp -> next);
      if (pEmp)
         return pEmp -> emp;
      else
         return 0;
   }

   __property void set_Report(String* s, Employee* e) {
     for(pEmp = Reports; pEmp && (pEmp -> emp -> name != s);
      pEmp = pEmp -> next);
      if (!pEmp) {
         EmpList * emp1 = new EmpList;
         emp1 -> emp = e;
         emp1 -> next = Reports;
         Reports = emp1;
      }
   }
private:
   static Manager() {
      Reports = 0;
   }
   __gc struct EmpList {
   Employee * emp;
   EmpList * next;
   };
   EmpList * pEmp;
   static EmpList * Reports;
};
```

And then the C# program:

```
// CS1545.cs
// compile with: /r:CPP1545.dll
class x {
   public static void Main() {
      Manager Ed = new Manager();
      Employee Bob = new Employee("Bob Smith", 12);

      // track Ed's reports
      Ed.Report( Bob.name ) = Bob;   // CS1545
      // try the following line instead
      // Ed.set_Report( Bob.name, Bob);
   }
}
```

Compiler Error CS1546

Property, indexer, or event '*property*' is not supported by the language; try directly calling accessor method '*accessor*'

Your code is consuming an object that has a default indexed property and tried to use the indexed syntax. To resolve this error, call the property's accessor method.

The following sample generates CS1546. It consists of a .cpp file, which compiles to a .dll, and a .cs file, which uses that .dll.

```
// CPP1546.cpp
// compile with: /clr /LD
#using <mscorlib.dll>

using namespace System;
public __gc class MCPP
{
public:
   __property int get_Prop(int i, int b){return i;}
};
```

This is the C# file:

```
// CS1546.cs
// compile with: /r:CPP1546.dll
using System;
public class Test
{
   public static void Main()
   {
      int i = 0;
      MCPP mcpp = new MCPP();
      i = mcpp.Prop(1,1);   // CS1546
      // try the following line instead
      // i = mcpp.get_Prop(1,1);
   }
}
```

See Also

Indexers | Defining and Using Properties

Compiler Error CS1547

Keyword 'void' cannot be used in this context

The compiler detected an invalid use of the void keyword.

The following sample generates CS1547:

```
// CS1547.cs
public class MyClass
{
   void BadMethod()
   {
      void i;   // CS1547, cannot have variables of type void
   }

   public static void Main()
   {
   }
}
```

Compiler Error CS1548

Cryptographic failure while signing assembly '*assembly*' — '*reason*'

This error occurs when the signing of the assembly fails for some reason. There are several possible reasons why this could occur. The most common are that the assembly key file directive (usually in the AssemblyInfo.cs file) has an incorrect path to the key file, or the path is not a relative path to the build directory, or there is something wrong with the key file. For example, if the key file is in the project folder, the following should appear in AssemblyInfo.cs:

```
[assembly: AssemblyKeyFile("..\\..\\SnkFileName.snk")]
```

For more information, see KB Article Q328379.

Compiler Error CS1549

Appropriate cryptographic service not found

An RSA compatible cryptographic service provider was not installed.

Compiler Error CS1551

Indexers must have at least one parameter

An indexer that takes no arguments was declared.

The following sample generates CS1551:

```
// CS1551.cs
public class MyClass
{
   int intI;

   int this[]   // CS1551
   // try the following line instead
   // int this[int i]
   {
      get
      {
         return intI;
      }
      set
      {
         intI = value;
      }
   }

   public static void Main()
   {
   }
}
```

Compiler Error CS1552

Array type specifier, [], must appear before parameter name

The position of the array type specifier is after the variable name in the array declaration.

The following sample generates CS1552:

```
// CS1552.cs
public class C
{
   public static void Main(string args[])   // CS1552
   // try the following line instead
   // public static void Main(string [] args)
   {
   }
}
```

Compiler Error CS1553

Declaration is not valid; use '*modifier* operator <dest-type> (...' instead

The return type for an operator must immediately precede the parameter list, and *modifier* is either **implicit** or **explicit**.

The following sample generates CS1553:

```
// CS1553.cs
class MyClass
{
   public static int implicit operator (MyClass f)   // CS1553
   // try the following line instead
   // public static implicit operator int (MyClass f)
   {
      return 6;
   }

   public static void Main()
   {
   }
}
```

Compiler Error CS1554

Declaration is not valid; use '<type> operator *op* (...' instead

The return type for a user-defined operator must appear before the keyword operator.

The following sample generates CS1554:

```
// CS1554.cs
class MyClass
{
   public static operator ++ MyClass (MyClass f)    // CS1554
   // try the following line instead
   // public static MyClass operator ++ (MyClass f)
   {
      return new MyClass ();
   }

   public static void Main()
   {
   }
}
```

Compiler Error CS1555

Could not find '*class*' for Main method in program '*program*'

A class was specified to the /main compiler option, but the class name was not found in the source code.

Compiler Error CS1556

'*construct*' specified for Main method must be a valid class or struct

The /main compiler option was passed an identifier that was not a class name.

Compiler Error CS1557

'*class*' specified for Main method not found in program '*output file1*', was found in '*output file2*'

The /main compiler option was specified for one output file in a multi-output file compilation. However, the class was not found in the source code for the /main compilation; it was found in a source code file for one of the other output files in the compilation.

Compiler Error CS1558

'*class*' does not have a suitable Main method

The /main compiler option specified a class in which to look for a Main method. However, the Main method was not defined correctly.

Compiler Error CS1559

'*object*' is imported and cannot be used as an entry point for program '*program*'

An invalid class was specified to the /main compiler option; the class cannot be used as a location for the Main method.

Compiler Error CS1560

Filename specified for #line is too long

The file name that was specified with #line exceeded **_MAX_PATH** (256 characters) or the line on which **#line** was found exceeded 2000 characters.

Compiler Error CS1562

Outputs without source must have the /out option specified

The compilation could create an output file, but there was no source code file as input from which the name of the output file could be implied. For example, you may be trying to compile a metadata- or resource-only file.

Use the /out compiler option to specify the name of the output file.

Compiler Error CS1563

Output '*output file*' does not have any source files

The /out compiler option was specified, but no source code files follow. You should either not use **/out**, or you should specify the source code files following **/out**.

Compiler Error CS1565

Conflicting options specified: Win32 resource file; Win32 icon

It is not valid to specify both the /win32res compiler option and the /win32ico compiler option in the same compilation.

Compiler Error CS1566

Error reading resource file '*file*' — '*reason*'

The compiler had trouble with the file name passed to the /resource compiler option.

Compiler Error CS1567

Error generating Win32 resource: '*file*'

Your compilation either used the /win32icon compiler option or did not use /win32res, which causes the compiler to generate a file that contains resource information, but the compiler was unable to create the file due to insufficient disk space or some other error.

If you are unable to resolve the file-generation problem, you could use /win32res, which does not generate a file that contains resource information.

Compiler Error CS1569

Error generating XML documentation file '*file*' ('*reason*')

When generating an XML file with /doc, the file could not be opened and written to for the indicated reason.

Compiler Warning (level 1) CS1570

XML comment on '*construct*' has badly formed XML — '*reason*'

When using /doc, any comments in the source code must be in XML. Any error with your XML markup will generate CS1570. For example:

- If you are passing a string to a **cref**, such as in an <exception> tag, the string must be enclosed in double quotation marks.
- If you are using a tag, such as <seealso>, which does not have a closing tag, you must specify a forward slash before the closing angle bracket.
- If you need to use a greater-than or less-than symbol in the text of description, you need to represent them with **>** or **<**.
- The file or path attribute on an <include> tag was missing or improperly formed.

The following sample generates CS1570:

```
// CS1570.cs
// compile with: /W:1
namespace ns
{
   // the following line generates CS1570
   /// <summary> returns true if < 5 </summary>
   // try this instead
   // /// <summary> returns true if &lt;5 </summary>

   public class MyClass
   {
      public static void Main ()
      {
      }
   }
}
```

Compiler Warning (level 2) CS1571

XML comment on '*construct*' has a duplicate param tag for '*parameter*'

When using the /doc compiler option, multiple comments were found for the same method parameter. Remove one of the duplicate lines.

The following sample generates CS1571:

```
// CS1571.cs
// compile with: /W:2 /doc:x.xml

/// <summary>help text</summary>
public class MyClass
{
   /// <param name='Int1'>Used to indicate status.</param>
   /// <param name='Char1'>An initial.</param>
   /// <param name='Int1'>Used to indicate status.</param> // CS1571
   public static void MyMethod(int Int1, char Char1)
   {
   }

   /// <summary>help text</summary>
   public static void Main ()
   {
   }
}
```

Compiler Warning (level 2) CS1572

XML comment on '*construct*' has a param tag for '*parameter*', but there is no parameter by that name

When using the /doc compiler option, a comment was specified for a parameter that does not exist for the method. Change the value passed to the name attribute or remove one of the comment lines.

The following sample generates CS1572:

```
// CS1572.cs
// compile with: /W:2 /doc:x.xml

/// <summary>help text</summary>
public class MyClass
{
   /// <param name='Int1'>Used to indicate status.</param>
   /// <param name='Char1'>Used to indicate status.</param>
   /// <param name='Char2'>???</param> // CS1572
   public static void MyMethod(int Int1, char Char1)
   {
   }

   /// <summary>help text</summary>
   public static void Main ()
   {
   }
}
```

Compiler Warning (level 4) CS1573

Parameter '*parameter*' has no matching param tag in XML comment (but other parameters do)

When using the /doc compiler option, a comment was specified for some but not all parameters in a method. You may have forgotten to enter a comment for these parameters.

The following sample generates CS1573:

```
// CS1573.cs
// compile with: /W:4
public class MyClass
{
   /// <param name='Int1'>Used to indicate status.</param>
   // enter a comment for Char1?
   public static void MyMethod(int Int1, char Char1)
   {
   }

   public static void Main ()
   {
   }
}
```

Compiler Warning (level 1) CS1574

XML comment on '*construct*' has cref attribute '*item*' that could not be found

A string passed to a cref tag, for example, within an <exception> tag, referred to a member that is not available within the current build environment. The string that you pass to a cref tag must be the syntactically correct name of a member or field.

For more information, see Recommended Tags for Documentation Comments.

The following sample generates CS1574:

```
// CS1574.cs
// compile with: /W:1 /doc:x.xml
using System;

/// <exception cref="System.Console.WriteLin">An exception class.</exception>   // CS1574
// instead, uncomment and try the following line
// /// <exception cref="System.Console.WriteLine">An exception class.</exception>
class EClass : Exception
{
}

class TestClass
{
   public static void Main()
   {
      try
      {
      }
      catch(EClass)
      {
      }
   }
}
```

Compiler Error CS1575

A stackalloc expression requires [] after type

The size of the requested allocation, with stackalloc, must be specified in square brackets.

The following sample generates CS1575:

```
// CS1575.cs
// compile with: /unsafe
public class MyClass
{
   unsafe public static void Main()
   {
      int *p = stackalloc int (30);   // CS1575
      // try the following line instead
      // int *p = stackalloc int [30];
   }
}
```

Compiler Error CS1576

The line number specified for #line directive is missing or invalid

The compiler detected an error with the value passed to the #line directive.

The following sample generates CS1576:

```
// CS1576.cs
public class MyClass
{
   static void Main()
   {
      #line "abc.sc"         // CS1576
      // try the following line instead
      //#line  101 "abc.sc"
      intt i;  // error will be reported on line 101
   }
}
```

Compiler Error CS1577

Assembly generation failed — *reason*

The assembly-generation part of the compilation failed. See the errors documentation for the alink utility (Al.exe) for more information.

Compiler Error CS1578

Filename, single-line comment or end-of-line expected

After a #line directive, only a file name (in double quotation marks) or a single-line comment is allowed.

The following sample generates CS1578:

```
// CS1578.cs
class MyClass
{
   static void Main()
   {
      #line 101 abc.cs   // CS1578
      // try the following line instead
      //#line 101 "abc.cs"
      intt i;          // error will be reported on line 101
   }
}
```

Compiler Error CS1579

foreach statement cannot operate on variables of type '*type1*' because '*type2*' does not contain a definition for '*member*', or it is inaccessible

A data type (*type1*) does not contain a member that allows foreach to iterate through *type2*.

The following sample generates CS1579:

```
// CS1579.cs
using System;

public class C1
{
   public int GetEnumerator ()
   {
      return 0;
   }
}

public class MainClass
{
   public static void Main ()
   {
      C1 c1 = new C1();

      foreach (int x in c1)   // CS1579
      {
      }
   }
}
```

Compiler Warning (level 1) CS1580

Invalid type for parameter '*parameter number*' in XML comment cref attribute

When attempting to reference an overload form of a method, the compiler detected a syntax error. Typically, this indicates that the parameter name, and not the type, was specified. A malformed line will appear in the generated XML file.

The following sample generates CS1580:

```
// CS1580.cs
// compile with: /W:1 /doc:x.xml
using System;

/// <seealso cref="Test(i)"/>   // CS1580
// try the following line instead
// /// <seealso cref="Test(int)"/>
public class MyClass
{
   /// <summary>help text</summary>
   public static void Main()
   {
   }

   /// <summary>help text</summary>
   public void Test(int i)
   {
   }

   /// <summary>help text</summary>
   public void Test(char i)
   {
   }
}
```

Compiler Warning (level 1) CS1581

Invalid return type in XML comment cref attribute

When attempting to reference a method, the compiler detected an error due to an invalid return type.

The following sample generates CS1581:

```
// CS1581.cs
// compile with: /W:1 /doc:x.xml

/// <summary>help text</summary>
public class MyClass
{
   /// <summary>help text</summary>
   public static void Main()
   {
   }

   /// <summary>help text</summary>
   public static explicit operator int(MyClass f)
   {
      return 0;
   }
}

/// <seealso cref="MyClass.explicit operator intt(MyClass)"/>   // CS1581
// try the following line instead
// /// <seealso cref="MyClass.explicit operator int(MyClass)"/>
public class MyClass2
{
}
```

Compiler Error CS1583

'*file*' is not a valid Win32 resource file

A file was passed to the /win32res compiler option that was not a valid resource file.

Compiler Warning (level 1) CS1584

XML comment on '*member*' has syntactically incorrect cref attribute '*invalid_syntax*'

One of the parameters passed to a tag for documentation comments had invalid syntax.

The following sample generates CS1584:

```
// CS1584.cs
// compile with: /W:1 /doc:x.xml
/// <remarks>Test class</remarks>
public class Test
{
   /// <remarks>
   /// Called in <see cref="Test::Main"/>.   // CS1584, try Test.Main
   /// </remarks>
   public static void Test2()
   {
   }

   /// <remarks>Main method</remarks>
   public static void Main()
   {
      Test2();
   }
}
```

Compiler Error CS1585

Member modifier '*keyword*' must precede the member type and name

The access specifier in a method signature did not occur in the correct location.

The following sample generates CS1585:

```
// CS1585.cs
public class Class1
{
   public void static Main(string[] args)   // CS1585
   // try the following line instead
   // public static void Main(string[] args)
   {
   }
}
```

Compiler Error CS1586

Array creation must have array size or array initializer

An array was declared incorrectly.

The following sample generates CS1586:

```
// CS1586.cs
using System;
class MyClass
{
   public static void Main()
   {
      int[] a = new int[];   // CS1586
      // try the following line instead
      int[] b = new int[5];
   }
}
```

Compiler Warning (level 2) CS1587

XML comment is not placed on a valid language element

Recommended tags for documentation comments are not allowed on all language elements. For example, a tag is not allowed on a namespace.

The following sample generates CS1587:

```
// CS1587.cs
// compile with: /W:2 /doc:x.xml

/// <summary>test</summary>   // CS1587, tag not allowed on namespace
namespace MySpace
{
   class MyClass
   {
      public static void Main()
      {
      }
   }
}
```

See Also

Recommended Tags for Documentation Comments

Compiler Error CS1588

Cannot determine common language runtime directory -- '*reason*'

The compiler cannot retrieve the directory that the .NET Framework common language runtime is installed in. This error indicates that the common language runtime is improperly installed.

Compiler Warning (level 1) CS1589

Unable to include XML fragment '*fragment*' of file '*file*' -- *reason*

The syntax (*fragment*) of a <include> tag, which referenced a file (*file*), was incorrect for the specified ***reason***.

A malformed line will be placed in the generated XML file.

The following sample generates CS1589:

```
// CS1589.cs
// compile with: /W:1 /doc:CS1589_out.xml

/// <include file='CS1589.doc' path='MyDocs/MyMembers[@name="test"]/' />   // CS1589
// try the following line instead
// /// <include file='CS1589.doc' path='MyDocs/MyMembers[@name="test"]/*' />
class Test
{
   public static void Main()
   {
   }
}
```

Compiler Warning (level 1) CS1590

Invalid XML include element -- Missing file attribute

A path or doc attribute, passed to the <include> tag, was missing or incomplete.

The following sample generates CS1590:

```
// CS1590.cs
// compile with: /W:1 /doc:x.xml

/// <include path='MyDocs/MyMembers[@name="test"]/*' />   // CS1590
// try the following line instead
// /// <include file='x.doc' path='MyDocs/MyMembers[@name="test"]/*' />
class Test
{
   public static void Main()
   {
   }
}
```

Compiler Warning (level 4) CS1591

Missing XML comment for publicly visible type or member '*Type_or_Member*'

The /doc compiler option was specified, but one or more constructs did not have comments.

The following sample generates CS1591:

```
// CS1591.cs
// compile with: /W:4 /doc:x.xml

/// text
public class Test
{
   // /// text
   public static void Main()   // CS1591, remove "//" from previous line
   {
   }
}
```

Compiler Warning (level 1) CS1592

Badly formed XML in included comments file -- '*reason*'

A problem, reported as ***reason***, was found in the file specified by the <include> tag.

Compiler Error CS1593

Delegate '*del*' does not take '*number*' arguments

The number of arguments passed to a delegate invocation does not agree with the number of parameters in the delegate declaration.

The following sample generates CS1593:

```
// CS1593.cs
using System;
delegate string func(int i);   // declare delegate

class a
{
   public static void Main()
   {
      func dt = new func(z);
      x(dt);
   }

   public static string z(int j)
   {
      Console.WriteLine(j);
      return j.ToString();
   }

   public static void x(func hello)
   {
      hello(8, 9);   // CS1593
      // try the following line instead
      // hello(8);
   }
}
```

Compiler Error CS1594

Delegate '*delegate*' has some invalid arguments

The type of an argument passed to a delegate invocation does not agree with the type of the parameter in the delegate declaration.

The following sample generates CS1594:

```
// CS1594.cs
using System;
delegate string func(int i);   // declare delegate

class a
{
   public static void Main()
   {
      func dt = new func(z);
      x(dt);
   }

   public static string z(int j)
   {
      Console.WriteLine(j);
      return j.ToString();
   }

   public static void x(func hello)
   {
      hello("8");   // CS1594
      // try the following line instead
      // hello(8);
   }
}
```

Compiler Warning (level 1) CS1595

'*type*' is defined in multiple places; using definition from '*assembly*'

A project referenced two assemblies, each of which contains the same type definition. The compiler chose one of the assemblies from which to take the definition of type based on the order of files passed to /reference.

See Add Reference Dialog Box for information on how to add a reference in the development environment.

The following sequence of compilations generates CS1595:

```
// CS1595_1.cs
// compile with: /W:1 /target:library
namespace N1
{
   public class C1
   {
      public C1()
      {
         System.Console.WriteLine("Hi1");
      }
   }
}
```

```
// CS1595_2.cs
// compile with: /t:library
namespace N1
{
   public class C1
   {
      public C1()
      {
         System.Console.WriteLine("Hi2");
      }
   }
}
```

```
// CS1595_3.cs
// compile with: /r:CS1595_1.dll,CS1595_2.dll
class Test
{
   public static void Main(string[] args)
   {
      N1.C1 obj = new N1.C1();   // CS1595
   }
}
```

Compiler Warning (level 1) CS1596

XML documentation not updated during this incremental rebuild; use /incremental- to update XML documentation

When using incremental builds (/incremental) with /doc, /doc is essentially ignored; the source code files will not be processed for comments and the XML file will not be updated.

Compiler Error CS1597

Semicolon after method or accessor block is not valid

Semicolons are not needed (or allowed) to end a method or accessor block.

The following sample generates CS1597:

```
// CS1597.cs
class TestClass
{
   public static void Main()
   {
   };   // CS1597, remove semicolon
}
```

Compiler Warning (level 1) CS1598

XML parser could be not be loaded for the following reason: '*reason*'. The XML documentation file '*file*' will not be generated.

The /doc option was specified, but the compiler could not find and load msxml3.dll. Make sure that the file msxml3.dll is installed and registered.

Compiler Error CS1599

Method or delegate cannot return type '*type*'

Some types in the base class library, for example, TypedReference and ArgIterator cannot be used as return types.

The following sample generates CS1599:

```
// CS1599.cs
using System;

class MyClass
{
   public static void Main()
   {
   }

   public TypedReference Test1()   // CS1599
   {
      return null;
   }

   public ArgIterator Test2()   // CS1599
   {
      return null;
   }
}
```

Compiler Error CS1600

Compilation cancelled by user

A compilation with the C# compiler was cancelled while using the Visual Studio IDE.

Compiler Error CS1601

Method or delegate parameter cannot be of type '*type*'

Some types in the base class library, for example, TypedReference and ArgIterator cannot be used as parameter types with the ref or out keywords.

The following sample generates CS1601:

```
// CS1601.cs
using System;

class MyClass
{
   public void Test1 (ref TypedReference t)   // CS1601
   {
   }

   public void Test2 (out ArgIterator t)   // CS1601
   {
   }
}
```

Compiler Error CS1602

'*symbol*' is already defined in this assembly

A module contains a symbol for a type, and that module is included in the compilation of an assembly that also defines the same symbol. Change the name of one of the symbols.

The following sequence generates C1602. First, create a module:

```
// CS1602.cs
// compile with: /target:module
namespace Test
{
   public class B
   {
   }
}
```

Then, create an assembly and add the module:

```
// CS1602a.cs
// compile with: /addmodule:CS1602.netmodule

namespace Test
{
   public class B   // CS1602
   // try the following line instead
   // public class C
   {
   }

   public class Client
   {
      static public void Main()
      {
      }
   }
}
```

Compiler Error CS1603

Incremental build requires a target name specified with /out

You must specify /out when using the /incremental option.

Compiler Error CS1613

The managed coclass wrapper class '*class*' for interface '*interface*' is invalid or cannot be found (are you missing an assembly reference?)

An attempt was made to instantiate a COM object from an interface. The interface has the **ComImport** and **CoClass** attributes, but the compiler cannot find the type given for the **CoClass** attribute.

To resolve this error, you can try one of the following:

- Add a reference to the assembly that has the coclass (most of the time the interface and coclass should be in the same assembly). See /reference or Add Reference Dialog Box for information.
- Fix the **CoClass** attribute on the interface.

The following sample demonstrates correct usage of **CoClassAttribute**:

```
// CS1613.cs
using System;
using System.Runtime.InteropServices;

[Guid("1FFD7840-E82D-4268-875C-80A160C23296")]
[ComImport()]
[CoClass(typeof(A))]
public interface IA{}
public class A : IA {}

public class AA
{
   public static void Main()
   {
      IA i;
      i = new IA(); // This is equivalent to new A().
                    // because of the CoClass attribute on IA
   }
}
```

Compiler Error CS1614

'*attribute*' is ambiguous; use either '@*attribute*' or '*attribute*Attribute'

The compiler has encountered an ambiguous attribute specification.

For convenience, the C# compiler allows you to specify **ExampleAttribute** as just `[Example]`. However, ambiguity arises if an attribute class named **Example** exists along with **ExampleAttribute**, because the compiler cannot tell if `[Example]` refers to the **Example** attribute or the **ExampleAttribute** attribute. To clarify, use `[@Example]` for the **Example** attribute and `[ExampleAttribute]` for **ExampleAttribute**.

The following sample generates CS1614:

```
// CS1614.cs
using System;

// Both of the following classes are valid attributes with valid
// names (MySpecial and MySpecialAttribute). However, because the lookup
// rules for attributes involves auto-appending the 'Attribute' suffix
// to the identifier, these two attributes become ambiguous; that is,
// if you specify MySpecial, the compiler can't tell if you want
// MySpecial or MySpecialAttribute.

public class MySpecial : Attribute {
   public MySpecial() {}
}

public class MySpecialAttribute : Attribute {
   public MySpecialAttribute() {}
}

class MakeAWarning {
   [MySpecial()] // CS1614
                 // Ambiguous: MySpecial or MySpecialAttribute?
   public static void Main() {
   }

   [@MySpecial()] // This isn't ambiguous, it binds to the first attribute above.
   public static void NoWarning() {
   }

   [MySpecialAttribute()] // This isn't ambiguous, it binds to the second attribute above.
   public static void NoWarning2() {
   }

   [@MySpecialAttribute()] // This is also legal.
   public static void NoWarning3() {
   }
}
```

Compiler Error CS1604

Cannot assign to '*variable*' because it is read-only

An assignment was made to a read-only variable.

The following sample generates CS1604:

```
// CS1604.cs
class NoWrite
{
   public static void Main()
   {
      int [] a = {1,2,3};

      foreach(int k in a)
      {
         k = k + 1;   // CS1604, k is read-only
         System.Console.WriteLine(k);
      }
   }
}
```

Compiler Error CS1605

Cannot pass '*var*' as a ref or out argument because it is read-only

A variable passed as a ref or out parameter is expected to be modified in the called method. Therefore, it is not possible to pass a read-only parameter as **ref** or **out**.

The following sample generates CS1605:

```
// CS1605.cs
class NoWrite
{
   public static void Main()
   {
      int [] a = {1,2,3};
      foreach(int k in a)
      {
         RefMethod(ref k);   // CS1605, can't pass k as ref parameter
      }
   }

   static void RefMethod(ref int i)
   {
   }
}
```

Compiler Error CS1606

Assembly signing failed; output may not be signed -- *reason*

The assembly was produced, but when the compiler attempted to finish signing it, there was a failure.

Compiler Warning (level 1) CS1607

Assembly generation -- *reason*

A warning was generated from the assembly-creation phase of the compilation. For more information, see Al.exe Tool Errors and Warnings.

Compiler Error CS1608

The RequiredAttribute attribute is not permitted on C# types

RequiredAttributeAttribute is not allowed on types defined in C# code.

Compiler Error CS1609

Modifiers cannot be placed on property or event accessor declarations

Modifiers can only be placed on property or event declarations and not on the property or event accessor declarations. For more information, see Accessors.

The following sample generates CS1609:

```
// CS1609.cs
class c
{
   public int Prop   //  modifier on property declaration OK
   {
      get
      {
         return 0;
      }
      private set   // CS1609
      // try the following line instead
      // set
      {
      }
   }

   public static void Main()
   {
   }
}
```

Compiler Warning (level 4) CS1610

Unable to delete temporary file '*file*' used for default Win32 resource -- *resource*

When using the /win32res compiler option and when your **%TEMP%** directory does not have DELETE permission, this warning indicates that the compiler could not delete a temporary file that it created.

Make sure that you have read/write/delete permissions for the **%TEMP%** directory.

If necessary, you can manually delete these files and there will be no harm to C# or any of your programs.

Compiler Error CS1611

The params parameter cannot be declared as ref or out

The keywords ref or out cannot be used with the params keyword.

The following sample generates CS1611:

```
// CS1611.cs
public class MyClass
{
   public static void Test(params ref int[] a)   // CS1611, remove ref
   {
   }

   public static void Main()
   {
      Test(1);
   }
}
```

Compiler Error CS1612

Cannot modify the return value of '*expression*' because it is not a variable

An attempt was made to modify a value type that was the result of an intermediate expression. Because the value is not persisted, the value will be unchanged.

To resolve this error, store the result of the expression in an intermediate value, or use a reference type for the intermediate expression.

The following sample generates CS1612:

```
// CS1612.cs
class Reference
{
   public int i;

   public int Prop
   {
      get
      {
         return i;
      }
      set
      {
         i = value;
      }
   }

   public override string ToString()
   {
      return "i = " + i.ToString();
   }
}

struct Value
{
   public int i;

   public int Prop
   {
      get
      {
         return i;
      }
      set
      {
         i = value;
      }
   }

   public override string ToString()
   {
      return "i = " + i.ToString();
   }
}

class Example
{
   private Reference _r;   // Reference type
   private Value _v;       // Value type

   public Example()
   {
      _r = new Reference();
      _v = new Value();
   }
```

```
    public Reference RefProp
    {
       get
       {
          return _r;
       }
       set
       {
          _r = value;
       }
    }

    public Value ValueProp
    {
       get
       {
          return _v;
       }
       set
       {
          _v = value;
       }
    }

    public static void Main()
    {
       Example e = new Example();

       System.Console.WriteLine(e.RefProp);     // Prints 0
       e.RefProp.i = 5;                         // Legal
       System.Console.WriteLine(e.RefProp);     // Prints 5

       e.RefProp.Prop = 6;                      // Legal
       System.Console.WriteLine(e.RefProp);     // Prints 6
       System.Console.WriteLine(e.ValueProp);   // Prints 0

       // Invalid because ValueProp returns a value on the stack,
       // and although you could set that value,
       // it doesn't reflect back into the original property.
       e.ValueProp.i = 7;   // CS1612

       // If the above statment was allowed, this would still print 0.
       System.Console.WriteLine(e.ValueProp);

       // Invalid because ValueProp returns a value on the stack,
       // and although you could set that value,
       // it doesn't reflect back into the original property.
       e.ValueProp.Prop = 8;   // CS1612

       // If the above statement was allowed, this would still print 0.
       System.Console.WriteLine(e.ValueProp);

       Value v = e.ValueProp;
       v.i = 9;
       System.Console.WriteLine(v);             // Prints 9
       System.Console.WriteLine(e.ValueProp);   // Still prints 0
       v.Prop = 10;
       System.Console.WriteLine(v);             // Prints 10
       System.Console.WriteLine(e.ValueProp);   // Still prints 0
       e.ValueProp = v;

       // The following line prints 10 (this is the proper way).
       System.Console.WriteLine(e.ValueProp);
    }
}
```

Compiler Error CS1618

Cannot create delegate with '*method*' because it has a Conditional attribute

You cannot create a delegate with a conditional method because the method might not exist in some builds.

The following sample generates CS1618:

```
// CS1618.cs
using System;
using System.Diagnostics;

delegate void del();

class MakeAnError {
   public static void Main() {
      del d = new del(ConditionalMethod);   // CS1618
      // Invalid because on builds where DEBUG is not set,
      // there will be no "ConditionalMethod".
   }

   [Conditional("DEBUG")]
   public static void ConditionalMethod()
   {
      Console.WriteLine("Do something only in debug");
   }
}
```

Compiler Error CS1619

Cannot create temporary file '*filename*' -- *reason*

The compiler was unable to create a temporary file for the given reason (for example, the disk is full).

Compiler Error CS1900

Warning level must be in the range 0-4

The /warn compiler option can only take one of five possible values (0, 1, 2, 3, or 4). Any other value passed to **/warn** will result in CS1900.

The following sample generates CS1900:

```
// CS1900.cs
// compile with: /W:5
// CS1900 expected
class x
{
   public static void Main()
   {
   }
}
```

Compiler Error CS1901

Conflicting options specified: Warning level 0; Treat warnings as errors

The /warn and /warnaserror compiler options are mutually exclusive; you cannot both turn warnings off and have warnings reported as errors.

The following sample generates CS1901:

```
// CS1901.cs
// compile with: /W:0 /warnaserror
// CS1901 expected
class x
{
   public static void Main()
   {
   }
}
```

Compiler Error CS1902

Invalid option '*option*' for /debug; must be full or pdbonly

An invalid option was passed to the /debug compiler option.

The following sample generates CS1902:

```
// CS1902.cs
// compile with: /debug:x
// CS1902 expected
class x
{
   public static void Main()
   {
   }
}
```

Compiler Error CS1904

'*number*' is not a valid warning number

A number that was passed to the /nowarn compiler option was not a valid warning.

The following sample generates CS1904:

```
// CS1904.cs
// compile with: /nowarn:invalid_warning_number
// CS1904 generated
class X
{
   public static void Main()
   {
   }
}
```

Compiler Error CS2000

Compiler initialization failed unexpectedly

This error indicates various initialization failures. Reinstall Visual Studio.

Compiler Error CS2001

Source file '*file*' could not be found

A source file name was passed to the compiler, but could not be located. Check the spelling of the file name.

Compiler Warning (level 1) CS2002

Source file '*file*' specified multiple times

A source file name was passed to the compiler more than once. You can only specify a file once to the compiler to build an output file.

The following sample generates CS2002:

```
// CS2002.cs
// compile with: CS2002.cs
public class A
{
public static void Main(){}
}
```

To generate the error, compile the example with the command line:

```
csc CS2002.cs CS2002.cs
```

Compiler Error CS2003

Response file '*file*' included multiple times

A response file was passed to the compiler more than once. A response file can only be passed to the compiler once per output file.

For more on response files, see @ (Specify Response File).

Compiler Error CS2005

Missing file specification for '*option*' command-line option

A compiler option was only partially specified.

For example, when using /recurse, you must specify the file to search: **/recurse:***filename***.cs**.

The following sample generates CS2005:

```
// CS2005.cs
// compile with: /recurse:
// CS2005 expected
class x
{
   public static void Main()
   {
   }
}
```

Compiler Error CS2006

Command-line syntax error: Missing '*text*' for '*option*' switch

The syntax for *option* requires additional text. For information, see Compiler Options.

Compiler Error CS2007

Unrecognized command-line option: '*option*'

The compiler was passed a string that was not a compiler option, even though it began with a forward slash (/).

The following sample generates CS2007:

```
// CS2007.cs
// compile with: /recur
// CS2007 expected
class x
{
   public static void Main()
   {
   }
}
```

Compiler Error CS2008

No inputs specified

The compiler was invoked and compiler options were specified, but no source-code files were passed.

Compiler Error CS2011

Unable to open response file '*file*'

A response file was specified in a compilation, but the compiler was unable to locate and open the file.

For more on response files, see @ (Specify Response File).

Compiler Error CS2012

Can't open '*file*' for writing

While using the /bugreport:*file* compiler option, the file could not be opened for writing. Make sure you specified a valid file name and that the file is not read-only.

Compiler Error CS2013

Invalid image base number '*value*'

An invalid value (not a number) was passed to the /baseaddress compiler option.

The following sample generates CS2013:

```
// CS2013.cs
// compile with: /target:library /baseaddress:x
// CS2013 expected
class MyClass
{
}
```

Compiler Warning (level 1) CS2014

Compiler option '*old option*' is obsolete, please use '*new option*' instead

The form of the compiler option is deprecated. See C# Compiler Options for more information.

Compiler Error CS2015

'*file*' is a binary file instead of a source code file

A file was passed to the compiler that was a binary file. The compiler expects a source code file.

Compiler Error CS2016

Code page '*codepage*' is invalid or not installed

The /codepage compiler option was passed an invalid value.

The following sample generates CS2016:

```
// CS2016.cs
// compile with: /codepage:x
// CS2016 expected
class MyClass
{
   public static void Main()
   {
   }
}
```

Compiler Error CS2017

Cannot specify /main if building a module or library

You cannot specify a main entry point when you are building a /target:library.

The following sample generates CS2017:

```
// CS2017.cs
// compile with: /main:MyClass /target:library
// CS2017 expected
class MyClass
{
   public static void Main()
   {
   }
}
```

Compiler Error CS2018

Unable to find messages file 'cscmsgs.dll'

The .dll file that contains the compiler's error and warning messages was not found. This file must be present in the same directory as the other compiler support files.

Compiler Error CS2019

Invalid target type for /target: must specify 'exe', 'winexe', 'library', or 'module'

The /target compiler option was used, but an invalid parameter was passed. To resolve this error, recompile the program using the form of the **/target** option that is appropriate to your output file.

The following sample generates CS2017:

```
// CS2019.cs
// compile with: /target:libra
// CS2019 expected
class MyClass
{
}
```

Compiler Error CS2020

Only the first set of input files can build a target other than 'module'

In a multi-output compilation, the first output file must be built with /target:exe, /target:winexe, or /target:library. Any subsequent output files must be built with /target:module.

Compiler Error CS2021

File name '*file*' is too long or invalid

All file names passed to the C# compiler must be no longer than **_MAX_PATH** (defined in a Windows header file). the compiler will give this error in the following situations:

- A file name (including the path) is longer than **_MAX_PATH**.
- The file name contains invalid characters.
- The file name contains wildcards where wildcards are not allowed (such as in resource file names).

Compiler Error CS2022

Options '/out' and '/target' must appear before source file names

The /out (Set Output Filename) and /target (Specify Output File Format) compiler options, when specified on the command line, must precede the source code files.

Compiler Warning (level 1) CS2023

Ignoring /noconfig option because it was specified in a response file

The /noconfig compiler option was specified in a response file, which is not allowed.

Compiler Error CS2024

Invalid file section alignment number '#'

An invalid value was passed to the /filealign compiler option.

The following sample generates CS2024:

```
// CS2024.cs
// compile with: /filealign:ex
// CS2024 expected
class MyClass
{
   public static void Main()
   {
   }
}
```

Compiler Error CS3001

Argument type '*type*' is not CLS-compliant

A public, protected, or **protected internal** method must accept a parameter whose type is compliant with the Common Language Specification (CLS).

The following sample generates CS3001:

```
// CS3001.cs
[assembly:System.CLSCompliant(true)]

public class a
{
   public void bad(ushort i)   // CS3001
   {
   }

   private void OK(ushort i)   // OK, private method
   {
   }

   public static void Main()
   {
   }
}
```

See Also

What Is the Common Language Specification

Compiler Error CS3002

Return type of '*method*' is not CLS-compliant

A public, protected, or **protected internal** method must return a value whose type is compliant with the Common Language Specification (CLS).

The following sample generates CS3002:

```
// CS3002.cs
[assembly:System.CLSCompliant(true)]

public class a
{
   public ushort bad()   // CS3002, public method
   {
      ushort a;
      a = ushort.MaxValue;
      return a;
   }

   private ushort OK()   // OK, private method
   {
      ushort a;
      a = ushort.MaxValue;
      return a;
   }

   public static void Main()
   {
   }
}
```

See Also

What Is the Common Language Specification

Compiler Error CS3003

Type of '*variable*' is not CLS-compliant

A public, protected, or **protected internal** variable must be of a type that is compliant with the Common Language Specification (CLS).

The following sample generates CS3003:

```
// CS3003.cs
[assembly:System.CLSCompliant(true)]

public class a
{
   public ushort a1;   // CS3003, public variable
   public static void Main()
   {
   }
}
```

See Also

What Is the Common Language Specification

Compiler Error CS3004

Mixed and decomposed Unicode characters are not CLS-compliant

Only composed UNICODE characters are allowed in public, protected, or **protected internal** identifiers in order to be compliant with the Common Language Specification (CLS).

See Also

What Is the Common Language Specification

Compiler Error CS3005

Identifier '*identifier*' differing only in case is not CLS-compliant

A public, protected, or **protected internal** identifier, which differs from another **public**, **protected**, or **protected internal** identifier only in the case of one or more letters, is not compliant with the Common Language Specification (CLS).

The following sample generates CS3003:

```
// CS3005.cs
using System;
[assembly:CLSCompliant(true)]

public class a
{
   public static int a1 = 0;
   public static int A1 = 1;   // CS3005

   public static void Main()
   {
      Console.WriteLine(a1);
      Console.WriteLine(A1);
   }
}
```

See Also

What Is the Common Language Specification

Compiler Error CS3006

Overloaded method '*method*' differing only in ref or out is not CLS-compliant

A method does not cannot be overloaded based on the ref or out parameter and still comply with the Common Language Specification (CLS).

The following sample generates CS3006:

```
// CS3006.cs
using System;
[assembly: CLSCompliant(true)]

public class MyClass
{
   public void f(int i)
   {
   }

   public void f(ref int i)   // CS3006
   {
   }

   public static void Main()
   {
   }
}
```

To resolve this error, comment out the assembly-level attribute or remove one of the method definitions.

See Also

What Is the Common Language Specification

Compiler Error CS3008

Identifier '*identifier*' is not CLS-compliant

A public, protected, or **protected internal** identifier breaks compliance with the Common Language Specification (CLS) if it begins with an underscore character (_).

The following sample generates CS3008:

```
// CS3008.cs
using System;
[assembly:CLSCompliant(true)]

public class a
{
   public static int _a = 0;  // CS3008
   // OK, private
   // private static int _a1 = 0;

   public static void Main()
   {
   }
}
```

See Also

What Is the Common Language Specification

Compiler Error CS3009

'*type*': base type '*type*' is not CLS-compliant

A base type was marked as not having to be compliant with the Common Language Specification (CLS) in a module that was marked as being CLS compliant. Either remove the attribute that specifies the module is CLS compliant or remove the attribute that indicates the type is not CLS compliant.

The following sample generates CS3009:

```
// CS3009.cs
using System;
[assembly:CLSCompliant(true)]

[CLSCompliant(false)]
public class B
{
}

public class C : B   // CS3009
{
   public static void Main () {}
}
```

See Also

What Is the Common Language Specification

Compiler Error CS3010

'*member*': CLS-compliant interfaces must have CLS-compliant members

In a module marked with `[assembly:CLCSompliant(true)]`, an interface contains a member marked with `[CLCSompliant(false)]`. Remove one of the Common Language Specification (CLS) compliance attributes.

The following sample generates CS3010:

```
// CS3010.cs
using System;
[assembly:CLSCompliant(true)]

public interface I
{
   [CLSCompliant(false)]
   int mf();   // CS3010
}

public class C : I
{
   public int mf()
   {
      return 1;
   }

   public static void Main()
   {
   }
}
```

See Also

What Is the Common Language Specification

Compiler Error CS3011

'*member*': only CLS-compliant members can be abstract

A class member cannot be both abstract and non-compliant with the Common Language Specification (CLS). The CLS specifies that all class members shall be implemented.

The following sample generates CS3011:

```
// CS3011.cs
using System;
[assembly:CLSCompliant(true)]

public abstract class I
{
   [CLSCompliant(false)]
   public abstract int mf();   // CS3011

   // OK
   [CLSCompliant(false)]
   public void mf2()
   {
   }
}

public class C : I
{
   public override int mf()
   {
      return 1;
   }

   public static void Main()
   {
   }
}
```

See Also

What Is the Common Language Specification

Compiler Warning (level 1) CS3012

You must specify the CLSCompliant attribute on the assembly

In order for a module to be compliant with the Common Language Specification (CLS) through `[module:System.CLCSompliant(true)]`, it must be built with the /target:module compiler option.

The following sample, when built without **/target:module**, generates CS3012:

```
// CS3012.cs
// compile with: /W:1
[module:System.CLSCompliant(true)]   // CS3012

public class C
{
   public static void Main()
   {
   }
}
```

See Also

What Is the Common Language Specification

Compiler Error CS3013

Added module is not CLS-compliant, or is missing CLSCompliant attribute

A module that was compiled with the /target:module compiler option was added to a compilation with /addmodule. However, the module's compliance with the Common Language Specification (CLS) does not agree with the CLS state of the current compilation.

CLS compliance is indicated with the module attribute. For example, `[module:CLSCompliant(true)]` indicates that the module is CLS compliant, and `[module:CLSCompliant(false)]` indicates that the module is not CLS compliant. The default is `[module:CLSCompliant(false)]`.

See Also

What Is the Common Language Specification

Compiler Error CS3014

'*member*' cannot be marked as CLS compliant because the assembly is not marked as compliant

In a source code file where compliance with the Common Language Specification (CLS) was declined, a construct in the file was marked as being CLS compliant. This is not allowed. Remove one of the **CLSCompliant** specifications.

The following sample generates CS3014:

```
// CS3014.cs
using System;
[assembly:CLSCompliant(false)]

public class I
{
   [CLSCompliant(true)]   // CS3014
   public void mf()
   {
   }

   public static void Main()
   {
   }
}
```

See Also

What Is the Common Language Specification

Compiler Error CS3015

'*method signature*' has no accessible constructors which use only CLS compliant types

To be compliant with the Common Language Specification (CLS), the argument list of an attribute class cannot contain an array.

The following sample generates C3015:

```
// CS3015.cs
using System;
[assembly:CLSCompliant(true)]

public class MyAttribute : Attribute
{
   public MyAttribute(int[] ai)   // CS3015
   // try the following line instead
   // public MyAttribute(int ai)
   {
   }
}

public class MyClass
{
   public static void Main()
   {
   }
}
```

See Also

What Is the Common Language Specification

Compiler Error CS3016

Arrays as attribute arguments is not CLS-compliant

It is not compliant with the Common Language Specification (CLS) to pass an array to an attribute.

The following sample generates CS3016:

```
// CS3016.cs
using System;
[assembly : CLSCompliant(true)]

[C(new int[] {1, 2})]   // CS3016
// try the following line instead
// [C()]
class C : Attribute
{
   public C()
   {
   }

   public C(int[] a)
   {
   }

   public static void Main ()
   {
   }
}
```

See Also

What Is the Common Language Specification

Compiler Warning (level 1) CS5000

Unknown compiler option '*/option*'

An invalid compiler option was specified.

Compiler Error CS5001

Program '*output file*' does not have an entry point defined

No Main method was found in the code that produces an executable file.

The following sample generates CS5001:

```
// CS5001.cs
// CS5001 expected
public class a
{
   // uncomment the following to resolve this error
   // public static void Main()
   // {
   // }
}
```

Visual C# Code Wizards

You can create a range of Visual C# projects in Visual Studio. For a list and description of the types of projects you can create, see Visual Basic and Visual C# Projects.

Templates make it easy to build robust projects. For details, see Using Visual C# Add New Item Templates.

Once you create a project, you can change it or add items to it from one of two views:

- From Solution Explorer, you can add items, forms, controls, components, or classes to your project. When you modify your project from Solution Explorer, generic code is automatically inserted into your project.
- From Class View, you can use one or more of the following wizards to add C# programming elements to your source code:
    - C# Class Wizard
    - C# Method Wizard
    - C# Property Wizard
    - C# Field Wizard
    - C# Indexer Wizard
    - C# Interface Method Wizard
    - C# Interface Property Wizard
    - C# Interface Indexer Wizard

See Also

Visual C# Language

Using Visual C# Add New Item Templates

You can easily add items that are common to Visual C# projects using the **Add New Item** command. When you use the **Add New Item** command, the Add New Item dialog box appears with a list of item templates, which add the appropriate files to your project.

> **Note** The list of item templates that appear in the Add New Item dialog box vary, depending on the type of project you are working with.

The following table shows an alphabetical list of Visual C# Add New Item templates.

Templates	Description
Add User Control	Adds a class for creating a Windows Form control using the visual designer.
Application Configuration File	Adds a .config file that you can use to configure your application settings.
Assembly Resource File	Adds a .NET resource file.
Bitmap File	Adds a Win32 bitmap file.
Class	Adds an empty class declaration.
Code File	Adds a blank .cs code file.
Component Class	Adds a class for creating components using the visual designer.
Crystal Report	Adds a Crystal Report file that publishes data to a Windows Form or Web Form . This template is not a Standard Edition feature.
Cursor File	Adds a Win32 cursor file.
Custom Control	Adds a class for creating a user-drawn Windows Form control.
Data Form Wizard	Adds a .cs file for creating an XML schema with DataSet classes.
Data Set	Adds an .xsd file for creating an XML schema with DataSet classes.
Frame Set	Adds an HTML file that hosts multiple HTML pages.
Global Applications Class	Adds a class for handling Web Application events.
HTML Page	Adds an HTML page that can include client-side code.
Icon File	Adds a Win32 icon file.
Inherited Form	Adds a new form based on an existing Windows Form.
Inherited User Control	Adds a new control based on an existing Windows Form control.
Installer Class	Adds a class to be invoked at set-up time.
JScript File	Adds a script file containing JScript code.
Mobile Web Form	Adds a Web form for use in applications that target mobile devices.
Mobile Web User Control	Adds a Web user control for use in applications that target mobile devices.
Static Discovery File	Adds a file used to publish information about a Web Service.
Style Sheet	Adds a cascading style sheet used for rich HTML style definitions.
Text File	Adds a blank text file.
User Control	Adds a class for creating a Windows Form control using the visual designer.
VBScript File	Adds a script file that contains VBScript code.
Web Configuration File	Adds a .config file you can use to configure your Web application settings.
Web Custom Control	Adds a class for creating an ASP.NET server control.
Web Form	Adds a form for Web applications.
Web Service	Adds a class for creating Web Services to use from other applications.
Web User Control	Adds an ASP.NET server control created using the visual designer.
Windows Form	Adds a form for Windows applications to your project.
Windows Service	Adds a class for creating Windows services.
Windows Script Host	Adds a .WSF file that contains script that is run as a Windows program.
XML File	Adds a blank .xml file.
XML Schema	Adds a file for creating a schema for XML documents.
XSLT File	Adds a file used to transform XML documents.

See Also

Adding a C# Class

You can add a class to your project either from Solution Explorer or from Class View.

Adding a C# Class Using Solution Explorer

You can build a generic class that is automatically declared under the current default namespace. For example, classes added to a project named WindowsApplication1 are added to the namespace WindowsApplication1.

To add a class in Solution Explorer

1. In Solution Explorer, right-click the project name and click **Add**, and then click **Add Class**.

   The **Add New Item** dialog box appears with the **C# Class** icon already selected.

2. In the dialog box, enter a class name in the **Name** field and click **Open**.

   The new class is added to your project.

For more control over user-defined elements when generating classes, you can either type the code or add the class using Class View.

Adding a Class Using Class View

When you add a class from Class View, you have maximum control defining elements while using C# Class Wizard.

To add a class in Class View

- In Class View, right-click the project name and click **Add**, and then click **Add Class**.

  The C# Class Wizard appears.

See Also

Visual C# Code Wizards | Visual C# Samples

C# Class Wizard

The C# Class Wizard appears when you add a C# class from Class View. For details, see Adding a C# Class The wizard has three pages that help you define class elements. You can access each page by clicking the page titles listed in the column on the left side of the wizard.

Page tab	Description
Class Options	Opens the default start page where you define general class elements.
Base Class	Opens the **Base Class** page where you can specify the class from which your class de rives.
Inheritance	Opens the **Inheritance** page where you can specify which inherited interface imple mentations your class uses.

Visual C# Code Wizards | Visual C# Samples

Class Options Tab, C# Class Wizard

The **Class Options** tab serves as the first page of the Class Wizard.

Class name

You must provide a class name. If you provide only a class name, the wizard creates a simple public class using the provided name, with no base class, no inheritance information, and no interfaces.

File name

The wizard automatically generates a new .cs file for each new class.

A separate .cs file per C# class is a natural unit in programming. The unit makes it easy to locate and access your class for editing. However, if you want to add your new class to an existing .cs file, then you can use the ellipsis button (...) to specify the .cs file in which the wizard will insert your new class.

Namespace

Select a namespace from the **Namespace** list, or type a name of your choice.

A top-level namespace is declared. For more information, see 9.2 Namespace declarations.

If you do not declare a namespace, the wizard creates your class under the project namespace. In Class View, you can see that your class will remain as part of that namespace until you provide, in the Source Editor, a namespace for that class.

Access

Define the accessibility level of your class by selecting from the **Access** list. For more information, see Accessibility Levels.

Option	Description
public	Specifies class declaration with no access restrictions.
internal	Specifies class declaration accessibility from files within the same assembly.

Class modifiers

Declare your class with type modifiers, particular to class inheritance, by selecting one of the **Class Modifier** options. The option **None** is set by default.

Option	Description
None	No additional modifiers are included in your class declaration.
Abstract	The class can only be used as the base class of other classes.
Sealed	Declares your class cannot be inherited.

Comment

The wizard converts comments you provide into XML documentation and encapsulates them in <summary> tags, which are used to provide short descriptions. You can find the comments in the .cs file (specified in **File Name**), placed right before the class declaration.

See Also

C# Class Wizard | Visual C# Code Wizards | Visual C# Samples

Base Class Tab, C# Class Wizard

The **Base Class** tab displays the Base Class page, where you can specify the class from which your class derives. For details, see 10.2.1 Inheritance.

Namespace

Select a namespace from the list.

Base class

Once you have specified a namespace, select a base class from the list.

C# does not allow derived classes with inheritance from multiple base classes. For details, see 10.1.2 Class base specification.

See Also

C# Class Wizard | Visual C# Code Wizards | Visual C# Samples

Inheritance Tab, C# Class Wizard

The **Inheritance** tab opens the Inheritance page, where you can specify which inherited interface implementations your class uses. For details, see 13.4.3 Interface implementation inheritance.

Current namespace

Select a namespace from the list.

Interfaces

Once you specify a namespace, a list of interfaces appears in the **Available interfaces** list. Choose the interface that will implement inheritance by highlighting the name of the interface and clicking the appropriate buttons.

Option	Description
Add	Adds the interface, selected in the **Available interfaces** list, to the **Selected interfaces** list .
Add all	Adds all interfaces in the **Available interfaces** list to the **Selected interfaces** list.
Remove	Removes the selected interface from the **Selected interfaces** list.
Remove all	Removes all interfaces from the **Selected interfaces** list.

Selected interfaces

Provides all implementations specified to be added to your new derived class.

See Also

C# Class Wizard | Visual C# Code Wizards | Visual C# Samples

Adding a Method to a C# Class

You can add a method to a C# class using a code wizard in Class View only.

To add a method to a C# class

1. In Class View, expand the project node to display the classes in the project.
2. Right-click the class to which you want to add a method.
3. On the shortcut menu, click **Add** and then click **Add Method** to display the C# Method Wizard.
4. Provide the method information in the appropriate wizard boxes. See C# Method Wizard for details.

To add a parameter to a method in a C# class

1. If the method has parameters, provide the modifier and the type, and then supply the **Parameter name**.
2. Click **Add** to add the parameter name to **Parameter list**. The parameter is added to the **Method signature** box at the bottom of the wizard.
3. Review the code in the **Method signature** line. If you need to remove a parameter, highlight it in the **Parameter list** and click **Remove**.
4. Click **Finish** to generate the method code in your project.

See Also

Visual C# Code Wizards | C# Class Wizard

C# Method Wizard

Add a method and define its elements using the one-page C# Method Wizard.

Method access

Sets the access to the method. Access modifiers are keywords for setting access to the method. Specify the access other classes have to the method. The method access level is set to **public** by default. The following access levels are available:

- public
- protected
- private
- internal
- **protected internal**

See Accessibility Levels for an overview of these five levels and for more information about specifying access for methods.

Return type

Sets the value type that the method returns in the return statement. The return type is set to **void** by default. For more information about C# return types, see Value Types and Built-in Types Table.

The following value types are available:

- bool
- byte
- char
- decimal
- double
- float
- int
- long
- object
- sbyte
- short
- string
- uint
- ulong
- ushort
- **void**

Method name

Sets the name of the method to add to the class. You must add at least the method name.

Modifier

Sets the parameter's modifier. **Modifier** is set to **None** by default. See Method Parameters for more information.

- **none**
- ref
- out

Parameter type

Sets the type for the **Parameter name**. The parameter type is set to **int** by default. The type is reflected in the **Method signature** box at the bottom of the wizard. The same values available for the return type are available for the parameter type, with the exception of **void**.

Parameter name

Sets the name of a parameter to pass through your method. After typing the name, you must click **Add** to add it to the list of parameters that will pass through your method.

If you do not provide a parameter name, the wizard ignores any **Modifier** or **Parameter type** selections.

Once you click **Add**, the parameter name appears in **Parameter list** and in **Method signature** at the bottom of the wizard.

If you supply a parameter name and then click **Finish** before you click **Add**, the parameter is not added to the method. You

must find the method and insert the parameter manually.

Add

Adds the parameter you specify in **Parameter name**, and its modifier and type, to the **Parameter list** and displays the complete parameter in the **Method signature** box at the bottom of the wizard. You must click **Add** to add a parameter to the list.

Remove

Removes the parameter you select in **Parameter list** from the list and from the **Method signature** box at the bottom of the wizard.

Parameter list

Displays all parameters and their modifiers and types currently added for the method. As you add parameters, the wizard updates the **Parameter list** to display each parameter, with its modifier and type.

Method modifiers

Modifies the declarations of types and type members. By default, no modifiers are included. Using this wizard, you can add the following modifiers.

Modifier option	Description
Static	The method belongs to the type itself rather than to a specific object.
Abstract	The method is a member of an abstract class. If the class to which you are adding the method is n ot abstract, this option is not available. See C# Class Wizard for more information.
Virtual	The method's implementation can be changed by an overriding member in a derived class.
Extern	Advanced. The method is implemented externally.
Override	Advanced. Provide a new implementation of a virtual member inherited from a base class.
New	Advanced. Explicitly hides a member inherited from a base class. To hide an inherited member, de clare it in the derived class using the same name, and modify it with the new modifier.

You can combine certain modifiers; however, some modifiers are mutually exclusive. For example, you can set your modifiers to **new virtual**, **extern static** or **override abstract**.

The following combinations of options are not allowed:

Modifier	Cannot combine with ...
static	**virtual**, **abstract**, and **override**
virtual	**static**, **abstract**, and **override**
override	**new**, **static**, and **virtual**
abstract	**virtual** and **static**
new	**override**
extern	**abstract**

Depending on your modifier selection, check boxes appear dimmed for excluded combinations.

Comment

Provides a short description to the method. Comments are converted into XML documentation and encapsulated by <summary> tags, which are used to provide the descriptions.

Method signature

Displays the method as it is added to your code when you click **Finish**. You cannot edit the text in this box. To change the method, change the appropriate fields in the wizard.

See Also

Types Reference Tables | Visual C# Code Wizards | Adding a Method to C# Class

Adding a Property to a C# Class

You can add a property to a C# class using a code wizard in Class View only. Adding a property using the C# Property Wizard adds a property stub body to the class's .cs file, which you can then edit.

To add a property to a C# class

1. In Class View, right-click the class name.

   If necessary, expand the nodes that expose the class where you want to add the property.

2. On the shortcut menu, click **Add**.
3. On the submenu, click **Add Property**, and the C# Property Wizard appears. For details, see C# Property Wizard.

See Also

Visual C# Code Wizards | Properties | Visual C# Samples

C# Property Wizard

Properties are named members of classes, structs, and interfaces. They provide a flexible mechanism to read, write, or compute the values of private fields through accessors.

Add a property and define its elements using the one-page C# Property Wizard.

Property access

Specifies the access other classes have to the property. Access modifiers are keywords for setting access to the property. The property access level is set to **public** by default. The following access levels are available:

- public
- protected
- private
- internal
- **protected internal**

See Accessibility Levels for an overview of these five levels and for more information about specifying access.

Property type

Sets the value type for the property. The property type is set to **int** by default. For more information about C# property types, see Value Types and Built-in Types Table.

The following value types are available:

- bool
- byte
- char
- decimal
- double
- float
- int
- long
- object
- sbyte
- short
- string
- uint
- ulong
- ushort

Property name

Sets the name of the property to add to the class.

Accessors

Sets the property accessor to **get**, **set**, or to both. The option for **Accessors** is set to **get/set** by default. See Accessors for more information about the accessors and their executable statements.

Accessor options	Description
get	Reads or computes the property and returns a value of the property and its type. If you choose this option, your property is read-only.
set	When you assign a value to the property, you invoke the set accessor with an argument that provides the new value. If you choose this option, your property is write-only. You cannot reference a write-only property except as a target of an assignment.
get/set	When you choose this option, your property contains both a get and a set accessor. It is a read-write property.

Property modifiers

Modifies the declarations of types and type members. Using this wizard, you can add one of the modifiers listed below. The static, virtual, and abstract modifiers are mutually exclusive.

Modifier options	Description
None	No additional modifiers are included in the property declaration.

Static	The property belongs to the type itself rather than to a specific object.
Virtual	The property's implementation can be changed by an overriding property in a derived class.
Abstract	The property is a member of an abstract class. If the class to which you are adding the property is not abstract, this option is not available. See C# Class Wizard for more information.

Comment

Comments are converted into XML documentation and encapsulated by <summary> tags, which are used to provide short descriptions.

See Also

Adding a Property to a C# Class | Visual C# Code Wizards | Properties | Visual C# Samples

Adding a Field to a C# Class

You can add a field to a C# class using a code wizard in Class View only.

To add a field to a C# class

1. In Class View, right-click the class name.

   If necessary, expand the nodes that expose the class where you want to add the field.

2. From the shortcut menu, click **Add**.
3. From the submenu, click **Add Field**. The C# Field Wizard appears. For details, see C# Field Wizard.

See Also

Visual C# Code Wizards | Visual C# Samples

C# Field Wizard

Add a field and define its elements using the one-page C# Field Wizard.

Field access

Access modifiers are keywords for setting access to the field. Specify the access other classes have to the field. The field access level is set to **public** by default. The following access levels are available:

- public
- protected
- private
- internal
- **protected internal**

See Accessibility Levels for an overview of these five levels and for more information about specifying access.

Field type

Either provide a value type or select a value type from the list. The field type is set to **int** by default. For more information about C# field types, see Value Types and Built-in Types Table.

The following value types are available:

- bool
- byte
- char
- decimal
- double
- float
- int
- long
- object
- sbyte
- short
- string
- uint
- ulong
- ushort

Field name

Provide the name of the field to add to the class.

Field modifiers

Modify the declarations of types and type members. Using this wizard, you can add one of the following modifiers. These modifiers are mutually exclusive.

Modifier option	Description
None	No additional modifiers are included in the field declaration.
Static	The field belongs to the type itself rather than to a specific object.
Constant	The field has a constant value and is declared as a const. It requires an initial value.

Field value

If you select **Constant** as a field modifier, you must assign a value to the field. If you do not select **Constant** as a field modifier, this box appears dimmed, and the field has no initial value.

Comment

Comments are converted into XML documentation and encapsulated by <summary> tags, which are used to provide short descriptions.

See Also

Adding a Field to a C# Class | Visual C# Code Wizards | Visual C# Samples

Adding an Indexer to a C# Class

You can add an indexer to a C# class using a code wizard in Class View only.

To add an indexer to a C# class

1. In Class View, right-click the class name.

   If necessary, expand the nodes to expose the class where you want to add the indexer.

2. On the shortcut menu, click **Add**.
3. On the submenu, click **Add Indexer**. The C# Indexer Wizard appears. For details, see C# Indexer Wizard.

See Also

Visual C# Code Wizards | Indexers | Visual C# Samples

C# Indexer Wizard

Add an indexer and define its elements using the one-page C# Indexer Wizard.

Indexer access

Access modifiers are keywords for setting access to the indexer. Specify the access other classes have to the indexer. The indexer access level is set to **public** by default.

The following access levels are available:

- public
- protected
- private
- internal
- **protected internal**

See Accessibility Levels for an overview of these five levels and for more information about specifying access.

Indexer type

From the list, select the value type for the indexer. The indexer type is set to **int** by default. For more information about C# indexer types, see Value Types and Built-in Types Table. The following value types are available:

- bool
- byte
- char
- decimal
- double
- float
- int
- long
- object
- sbyte
- short
- string
- uint
- ulong
- ushort

Parameter type

From the list, select the type for the **Parameter name**. The parameter type is set to **int** by default. The same values available for the indexer type are available for the parameter type.

Parameter name

Sets the name of a parameter for the interface indexer. After typing the name, you must click **Add** to add it to the list of parameters.

You must provide at least one parameter name for the interface indexer.

The parameters **out** and **ref** are not allowed.

Once you click **Add**, the parameter name is displayed in **Parameter list**.

If you supply a parameter name and then click **Finish** before you click **Add**, the parameter is not added to the indexer. You must find the interface indexer in your code and insert the parameter manually.

Add

Adds the parameter you specify in **Parameter name**, and its type, to the **Parameter list**. You must click **Add** to add a parameter to the list.

Remove

Removes the parameter you select in **Parameter list** from the list.

Parameter list

Displays all parameters and their types currently added for the indexer. As you add parameters, the wizard updates the **Parameter list** to display each parameter, with its type.

Indexer modifiers

Modify the declarations of types and type members. Using this wizard, you can add one of the following modifiers. These modifiers are mutually exclusive.

Modifier option	Description
None	No additional modifiers are included in the indexer declaration.
Virtual	The indexer's implementation can be changed by an overriding indexer in a derived class.
Abstract	The indexer is a member of an abstract class. If the class to which you are adding the indexer is n ot abstract, this option is not available. See C# Class Wizard for more information.

Comment

Comments are converted into XML documentation and encapsulated by <summary> tags, which are used to provide short descriptions.

See Also

Adding an Indexer to a C# Class | Visual C# Code Wizards | Indexers | Visual C# Samples

Adding a Method to a C# Interface

In order to add a method to an interface from Class View, you must have already declared the interface in the Text Editor, so that it appears in the Class View window. For more information, see 13.1 Interface declarations.

You can use the C# Interface Method Wizard, which is only accessible in Class View, to add a new method to your interface.

To add a method to a C# interface

1. In Class View, select the interface.

   If necessary, expand the nodes that expose the interface where you want to add the method.

2. In Class View, right-click the interface name, and then select the **Add** menu item.

   -or-

   Open the **Project** menu.

3. Select the **Add Method** command to display the C# Interface Method Wizard.

See Also

Visual C# Code Wizards | Visual C# Samples

C# Interface Method Wizard

Add a method to an interface and define its elements using the one-page C# Interface Method Wizard.

Return type

Sets the value type that the interface method returns in the return statement. The return type is set to **void** by default. For more information about C# return types, see Value Types and Built-in Types Table.

The following value types are available:

- bool
- byte
- char
- decimal
- double
- float
- int
- long
- object
- sbyte
- short
- string
- uint
- ulong
- ushort
- **void**

Method name

Sets the name of the interface method to add to the class. You must add at least the method name.

Modifier

Sets the parameter's modifier. **Modifier** is set to **None** by default. See Method Parameters for more information.

- **None**
- ref
- out

Parameter type

Sets the type for the **Parameter name**. The parameter type is set to **int** by default. The type is reflected in the **Method signature** box at the bottom of the wizard. The same values available for the return type are available for the parameter type, with the exception of **void**.

Parameter name

Sets the name of a parameter to pass through your interface method. After typing the name, you must click **Add** to add it to the list of parameters that will pass through your interface method.

If you do not provide a parameter name, the wizard ignores any **Parameter type** selections.

Once you click **Add**, the parameter name appears in **Parameter list** and in **Method signature** at the bottom of the wizard.

If you supply a parameter name and then click **Finish** before you click **Add**, the parameter is not added to the interface method. You must find the interface method and insert the parameter manually.

Add

Adds the parameter you specify in **Parameter name**, its modifier, and its type to the **Parameter list** and displays the complete parameter in the **Method signature** box at the bottom of the wizard. You must click **Add** to add a parameter to the list.

Remove

Removes the parameter you select in **Parameter list** from the list and from the **Method signature** box at the bottom of the wizard.

Parameter list

Displays all parameters, their modifiers, and their types currently added for the method. As you add parameters, the wizard updates the **Parameter list** to display each parameter, with its modifier and type.

Comment

Provides a short description to the interface method. Comments are converted into XML documentation and encapsulated by <summary> tags, which are used to provide the descriptions.

Method signature

Displays the interface method as it is added to your code when you click **Finish**. You cannot edit the text in this box. To change the interface method, change the appropriate fields in the wizard.

See Also

Adding a Method to a C# Interface | Visual C# Code Wizards | Visual C# Samples

Adding a Property to a C# Interface

In order to add a property to an interface from Class View, you must have already declared the interface in the Text Editor, so that it appears in the Class View window. For more information, see 13.1 Interface declarations.

You can use the C# Interface Property Wizard, which is only accessible in Class View, to add a new property to your interface.

To add a property to a C# interface

1. In Class View, select the interface.

   If necessary, expand the nodes that expose the interface where you want to add the property.

2. In Class View, right-click the interface name, and then select the **Add** menu item.

   -or-

   Open the **Project** menu.

3. Select the **Add Property** command to display the C# Interface Property Wizard.

See Also

Visual C# Code Wizards | Visual C# Samples

C# Interface Property Wizard

Properties are named members of classes, structs, and interfaces. They provide a flexible mechanism to read, write, or compute the values of private fields through the **get** and **set** modifiers.

Add an interface property and define its elements using the one-page C# Interface Property Wizard.

Property type

Sets the value type for the property. The property type is set to **int** by default. For more information about C# property types, see Value Types and Built-in Types Table.

The following value types are available:

- bool
- byte
- char
- decimal
- double
- float
- int
- long
- object
- sbyte
- short
- string
- uint
- ulong
- ushort

Property name

Sets the name of the property to add to the interface.

Accessors

Sets the property to **get**, **set**, or **get/set**. The option for **Accessors** is set to **get/set** by default.

Modifier options	Description
get	Reads or computes the property and returns a value of the property and its type. If you choose this option, your property is read-only.
set	When you assign a value to the property, you invoke the **set** modifier with an argument that provides the new value. If you choose this option, your property is write-only. You cannot reference a write-only property except as a target of an assignment.
get/set	The property contains both a **get** and a **set** modifier. It is a read-write property.

Comment

Inserts a comment into the project immediately above the interface property.

See Also

Adding a Property to a C# Interface | Visual C# Code Wizards | Visual C# Samples

Adding an Indexer to a C# Interface

In order to add an indexer to an interface from Class View, you must have already declared the interface in the Text Editor, so that it appears in the Class View window. For more information, see 13.1 Interface declarations.

You can use the C# Interface Indexer Wizard, which is only accessible in Class View, to add a new indexer to your interface.

To add an indexer to a C# interface

1. In Class View, select the interface.

   If necessary, expand the nodes that expose the interface where you want to add the indexer.

2. In Class View, right-click the interface name, and then select the **Add** menu item.

   -or-

   Open the **Project** menu.

3. Select the **Add Indexer** command to display the C# Interface Indexer Wizard.

See Also

Visual C# Code Wizards | Indexers | Visual C# Samples

C# Interface Indexer Wizard

Add an indexer to an interface and define its elements using the one-page C# Interface Indexer Wizard.

Indexer type

Sets the value type for the indexer. The indexer type is set to **int** by default. For more information about C# indexer types, see Value Types and Built-in Types Table.

The following value types are available:

- bool
- byte
- char
- decimal
- double
- float
- int
- long
- object
- sbyte
- short
- string
- uint
- ulong
- ushort

Parameter type

From the list, select the type for the **Parameter name**. The parameter type is set to **int** by default. The same values available for the indexer type are available for the parameter type.

Parameter name

Sets the name of a parameter for the interface indexer. After typing the name, you must click **Add** to add it to the list of parameters.

You must provide at least one parameter name for the interface indexer.

The parameters **out** and **ref** are not allowed.

Once you click **Add**, the parameter name is displayed in **Parameter list**.

If you supply a parameter name and then click **Finish** before you click **Add**, the parameter is not added to the indexer. You must find the interface indexer in your code and insert the parameter manually.

Add

Adds the parameter that you specify in **Parameter name** and its type to the **Parameter list**. You must click **Add** to add a parameter to the list.

Remove

Removes the parameter you select in **Parameter list** from the list.

Parameter list

Displays all parameters and their types currently added for the interface indexer. As you add parameters, the wizard updates the **Parameter list** to display each parameter with its type.

Comment

Comments are converted into XML documentation and encapsulated by <summary> tags, which are used to provide short descriptions.

See Also

Adding an Indexer to a C# Interface | Indexers | Visual C# Code Wizards

C# Language Specification

Abstract

This document describes the syntax, semantics, and design of the C# programming language.

Authors

Scott Wiltamuth and Anders Hejlsberg

Navigation

To navigate within this document, use the help Contents window (CTRL+ALT+F1).

1. Introduction

C# (pronounced "C sharp") is a simple, modern, object-oriented, and type-safe programming language. It will immediately be familiar to C and C++ programmers. C# combines the high productivity of Rapid Application Development (RAD) languages and the raw power of C++.

Visual C# .NET is Microsoft's C# development tool. It includes an interactive development environment, visual designers for building Windows and Web applications, a compiler, and a debugger. Visual C# .NET is part of a suite of products, called Visual Studio .NET, that also includes Visual Basic .NET, Visual C++ .NET, and the JScript scripting language. All of these languages provide access to the Microsoft .NET Framework, which includes a common execution engine and a rich class library. The .NET Framework defines a "Common Language Specification" (CLS), a sort of lingua franca that ensures seamless interoperability between CLS-compliant languages and class libraries. For C# developers, this means that even though C# is a new language, it has complete access to the same rich class libraries that are used by seasoned tools such as Visual Basic .NET and Visual C++ .NET. C# itself does not include a class library.

The rest of this chapter describes the essential features of the language. While later chapters describe rules and exceptions in a detail-oriented and sometimes mathematical manner, this chapter strives for clarity and brevity at the expense of completeness. The intent is to provide the reader with an introduction to the language that will facilitate the writing of early programs and the reading of later chapters.

1.1 Getting started

The canonical "hello, world" program can be written as follows:

```
using System;
class Hello
{
   static void Main() {
      Console.WriteLine("hello, world");
   }
}
```

The source code for a C# program is typically stored in one or more text files with a file extension of `.cs`, as in `hello.cs`. Using the command-line compiler provided with Visual Studio .NET, such a program can be compiled with the command-line directive

```
csc hello.cs
```

which produces an application named `hello.exe`. The output produced by this application when it is run is:

```
hello, world
```

Close examination of this program is illuminating:

- The `using System;` directive references a namespace called `System` that is provided by the Microsoft .NET Framework class library. This namespace contains the `Console` class referred to in the `Main` method. Namespaces provide a hierarchical means of organizing the elements of one or more programs. A "using" directive enables unqualified use of the types that are members of the namespace. The "hello, world" program uses `Console.WriteLine` as shorthand for `System.Console.WriteLine`.
- The `Main` method is a member of the class `Hello`. It has the `static` modifier, and so it is a method on the class `Hello` rather than on instances of this class.
- The entry point for an application — the method that is called to begin execution — is always a static method named `Main`.
- The "hello, world" output is produced using a class library. The language does not itself provide a class library. Instead, it uses a class library that is also used by Visual Basic .NET and Visual C++ .NET.

For C and C++ developers, it is interesting to note a few things that do *not* appear in the "hello, world" program.

- The program does not use a global method for `Main`. Methods and variables are not supported at the global level; such elements are always contained within type declarations (e.g., class and struct declarations).
- The program does not use either "`::`" or "`->`" operators. The "`::`" is not an operator at all, and the "`->`" operator is used in only a small fraction of programs — those that employ unsafe code (Section A). The separator "`.`" is used in compound names such as `Console.WriteLine`.
- The program does not contain forward declarations. Forward declarations are never needed, as declaration order is not significant.
- The program does not use `#include` to import program text. Dependencies among programs are handled symbolically rather than textually. This approach eliminates barriers between applications written using multiple languages. For example, the `Console` class need not be written in C#.

1.2 Types

C# supports two kinds of types: value types and reference types. Value types include simple types (e.g., `char`, `int`, and `float`), enum types, and struct types. Reference types include class types, interface types, delegate types, and array types.

Value types differ from reference types in that variables of the value types directly contain their data, whereas variables of the reference types store references to objects. With reference types, it is possible for two variables to reference the same object, and thus possible for operations on one variable to affect the object referenced by the other variable. With value types, the variables each have their own copy of the data, and it is not possible for operations on one to affect the other.

The example

```
using System;
class Class1
{
   public int Value = 0;
}
class Test
{
   static void Main() {
      int val1 = 0;
      int val2 = val1;
      val2 = 123;
      Class1 ref1 = new Class1();
      Class1 ref2 = ref1;
      ref2.Value = 123;
      Console.WriteLine("Values: {0}, {1}", val1, val2);
      Console.WriteLine("Refs: {0}, {1}", ref1.Value, ref2.Value);
   }
}
```

shows this difference. The output produced is

```
Values: 0, 123
Refs: 123, 123
```

The assignment to the local variable `val1` does not impact the local variable `val2` because both local variables are of a value type (the type `int`) and each local variable of a value type has its own storage. In contrast, the assignment `ref2.Value = 123;` affects the object that both `ref1` and `ref2` reference.

The lines

```
Console.WriteLine("Values: {0}, {1}", val1, val2);
Console.WriteLine("Refs: {0}, {1}", ref1.Value, ref2.Value);
```

deserve further comment, as they demonstrate some of the string formatting behavior of `Console.WriteLine`, which, in fact, takes a variable number of arguments. The first argument is a string, which may contain numbered placeholders like `{0}` and `{1}`. Each placeholder refers to a trailing argument with `{0}` referring to the second argument, `{1}` referring to the third argument, and so on. Before the output is sent to the console, each placeholder is replaced with the formatted value of its corresponding argument.

Developers can define new value types through enum and struct declarations, and can define new reference types via class, interface, and delegate declarations. The example

```
using System;
public enum Color
{
   Red, Blue, Green
}
public struct Point
{
   public int x, y;
}
public interface IBase
```

```
{
   void F();
}
public interface IDerived: IBase
{
   void G();
}
public class A
{
   protected virtual void H() {
      Console.WriteLine("A.H");
   }
}
public class B: A, IDerived
{
   public void F() {
      Console.WriteLine("B.F, implementation of IDerived.F");
   }
   public void G() {
      Console.WriteLine("B.G, implementation of IDerived.G");
   }
   override protected void H() {
      Console.WriteLine("B.H, override of A.H");
   }
}
public delegate void EmptyDelegate();
```

shows an example of each kind of type declaration. Later sections describe type declarations in detail.

1.2.1 Predefined types

C# provides a set of predefined types, most of which will be familiar to C and C++ developers.

The predefined reference types are `object` and `string`. The type `object` is the ultimate base type of all other types. The type `string` is used to represent Unicode string values. Values of type `string` are immutable.

The predefined value types include signed and unsigned integral types, floating point types, and the types `bool`, `char`, and `decimal`. The signed integral types are `sbyte`, `short`, `int`, and `long`; the unsigned integral types are `byte`, `ushort`, `uint`, and `ulong`; and the floating point types are `float` and `double`.

The `bool` type is used to represent Boolean values: values that are either true or false. The inclusion of `bool` makes it easier to write self-documenting code, and also helps eliminate the all-too-common C++ coding error in which a developer mistakenly uses "`=`" when "`==`" should have been used. In C#, the example

```
int i = ...;
F(i);
if (i = 0)   // Bug: the test should be (i == 0)
   G();
```

results in a compile-time error because the expression `i = 0` is of type `int`, and `if` statements require an expression of type `bool`.

The `char` type is used to represent Unicode characters. A variable of type `char` represents a single 16-bit Unicode character.

The `decimal` type is appropriate for calculations in which rounding errors caused by floating point representations are unacceptable. Common examples include financial calculations such as tax computations and currency conversions. The `decimal` type provides 28 significant digits.

The table below lists the predefined types, and shows how to write literal values for each of them.

Type	Description	Example
`object`	The ultimate base type of all other types	`object o = null;`
`string`	String type; a string is a sequence of Unicode characters	`string s = "hello";`
`sbyte`	8-bit signed integral type	`sbyte val = 12;`
`short`	16-bit signed integral type	`short val = 12;`
`int`	32-bit signed integral type	`int val = 12;`
`long`	64-bit signed integral type	`long val1 = 12;` `long val2 = 34L;`
`byte`	8-bit unsigned integral type	`byte val1 = 12;`
`ushort`	16-bit unsigned integral type	`ushort val1 = 12;`
`uint`	32-bit unsigned integral type	`uint val1 = 12;` `uint val2 = 34U;`

`ulong`	64-bit unsigned integral type	`ulong val1 = 12;`
		`ulong val2 = 34U;`
		`ulong val3 = 56L;`
		`ulong val4 = 78UL;`

`float`	Single-precision floating point type	`float val = 1.23F;`
`double`	Double-precision floating point type	`double val1 = 1.23;` `double val2 = 4.56D;`
`bool`	Boolean type; a `bool` value is either true or false	`bool val1 = true;` `bool val2 = false;`
`char`	Character type; a `char` value is a Unicode character	`char val = 'h';`
`decimal`	Precise decimal type with 28 significant digits	`decimal val = 1.23M;`

Each of the predefined types is shorthand for a system-provided type. For example, the keyword `int` refers to the struct `System.Int32`. As a matter of style, use of the keyword is favored over use of the complete system type name.

Predefined value types such as `int` are treated specially in a few ways but are for the most part treated exactly like other structs. Operator overloading enables developers to define new struct types that behave much like the predefined value types. For instance, a `Digit` struct can support the same mathematical operations as the predefined integral types, and can define conversions between `Digit` and predefined types.

The predefined types employ operator overloading themselves. For example, the comparison operators `==` and `!=` have different semantics for different predefined types:

- Two expressions of type `int` are considered equal if they represent the same integer value.
- Two expressions of type `object` are considered equal if both refer to the same object, or if both are `null`.
- Two expressions of type `string` are considered equal if the string instances have identical lengths and identical characters in each character position, or if both are `null`.

The example

```
using System;
class Test
{
   static void Main() {
      string s = "Test";
      string t = string.Copy(s);
      Console.WriteLine(s == t);
      Console.WriteLine((object)s == (object)t);
   }
}
```

produces the output

```
True
False
```

because the first comparison compares two expressions of type `string`, and the second comparison compares two expressions of type `object`.

1.2.2 Conversions

The predefined types also have predefined conversions. For instance, conversions exist between the predefined types `int` and `long`. C# differentiates between two kinds of conversions: implicit conversions and explicit conversions. Implicit conversions are supplied for conversions that can safely be performed without careful scrutiny. For instance, the conversion from `int` to `long` is an implicit conversion. This conversion always succeeds, and never results in a loss of information. The following example

```
using System;
class Test
{
   static void Main() {
      int intValue = 123;
      long longValue = intValue;
      Console.WriteLine("{0}, {1}", intValue, longValue);
   }
}
```

implicitly converts an `int` to a `long`.

In contrast, explicit conversions are performed with a cast expression. The example

```
using System;
class Test
{
   static void Main() {
      long longValue = Int64.MaxValue;
      int intValue = (int) longValue;
      Console.WriteLine("(int){0} = {1}", longValue, intValue);
   }
}
```

uses an explicit conversion to convert a `long` to an `int`. The output is:

```
(int) 9223372036854775807 = -1
```

because an overflow occurs. Cast expressions permit the use of both implicit and explicit conversions.

1.2.3 Array types

Arrays may be single-dimensional or multi-dimensional. Both "rectangular" and "jagged" arrays are supported.

Single-dimensional arrays are the most common type. The example

```
using System;
class Test
{
   static void Main() {
      int[] arr = new int[5];
      for (int i = 0; i < arr.Length; i++)
         arr[i] = i * i;
      for (int i = 0; i < arr.Length; i++)
         Console.WriteLine("arr[{0}] = {1}", i, arr[i]);
   }
}
```

creates a single-dimensional array of `int` values, initializes the array elements, and then prints each of them out. The output produced is:

```
arr[0] = 0
arr[1] = 1
arr[2] = 4
arr[3] = 9
arr[4] = 16
```

The type `int[]` used in the previous example is an array type. Array types are written using a non-*array-type* followed by one or more rank specifiers. The example

```
class Test
{
   static void Main() {
      int[] a1;         // single-dimensional array of int
      int[,] a2;         // 2-dimensional array of int
      int[,,] a3;         // 3-dimensional array of int
      int[][] j2;         // "jagged" array: array of (array of int)
      int[][][] j3;      // array of (array of (array of int))
   }
}
```

shows a variety of local variable declarations that use array types with `int` as the element type.

Array types are reference types, and so the declaration of an array variable merely sets aside space for the reference to the array. Array instances are actually created via array initializers and array creation expressions. The example

```
class Test
{
   static void Main() {
      int[] a1 = new int[] {1, 2, 3};
      int[,] a2 = new int[,] {{1, 2, 3}, {4, 5, 6}};
      int[,,] a3 = new int[10, 20, 30];
      int[][] j2 = new int[3][];
      j2[0] = new int[] {1, 2, 3};
      j2[1] = new int[] {1, 2, 3, 4, 5, 6};
      j2[2] = new int[] {1, 2, 3, 4, 5, 6, 7, 8, 9};
   }
}
```

shows a variety of array creation expressions. The variables `a1`, `a2` and `a3` denote rectangular arrays, and the variable `j2` denotes a jagged array. It should be no surprise that these terms are based on the shapes of the arrays. Rectangular arrays always have a rectangular shape. Given the length of each dimension of the array, its rectangular shape is clear. For example, the lengths of `a3`'s

three dimensions are 10, 20, and 30 respectively, and it is easy to see that this array contains `10*20*30` elements.

In contrast, the variable `j2` denotes a "jagged" array, or an "array of arrays". Specifically, `j2` denotes an array of an array of `int`, or a single-dimensional array of type `int[]`. Each of these `int[]` variables can be initialized individually, and this allows the array to take on a jagged shape. The example gives each of the `int[]` arrays a different length. Specifically, the length of `j2[0]` is `3`, the length of `j2[1]` is `6`, and the length of `j2[2]` is `9`.

In C++, an array declared as `int x[3][5][7]` would be considered a three-dimensional rectangular array, while in C#, the declaration `int[][][]` declares a jagged array type.

The element type and shape of an array — including whether it is jagged or rectangular, and the number of dimensions it has — are part of its type. On the other hand, the size of the array — as represented by the length of each of its dimensions — is not part of an array's type. This split is made clear in the language syntax, as the length of each dimension is specified in the array creation expression rather than in the array type. For instance the declaration

```
int[,,] a3 = new int[10, 20, 30];
```

has an array type of `int[,,]` and an array creation expression of `new int[10, 20, 30]`.

For local variable and field declarations, a shorthand form is permitted so that it is not necessary to re-state the array type. For instance, the example

```
int[] a1 = new int[] {1, 2, 3};
```

can be shortened to

```
int[] a1 = {1, 2, 3};
```

without any change in program semantics.

The context in which an array initializer such as `{1, 2, 3}` is used determines the type of the array being initialized. The example

```
class Test
{
   static void Main() {
      short[] a = {1, 2, 3};
      int[]   b = {1, 2, 3};
      long[] c = {1, 2, 3};
   }
}
```

shows that the same array initializer syntax can be used for several different array types. Because context is required to determine the type of an array initializer, it is not possible to use an array initializer in an expression context without explicitly stating the type of the array.

1.2.4 Type system unification

C# provides a "unified type system". All types — including value types — derive from the type `object`. It is possible to call object methods on any value, even values of "primitive" types such as `int`. The example

```
using System;
class Test
{
   static void Main() {
      Console.WriteLine(3.ToString());
   }
}
```

calls the `object`-defined `ToString` method on an integer literal, resulting in the output "`3`".

The example

```
class Test
{
   static void Main() {
      int i = 123;
      object o = i;      // boxing
      int j = (int) o;   // unboxing
   }
}
```

is more interesting. An `int` value can be converted to `object` and back again to `int`. This example shows both boxing and unboxing. When a variable of a value type needs to be converted to a reference type, an object box is allocated to hold the value, and the value is copied into the box. Unboxing is just the opposite. When an object box is cast back to its original value type, the value is copied out of the box and into the appropriate storage location.

This type system unification provides value types with the benefits of object-ness without introducing unnecessary overhead. For programs that don't need `int` values to act like objects, `int` values are simply 32-bit values. For programs that need `int` values to behave like objects, this capability is available on demand. This ability to treat value types as objects bridges the gap between value types and reference types that exists in most languages. For example, a `Stack` class can provide `Push` and `Pop` methods that take and return `object` values.

```
public class Stack
{
   public object Pop() {...}
   public void Push(object o) {...}
}
```

Because C# has a unified type system, the `Stack` class can be used with elements of any type, including value types like `int`.

1.3 Variables and parameters

Variables represent storage locations. Every variable has a type that determines what values can be stored in the variable. Local variables are variables that are declared in methods, properties, or indexers. A local variable is defined by specifying a type name and a declarator that specifies the variable name and an optional initial value, as in:

```
int a;
int b = 1;
```

but it is also possible for a local variable declaration to include multiple declarators. The declarations of `a` and `b` can be rewritten as:

```
int a, b = 1;
```

A variable must be assigned before its value can be obtained. The example

```
class Test
{
   static void Main() {
      int a;
      int b = 1;
      int c = a + b;   // error, a not yet assigned
      ...
   }
}
```

results in a compile-time error because it attempts to use the variable `a` before it is assigned a value. The rules governing definite assignment are defined in Section 5.3.

A field (Section 10.4) is a variable that is associated with a class or struct, or with an instance of a class or struct. A field declared with the `static` modifier defines a static variable, and a field declared without this modifier defines an instance variable. A static field is associated with a type, whereas an instance variable is associated with an instance. The example

```
using Personnel.Data;
class Employee
{
   private static DataSet ds;
   public string Name;
   public decimal Salary;
   ...
}
```

shows an `Employee` class that has a private static variable and two public instance variables.

Formal parameter declarations also define variables. There are four kinds of parameters: value parameters, reference parameters, output parameters, and parameter arrays.

A value parameter is used for "in" parameter passing, in which the value of an argument is passed into a method, and modifications of the parameter do not impact the original argument. A value parameter refers to its own variable, one that is distinct from the corresponding argument. This variable is initialized by copying the value of the corresponding argument. The example

```
using System;
class Test {
   static void F(int p) {
      Console.WriteLine("p = {0}", p);
      p++;
   }
   static void Main() {
      int a = 1;
      Console.WriteLine("pre:  a = {0}", a);
      F(a);
```

```
      Console.WriteLine("post: a = {0}", a);
   }
}
```

shows a method `F` that has a value parameter named `p`. The example produces the output

```
pre:  a = 1
p = 1
post: a = 1
```

even though the value parameter `p` is modified.

A reference parameter is used for "by reference" parameter passing, in which the parameter acts as an alias for a caller-provided argument. A reference parameter does not itself define a variable, but rather refers to the variable of the corresponding argument. Modifications of a reference parameter impact the corresponding argument. A reference parameter is declared with a `ref` modifier. The example

```
using System;
class Test {
   static void Swap(ref int a, ref int b) {
      int t = a;
      a = b;
      b = t;
   }
   static void Main() {
      int x = 1;
      int y = 2;

      Console.WriteLine("pre:  x = {0}, y = {1}", x, y);
      Swap(ref x, ref y);
      Console.WriteLine("post: x = {0}, y = {1}", x, y);
   }
}
```

shows a `Swap` method that has two reference parameters. The output produced is:

```
pre:  x = 1, y = 2
post: x = 2, y = 1
```

The `ref` keyword must be used in both the declaration of the formal parameter and in uses of it. The use of `ref` at the call site calls special attention to the parameter so that a developer reading the code will understand that the value of the argument could change as a result of the call.

An output parameter is similar to a reference parameter, except that the initial value of the caller-provided argument is unimportant. An output parameter is declared with an `out` modifier. The example

```
using System;
class Test {
   static void Divide(int a, int b, out int result, out int remainder) {
      result = a / b;
      remainder = a % b;
   }
   static void Main() {
      for (int i = 1; i < 10; i++)
         for (int j = 1; j < 10; j++) {
            int ans, r;
            Divide(i, j, out ans, out r);
            Console.WriteLine("{0} / {1} = {2}r{3}", i, j, ans, r);
         }
   }
}
```

shows a `Divide` method that includes two output parameters — one for the result of the division and another for the remainder.

For value, reference, and output parameters, there is a one-to-one correspondence between caller-provided arguments and the parameters used to represent them. A parameter array enables a many-to-one relationship: many arguments can be represented by a single parameter array. In other words, parameter arrays enable variable length argument lists.

A parameter array is declared with a `params` modifier. There can be only one parameter array for a given method, and it must always be the last parameter specified. The type of a parameter array is always a single dimensional array type. A caller can either pass a single argument of this array type, or any number of arguments of the element type of this array type. For instance, the example

```
using System;
class Test
{
   static void F(params int[] args) {
      Console.WriteLine("# of arguments: {0}", args.Length);
      for (int i = 0; i < args.Length; i++)
         Console.WriteLine("\targs[{0}] = {1}", i, args[i]);
   }
   static void Main() {
      F();
      F(1);
      F(1, 2);
      F(1, 2, 3);
      F(new int[] {1, 2, 3, 4});
   }
}
```

shows a method `F` that takes a variable number of `int` arguments, and several invocations of this method. The output is:

```
# of arguments: 0
# of arguments: 1
   args[0] = 1
# of arguments: 2
   args[0] = 1
   args[1] = 2
# of arguments: 3
   args[0] = 1
   args[1] = 2
   args[2] = 3
# of arguments: 4
   args[0] = 1
   args[1] = 2
   args[2] = 3
   args[3] = 4
```

Most of the examples presented in this introduction use the `WriteLine` method of the `Console` class. The argument substitution behavior of this method, as exhibited in the example

```
int a = 1, b = 2;
Console.WriteLine("a = {0}, b = {1}", a, b);
```

is accomplished using a parameter array. The `WriteLine` method provides several overloaded methods for the common cases in which a small number of arguments are passed, and one method that uses a parameter array.

```
namespace System
{
   public class Console
   {
      public static void WriteLine(string s) {...}
      public static void WriteLine(string s, object a) {...}
      public static void WriteLine(string s, object a, object b) {...}
      ...
      public static void WriteLine(string s, params object[] args) {...}
   }
}
```

1.4 Automatic memory management

Manual memory management requires developers to manage the allocation and de-allocation of blocks of memory. Manual memory management can be both time-consuming and difficult. In C#, automatic memory management is provided so that developers are freed from this burdensome task. In the vast majority of cases, automatic memory management increases code quality and enhances developer productivity without negatively impacting either expressiveness or performance.

The example

```
using System;
public class Stack
{
   private Node first = null;
   public bool Empty {
      get {
         return (first == null);
      }
   }
   public object Pop() {
      if (first == null)
         throw new Exception("Can't Pop from an empty Stack.");
      else {
         object temp = first.Value;
         first = first.Next;
         return temp;
      }
   }
   public void Push(object o) {
      first = new Node(o, first);
   }
   private class Node
   {
      public Node Next;
      public object Value;
      public Node(object value): this(value, null) {}
      public Node(object value, Node next) {
         Next = next;
         Value = value;
      }
   }
}
```

shows a `Stack` class implemented as a linked list of `Node` instances. Node instances are created in the `Push` method and are garbage collected when no longer needed. A `Node` instance becomes eligible for garbage collection when it is no longer possible for any code to access it. For instance, when an item is removed from the `Stack`, the associated `Node` instance becomes eligible for garbage collection.

The example

```
class Test
{
   static void Main() {
      Stack s = new Stack();
      for (int i = 0; i < 10; i++)
         s.Push(i);
      s = null;
   }
}
```

shows code that uses the `Stack` class. A `Stack` is created and initialized with 10 elements, and then assigned the value `null`. Once the variable `s` is assigned null, the `Stack` and the associated 10 `Node` instances become eligible for garbage collection. The garbage collector is permitted to clean up immediately, but is not required to do so.

The garbage collector underlying C# may work by moving objects around in memory, but this motion is invisible to most C# developers. For developers who are generally content with automatic memory management but sometimes need fine-grained

control or that extra bit of performance, C# provides the ability to write "unsafe" code. Such code can deal directly with pointer types and object addresses. However, C# requires the programmer to fix objects to temporarily prevent the garbage collector from moving them.

This "unsafe" code feature is in fact a "safe" feature from the perspective of both developers and users. Unsafe code must be clearly marked in the code with the modifier `unsafe`, so developers can't possibly use unsafe language features accidentally, and the compiler and the execution engine work together to ensure that unsafe code cannot masquerade as safe code. These restrictions limit the use of unsafe code to situations in which the code is trusted.

The example

```
using System;
class Test
{
   static void WriteLocations(byte[] arr) {
      unsafe {
         fixed (byte* pArray = arr) {
            byte* pElem = pArray;
            for (int i = 0; i < arr.Length; i++) {
               byte value = *pElem;
               Console.WriteLine("arr[{0}] at 0x{1:X} is {2}",
                  i, (uint)pElem, value);
               pElem++;
            }
         }
      }
   }
   static void Main() {
      byte[] arr = new byte[] {1, 2, 3, 4, 5};
      WriteLocations(arr);
   }
}
```

shows an unsafe block in a method named `WriteLocations` that fixes an array instance and uses pointer manipulation to iterate over the elements. The index, value, and location of each array element are written to the console. One possible example of output is:

```
arr[0] at 0x8E0360 is 1
arr[1] at 0x8E0361 is 2
arr[2] at 0x8E0362 is 3
arr[3] at 0x8E0363 is 4
arr[4] at 0x8E0364 is 5
```

but of course the exact memory locations may be different in different executions of the application.

1.5 Expressions

C# includes unary operators, binary operators, and one ternary operator. The following table summarizes the operators, listing them in order of precedence from highest to lowest:

Section	Category	Operators
Section 7.5	Primary	`x.y  f(x)  a[x]  x++  x--  new` `typeof  checked  unchecked`
Section 7.6	Unary	`+  -  !  ~  ++x  --x  (T)x`
Section 7.7	Multiplicative	`*  /  %`
Section 7.7	Additive	`+  -`
Section 7.8	Shift	`<<  >>`
Section 7.9	Relational and type testing	`<  >  <=  >=  is  as`
Section 7.9	Equality	`==  !=`
Section 7.10	Logical AND	`&`
Section 7.10	Logical XOR	`^`
Section 7.10	Logical OR	`|`
Section 7.11	Conditional AND	`&&`
Section 7.11	Conditional OR	`||`
Section 7.12	Conditional	`?:`
Section 7.13	Assignment	`=  *=  /=  %=  +=  -=  <<=  >>=  &=  ^=  |=`

When an expression contains multiple operators, the precedence of the operators controls the order in which the individual operators are evaluated. For example, the expression `x + y * z` is evaluated as `x + (y * z)` because the `*` operator has higher precedence than the `+` operator.

When an operand occurs between two operators with the same precedence, the associativity of the operators controls the order in which the operations are performed:

- Except for the assignment operators, all binary operators are left-associative, meaning that operations are performed from left to right. For example, `x + y + z` is evaluated as `(x + y) + z`.

- The assignment operators and the conditional operator (`?:`) are right-associative, meaning that operations are performed from right to left. For example, `x = y = z` is evaluated as `x = (y = z)`.

Precedence and associativity can be controlled using parentheses. For example, `x + y * z` first multiplies `y` by `z` and then adds the result to `x`, but `(x + y) * z` first adds `x` and `y` and then multiplies the result by `z`.

1.6 Statements

C# borrows most of its statements directly from C and C++, though there are some noteworthy additions and modifications. The table below lists the kinds of statements that can be used, and provides an example for each.

Statement	Example

Statement lists and block statements	`static void Main() {`
	`  F();`
	`  G();`
	`  {`
	`  H();`
	`  I();`
	`  }`
	`}`

Labeled statements and `goto` statements	`static void Main(string[] args) {`
	`  if (args.Length == 0)`
	`  goto done;`
	`  Console.WriteLine(args.Length);`
	
	`done:`
	`  Console.WriteLine("Done");`
	`}`

Local constant declarations	`static void Main() {`
	`  const float pi = 3.14f;`
	`  const int r = 123;`
	`  Console.WriteLine(pi * r * r);`
	`}`

Local variable declarations	`static void Main() {`
	`  int a;`
	`  int b = 2, c = 3;`
	`  a = 1;`
	`  Console.WriteLine(a + b + c);`
	`}`

Expression statements	`static int F(int a, int b) {`
	`  return a + b;`
	`}`
	`static void Main() {`
	`  F(1, 2);  // Expression statement`
	`}`

`if` statements	`static void Main(string[] args) {`
	`  if (args.Length == 0)`
	`  Console.WriteLine("No args");`
	`  else`
	`  Console.WriteLine("Args");`
	`}`

`switch` statements	`static void Main(string[] args) {`
	`  switch (args.Length) {`
	`  case 0:`
	`  Console.WriteLine("No args");`
	`  break;`
	`  case 1:`
	`  Console.WriteLine("One arg ");`
	`  break;`
	`  default:`
	`  int n = args.Length;`
	`  Console.WriteLine("{0} args", n);`
	`  break;`
	`  }`
	`}`

`while` statements	`static void Main(string[] args) {`
	`  int i = 0;`
	`  while (i < args.Length) {`
	`  Console.WriteLine(args[i]);`
	`  i++;`
	`  }`
	`}`

`do` statements	`static void Main() {`
	`  string s;`
	`  do { s = Console.ReadLine(); }`
	`  while (s != "Exit");`
	`}`

`for` statements	`static void Main(string[] args) {`
	`  for (int i = 0; i < args.Length; i++)`
	`  Console.WriteLine(args[i]);`
	`}`

`foreach` statements	`static void Main(string[] args) {`
	`  foreach (string s in args)`
	`  Console.WriteLine(s);`
	`}`

`break` statements	`static void Main(string[] args) {`
	`  int i = 0;`
	`  while (true) {`
	`  if (i == args.Length)`
	`  break;`
	`  Console.WriteLine(args[i++]);`
	`  }`
	`}`

`continue` statements	`static void Main(string[] args) {`
	`  int i = 0;`
	`  while (true) {`
	`  Console.WriteLine(args[i++]);`
	`  if (i < args.Length)`
	`  continue;`
	`  break;`
	`  }`
	`}`

`return` statements	`static int F(int a, int b) {`
	`  return a + b;`
	`}`
	`static void Main() {`
	`  Console.WriteLine(F(1, 2));`
	`  return;`
	`}`

`throw` statements and `try` statements	`static int F(int a, int b) {`
	`  if (b == 0)`
	`  throw new Exception("Divide by zero");`
	`  return a / b;`
	`}`
	`static void Main() {`
	`  try {`
	`  Console.WriteLine(F(5, 0));`
	`  }`
	`  catch(Exception e) {`
	`  Console.WriteLine("Error");`
	`  }`
	`}`

`checked` and `unchecked` statements	`static void Main() {`
	`  int x = Int32.MaxValue;`
	
	`  Console.WriteLine(x + 1);  // Overflow`
	
	`  checked {`
	`  Console.WriteLine(x + 1);  // Exception`
	`  }`
	
	`  unchecked {`
	`  Console.WriteLine(x + 1);  // Overflow`
	`  }`
	`}`

`lock` statements	`static void Main() {`
	`  A a = ...;`
	`  lock(a) {`
	`  a.P = a.P + 1;`
	`  }`
	`}`

using statements	`static void Main() {`
	`  using (Resource r = new Resource()) {`
	`  r.F();`
	`  }`
	`}`

1.7 Classes

Class declarations define new reference types. A class can inherit from another class, and can implement interfaces.

Class members can include constants, fields, methods, properties, events, indexers, operators, instance constructors, destructors, static constructors, and nested type declarations. Each member has an associated accessibility (Section 3.5), which controls the regions of program text that are able to access the member. There are five possible forms of accessibility. These are summarized in the table below.

Form	Intuitive meaning
`public`	Access not limited.
`protected`	Access limited to the containing class or types derived from the containing class.
`internal`	Access limited to this program.
`protected internal`	Access limited to this program or types derived from the containing class.
`private`	Access limited to the containing type.

The example

```
using System;
class MyClass
{
   public MyClass() {
      Console.WriteLine("Instance constructor");
   }
   public MyClass(int value) {
      MyField = value;
      Console.WriteLine("Instance constructor");
   }
   ~MyClass() {
      Console.WriteLine("Destructor");
   }
   public const int MyConst = 12;
   public int MyField = 34;
   public void MyMethod(){
      Console.WriteLine("MyClass.MyMethod");
   }
   public int MyProperty {
      get {
         return MyField;
      }
      set {
         MyField = value;
      }
   }
   public int this[int index] {
      get {
         return 0;
      }
      set {
         Console.WriteLine("this[{0}] = {1}", index, value);
      }
   }
   public event EventHandler MyEvent;
```

```
   public static MyClass operator+(MyClass a, MyClass b) {
      return new MyClass(a.MyField + b.MyField);
   }
   internal class MyNestedClass
   {}
}
```

shows a class that contains each kind of member. The example

```
class Test
{
   static void Main() {
      // Instance constructor usage
      MyClass a = new MyClass();
      MyClass b = new MyClass(123);
      // Constant usage
      Console.WriteLine("MyConst = {0}", MyClass.MyConst);
      // Field usage
      a.MyField++;
      Console.WriteLine("a.MyField = {0}", a.MyField);
      // Method usage
      a.MyMethod();
      // Property usage
      a.MyProperty++;
      Console.WriteLine("a.MyProperty = {0}", a.MyProperty);
      // Indexer usage
      a[3] = a[1] = a[2];
      Console.WriteLine("a[3] = {0}", a[3]);
      // Event usage
      a.MyEvent += new EventHandler(MyHandler);
      // Overloaded operator usage
      MyClass c = a + b;
   }
   static void MyHandler(object sender, EventArgs e) {
      Console.WriteLine("Test.MyHandler");
   }
   internal class MyNestedClass
   {}
}
```

shows uses of these members.

1.7.1 Constants

A constant is a class member that represents a constant value: a value that can be computed at compile-time. Constants are permitted to depend on other constants within the same program as long as there are no circular dependencies. The rules governing constant expressions are defined in Section 7.15. The example

```
class Constants
{
   public const int A = 1;
   public const int B = A + 1;
}
```

shows a class named `Constants` that has two public constants.

Even though constants are considered static members, a constant declaration neither requires nor allows the `static` modifier. Constants can be accessed through the class, as in

```
using System;
class Test
{
   static void Main() {
      Console.WriteLine("{0}, {1}", Constants.A, Constants.B);
   }
}
```

which prints out the values of `Constants.A` and `Constants.B`, respectively.

1.7.2 Fields

A field is a member that represents a variable associated with an object or class. The example

```
class Color
{
   public static Color Red = new Color(0xFF, 0, 0);
   public static Color Blue = new Color(0, 0xFF, 0);
   public static Color Green = new Color(0, 0, 0xFF);
   public static Color White = new Color(0xFF, 0xFF, 0xFF);
   internal ushort redPart;
   internal ushort bluePart;
   internal ushort greenPart;
   public Color(ushort red, ushort blue, ushort green) {
      redPart = red;
      bluePart = blue;
      greenPart = green;
   }
}
```

which shows a `Color` class that has internal instance fields named `redPart`, `bluePart`, and `greenPart`, and static fields named `Red`, `Blue`, `Green`, and `White`.

The use of static fields in this manner is not ideal. The fields are initialized at some point before they are used, but after this initialization there is nothing to stop a client from changing them. Such a modification could cause unpredictable errors in other programs that use `Color` and assume that the values do not change. `Readonly` fields can be used to prevent such problems. Assignments to a `readonly` field can only occur as part of the declaration, or in an instance constructor or static constructor in the same class. A static `readonly` field can be assigned in a static constructor, and a non-static `readonly` field can be assigned in an instance constructor. Thus, the `Color` class can be enhanced by adding the `readonly` modifier to the static fields:

```
class Color
{
   public static readonly Color Red = new Color(0xFF, 0, 0);
   public static readonly Color Blue = new Color(0, 0xFF, 0);
   public static readonly Color Green = new Color(0, 0, 0xFF);
   public static readonly Color White = new Color(0xFF, 0xFF, 0xFF);
   internal ushort redPart;
   internal ushort bluePart;
   internal ushort greenPart;
   public Color(ushort red, ushort blue, ushort green) {
      redPart = red;
      bluePart = blue;
      greenPart = green;
   }
}
```

1.7.3 Methods

A method is a member that implements a computation or action that can be performed by an object or class. Methods have a possibly empty list of formal parameters, a return value (unless the method's *return-type* is `void`), and are either static or non-static. Static methods are accessed through the class. Non-static methods, which are also called instance methods, are accessed through instances of the class. The example

```
using System;
public class Stack
{
   public static Stack Clone(Stack s) {...}
   public static Stack Flip(Stack s) {...}
   public object Pop() {...}
   public void Push(object o) {...}
   public override string ToString() {...}
   ...
}
class Test
{
   static void Main() {
      Stack s = new Stack();
      for (int i = 1; i < 10; i++)
         s.Push(i);
      Stack flipped = Stack.Flip(s);
      Stack cloned = Stack.Clone(s);
      Console.WriteLine("Original stack: " + s.ToString());
      Console.WriteLine("Flipped stack: " + flipped.ToString());
      Console.WriteLine("Cloned stack: " + cloned.ToString());
   }
}
```

shows a `Stack` that has several static methods (`Clone` and `Flip`) and several instance methods (`Push`, `Pop`, and `ToString`).

Methods can be overloaded, which means that multiple methods may have the same name so long as they have unique signatures. The signature of a method consists of the name of the method and the number, modifiers, and types of its formal parameters. The signature of a method does not include the return type. The example

```
using System;
class Test
{
   static void F() {
      Console.WriteLine("F()");
   }
   static void F(object o) {
      Console.WriteLine("F(object)");
   }
   static void F(int value) {
      Console.WriteLine("F(int)");
   }
   static void F(ref int value) {
      Console.WriteLine("F(ref int)");
   }
   static void F(int a, int b) {
      Console.WriteLine("F(int, int)");
   }
   static void F(int[] values) {
      Console.WriteLine("F(int[])");
   }
   static void Main() {
      F();
      F(1);
      int i = 10;
      F(ref i);
      F((object)1);
      F(1, 2);
```

```
        F(new int[] {1, 2, 3});
    }
}
```

shows a class with a number of methods called `F`. The output produced is

```
F()
F(int)
F(ref int)
F(object)
F(int, int)
F(int[])
```

1.7.4 Properties

A property is a member that provides access to a characteristic of an object or a class. Examples of properties include the length of a string, the size of a font, the caption of a window, the name of a customer, and so on. Properties are a natural extension of fields. Both are named members with associated types, and the syntax for accessing fields and properties is the same. However, unlike fields, properties do not denote storage locations. Instead, properties have accessors that specify the statements to be executed when their values are read or written.

Properties are defined with property declarations. The first part of a property declaration looks quite similar to a field declaration. The second part includes a get accessor and/or a set accessor. In the example below, the `Button` class defines a `Caption` property.

```
public class Button
{
   private string caption;
   public string Caption {
      get {
         return caption;
      }
      set {
         caption = value;
         Repaint();
      }
   }
}
```

Properties that can be both read and written, such as `Caption`, include both get and set accessors. The get accessor is called when the property's value is read; the set accessor is called when the property's value is written. In a set accessor, the new value for the property is made available via an implicit parameter named `value`.

The declaration of properties is relatively straightforward, but the real value of properties is seen when they are used. For example, the `Caption` property can be read and written in the same way that fields can be read and written:

```
Button b = new Button();
b.Caption = "ABC";      // set; causes repaint
string s = b.Caption;   // get
b.Caption += "DEF";      // get & set; causes repaint
```

1.7.5 Events

An event is a member that enables an object or class to provide notifications. A class defines an event by providing an event declaration (which resembles a field declaration) though with an added `event` keyword, and an optional set of event accessors. The type of this declaration must be a delegate type.

An instance of a delegate type encapsulates one or more callable entities. For instance methods, a callable entity consists of an instance and a method on that instance. For static methods, a callable entity consists of just a method. Invoking a delegate instance with an appropriate set of arguments causes each of the delegate instance's callable entities to be invoked with the given set of arguments. '

In the example

```
public delegate void EventHandler(object sender, System.EventArgs e);
public class Button
{
   public event EventHandler Click;
   public void Reset() {
      Click = null;
   }
}
```

the `Button` class defines a `Click` event of type `EventHandler`. Inside the `Button` class, the `Click` member is exactly like a private field of type `EventHandler`. However, outside the `Button` class, the `Click` member can only be used on the left-hand side of the `+=` and `-=` operators. The `+=` operator adds a handler for the event, and the `-=` operator removes a handler for the event. The example

```
using System;
public class Form1
{
   public Form1() {
      // Add Button1_Click as an event handler for Button1's Click event
      Button1.Click += new EventHandler(Button1_Click);
   }
   Button Button1 = new Button();
   void Button1_Click(object sender, EventArgs e) {
      Console.WriteLine("Button1 was clicked!");
   }
   public void Disconnect() {
      Button1.Click -= new EventHandler(Button1_Click);
   }
}
```

shows a `Form1` class that adds `Button1_Click` as an event handler for `Button1`'s `Click` event. In the `Disconnect` method, that event handler is removed.

For a simple event declaration such as

```
public event EventHandler Click;
```

the compiler automatically provides the implementation underlying the `+=` and `-=` operators.

An implementer who wants more control can get it by explicitly providing add and remove accessors. For example, the `Button` class could be rewritten as follows:

```
public class Button
{
      private EventHandler handler;
      public event EventHandler Click {
         add { handler += value; }
         remove { handler -= value; }
      }
}
```

This change has no effect on client code, but allows the `Button` class more implementation flexibility. For example, the event handler for `Click` need not be represented by a field.

1.7.6 Operators

An operator is a member that defines the meaning of an expression operator that can be applied to instances of the class. There are three kinds of operators that can be defined: unary operators, binary operators, and conversion operators. All operators must be declared as public and static.

The following example defines a `Digit` type that represents decimal digits — integral values between `0` and `9`.

```
using System;
public struct Digit
{
   byte value;
   public Digit(byte value) {
      if (value < 0 || value > 9) throw new ArgumentException();
      this.value = value;
   }
   public Digit(int value): this((byte) value) {}
   public static implicit operator byte(Digit d) {
      return d.value;
   }
   public static explicit operator Digit(byte b) {
      return new Digit(b);
   }
   public static Digit operator+(Digit a, Digit b) {
      return new Digit(a.value + b.value);
   }
   public static Digit operator-(Digit a, Digit b) {
      return new Digit(a.value - b.value);
   }
   public static bool operator==(Digit a, Digit b) {
      return a.value == b.value;
   }
   public static bool operator!=(Digit a, Digit b) {
      return a.value != b.value;
   }
   public override bool Equals(object value) {
      if (value == null) return false;
      if (GetType() == value.GetType()) return this == (Digit)value;
      return false;
   }
   public override int GetHashCode() {
      return value.GetHashCode();
   }
   public override string ToString() {
      return value.ToString();
   }
}
class Test
{
   static void Main() {
      Digit a = (Digit) 5;
      Digit b = (Digit) 3;
      Digit plus = a + b;
      Digit minus = a – b;
      bool equals = (a == b);
      Console.WriteLine("{0} + {1} = {2}", a, b, plus);
      Console.WriteLine("{0} - {1} = {2}", a, b, minus);
      Console.WriteLine("{0} == {1} = {2}", a, b, equals);
   }
}
```

The `Digit` type defines the following operators:

- An implicit conversion operator from `Digit` to `byte`.
- An explicit conversion operator from `byte` to `Digit`.

- An addition operator that adds two `Digit` values and returns a `Digit` value.
- A subtraction operator that subtracts one `Digit` value from another, and returns a `Digit` value.
- The equality (`==`) and inequality (`!=`) operators, which compare two `Digit` values.

1.7.7 Indexers

An indexer is a member that enables an object to be indexed in the same way as an array. Whereas properties enable field-like access, indexers enable array-like access.

As an example, consider the `Stack` class presented earlier. The designer of this class might want to expose array-like access so that it is possible to inspect or alter the items on the stack without performing unnecessary `Push` and `Pop` operations. That is, class `Stack` is implemented as a linked list, but it also provides the convenience of array access.

Indexer declarations are similar to property declarations, with the main differences being that indexers are nameless (the "name" used in the declaration is `this`, since `this` is being indexed) and that indexers include indexing parameters. The indexing parameters are provided between square brackets. The example

```
using System;
public class Stack
{
   private Node GetNode(int index) {
      Node temp = first;
      while (index > 0 && temp != null) {
         temp = temp.Next;
         index--;
      }
      if (index < 0 || temp == null)
         throw new Exception("Index out of range.");      return temp;
   }
   public object this[int index] {
      get {
         return GetNode(index).Value;
      }
      set {
         GetNode(index).Value = value;
      }
   }
   ...
}
class Test
{
   static void Main() {
      Stack s = new Stack();
      s.Push(1);
      s.Push(2);
      s.Push(3);
      s[0] = 33;   // Changes the top item from 3 to 33
      s[1] = 22;   // Changes the middle item from 2 to 22
      s[2] = 11;   // Changes the bottom item from 1 to 11
   }
}
```

shows an indexer for the `Stack` class.

1.7.8 Instance constructors

An instance constructor is a member that implements the actions required to initialize an instance of a class.

The example

```
using System;
class Point
{
   public double x, y;
   public Point() {
      this.x = 0;
      this.y = 0;
   }
   public Point(double x, double y) {
      this.x = x;
      this.y = y;
   }
   public static double Distance(Point a, Point b) {
      double xdiff = a.x – b.x;
      double ydiff = a.y – b.y;
      return Math.Sqrt(xdiff * xdiff + ydiff * ydiff);
   }
   public override string ToString() {
      return string.Format("({0}, {1})", x, y);
   }
}
class Test
{
   static void Main() {
      Point a = new Point();
      Point b = new Point(3, 4);
      double d = Point.Distance(a, b);
      Console.WriteLine("Distance from {0} to {1} is {2}", a, b, d);
   }
}
```

shows a `Point` class that provides two public instance constructors, one of which takes no arguments, while the other takes two `double` arguments.

If no instance constructor is supplied for a class, then an empty one with no parameters is automatically provided.

1.7.9 Destructors

A destructor is a member that implements the actions required to destruct an instance of a class. Destructors cannot have parameters, they cannot have accessibility modifiers, and they cannot be called explicitly. The destructor for an instance is called automatically during garbage collection.

The example

```
using System;
class Point
{
   public double x, y;
   public Point(double x, double y) {
      this.x = x;
      this.y = y;
   }
   ~Point() {
      Console.WriteLine("Destructed {0}", this);
   }
   public override string ToString() {
      return string.Format("({0}, {1})", x, y);
   }
}
```

shows a `Point` class with a destructor.

1.7.10 Static constructors

A static constructor is a member that implements the actions required to initialize a class. Static constructors cannot have parameters, they cannot have accessibility modifiers, and they cannot be called explicitly. The static constructor for a class is called automatically.

The example

```
using Personnel.Data;
class Employee
{
   private static DataSet ds;
   static Employee() {
      ds = new DataSet(...);
   }
   public string Name;
   public decimal Salary;
   ...
}
```

shows an `Employee` class with a static constructor that initializes a static field.

1.7.11 Inheritance

Classes support single inheritance, and the type `object` is the ultimate base class for all classes.

The classes shown in earlier examples all implicitly derive from `object`. The example

```
using System;
class A
{
   public void F() { Console.WriteLine("A.F"); }
}
```

shows a class `A` that implicitly derives from `object`. The example

```
using System;
class B: A
{
   public void G() { Console.WriteLine("B.G"); }
}
class Test
{
   static void Main() {
      B b = new B();
      b.F();         // Inherited from A
      b.G();         // Introduced in B
      A a = b;         // Treat a B as an A
      a.F();
   }
}
```

shows a class `B` that derives from `A`. The class `B` inherits `A`'s `F` method, and introduces a `G` method of its own.

Methods, properties, and indexers can be virtual, which means that their implementation can be overridden in derived classes. The example

```
using System;
class A
{
   public virtual void F() { Console.WriteLine("A.F"); }
}
class B: A
{
   public override void F() {
      base.F();
      Console.WriteLine("B.F");
   }
}
class Test
{
   static void Main() {
      B b = new B();
      b.F();
      A a = b;
      a.F();
   }
}
```

shows a class `A` with a virtual method `F`, and a class `B` that overrides `F`. The overriding method in `B` contains a call, `base.F()`, which calls the overridden method in `A`.

A class can indicate that it is incomplete, and is intended only as a base class for other classes, by including the `abstract` modifier. Such a class is called an abstract class. An abstract class can specify abstract members — members that a non-abstract derived class must implement. The example

```
using System;
abstract class A
{
   public abstract void F();
}
class B: A
{
   public override void F() { Console.WriteLine("B.F"); }
}
class Test
{
   static void Main() {
      B b = new B();
      b.F();
      A a = b;
      a.F();
   }
}
```

introduces an abstract method `F` in the abstract class `A`. The non-abstract class `B` provides an implementation for this method.

1.8 Structs

The list of similarities between classes and structs is long — structs can implement interfaces, and can have the same kinds of members as classes. Structs differ from classes in several important ways, however: structs are value types rather than reference types, and inheritance is not supported for structs. Struct values are stored "on the stack" or "in-line". Careful programmers can sometimes enhance performance through judicious use of structs.

For example, the use of a struct rather than a class for a `Point` can make a large difference in the number of memory allocations performed at run time. The program below creates and initializes an array of 100 points. With `Point` implemented as a class, 101 separate objects are instantiated — one for the array and one each for the 100 elements.

```
class Point
{
   public int x, y;
   public Point(int x, int y) {
      this.x = x;
      this.y = y;
   }
}
class Test
{
   static void Main() {
      Point[] points = new Point[100];
      for (int i = 0; i < 100; i++)
         points[i] = new Point(i, i*i);
   }
}
```

If `Point` is instead implemented as a struct, as in

```
struct Point
{
   public int x, y;
   public Point(int x, int y) {
      this.x = x;
      this.y = y;
   }
}
```

only one object is instantiated — the one for the array. The `Point` instances are allocated in-line within the array. This optimization can be misused. Using structs instead of classes can also make an application run slower, or take up more memory, as passing a struct instance by value causes a copy of that struct to be created.

1.9 Interfaces

An interface defines a contract. A class or struct that implements an interface must adhere to its contract. Interfaces can contain methods, properties, events, and indexers as members.

The example

```
interface IExample
{
   string this[int index] { get; set; }
   event EventHandler E;
   void F(int value);
   string P { get; set; }
}
public delegate void EventHandler(object sender, EventArgs e);
```

shows an interface that contains an indexer, an event `E`, a method `F`, and a property `P`.

Interfaces may employ multiple inheritance. In the example

```
interface IControl
{
   void Paint();
}
interface ITextBox: IControl
{
   void SetText(string text);
}
interface IListBox: IControl
{
   void SetItems(string[] items);
}
interface IComboBox: ITextBox, IListBox {}
```

the interface `IComboBox` inherits from both `ITextBox` and `IListBox`.

Classes and structs can implement multiple interfaces. In the example

```
interface IDataBound
{
   void Bind(Binder b);
}
public class EditBox: Control, IControl, IDataBound
{
   public void Paint() {...}
   public void Bind(Binder b) {...}
}
```

the class `EditBox` derives from the class `Control` and implements both `IControl` and `IDataBound`.

In the previous example, the `Paint` method from the `IControl` interface and the `Bind` method from `IDataBound` interface are implemented using public members on the `EditBox` class. C# provides an alternative way of implementing these methods that allows the implementing class to avoid having these members be public. Interface members can be implemented using a qualified name. For example, the `EditBox` class could instead be implemented by providing `IControl.Paint` and `IDataBound.Bind` methods.

```
public class EditBox: IControl, IDataBound
{
   void IControl.Paint() {...}
   void IDataBound.Bind(Binder b) {...}
}
```

Interface members implemented in this way are called explicit interface members because each member explicitly designates the

interface member being implemented. Explicit interface members can only be called via the interface. For example, the `EditBox`'s implementation of the `Paint` method can be called only by casting to the `IControl` interface.

```
class Test
{
   static void Main() {
      EditBox editbox = new EditBox();
      editbox.Paint();   // error: no such method
      IControl control = editbox;
      control.Paint();   // calls EditBox's Paint implementation
   }
}
```

1.10 Delegates

Delegates enable scenarios that some other languages have addressed with function pointers. However, unlike function pointers, delegates are object-oriented and type-safe.

A delegate declaration defines a class that is derived from the class `System.Delegate`. A delegate instance encapsulates an invocation list, which is a list of one or more methods, each of which is referred to as a callable entity. For instance methods, a callable entity consists of an instance and a method on that instance. For static methods, a callable entity consists of just a method. Invoking a delegate instance with an appropriate set of arguments causes each of the delegate instance's callable entities to be invoked with the given set of arguments.

An interesting and useful property of a delegate instance is that it does not know or care about the classes of the methods it encapsulates; all that matters is that those methods be compatible (Section 15.1) with the delegate's type. This makes delegates perfectly suited for "anonymous" invocation. This is a powerful capability.

There are three steps in defining and using delegates: declaration, instantiation, and invocation. Delegates are declared using delegate declaration syntax. The example

```
delegate void SimpleDelegate();
```

declares a delegate named `SimpleDelegate` that takes no arguments and returns no result.

The example

```
class Test
{
   static void F() {
      System.Console.WriteLine("Test.F");
   }
   static void Main() {
      SimpleDelegate d = new SimpleDelegate(F);
      d();
   }
}
```

creates a `SimpleDelegate` instance and then immediately calls it.

There is not much point in instantiating a delegate for a method and then immediately calling that method via the delegate, as it would be simpler to call the method directly. Delegates really show their usefulness when their anonymity is used. The example

```
void MultiCall(SimpleDelegate d, int count) {
   for (int i = 0; i < count; i++) {
      d();
   }
}
```

shows a `MultiCall` method that repeatedly calls a `SimpleDelegate`. The `MultiCall` method does not know or care about the type of the target method for the `SimpleDelegate`, what accessibility that method has, or whether or not that method is static. All that matters is that the target method is compatible (Section 15.1) with `SimpleDelegate`.

1.11 Enums

An enum type declaration defines a type name for a related group of symbolic constants. Enums are used for "multiple choice" scenarios, in which a runtime decision is made from a fixed number of choices that are known at compile-time.

The example

```
enum Color
{
   Red,
   Blue,
   Green
}
class Shape
{
   public void Fill(Color color) {
      switch(color) {
         case Color.Red:
            ...
            break;
         case Color.Blue:
            ...
            break;
         case Color.Green:
            ...
            break;
         default:
            break;
      }
   }
}
```

shows a `Color` enum and a method that uses this enum. The signature of the `Fill` method makes it clear that the shape can be filled with one of the given colors.

The use of enums is superior to the use of integer constants — as is common in languages without enums — because the use of enums makes the code more readable and self-documenting. The self-documenting nature of the code also makes it possible for the development tool to assist with code writing and other "designer" activities. For example, the use of `Color` rather than `int` for a parameter type enables smart code editors to suggest `Color` values.

1.12 Namespaces and assemblies

The programs presented so far have stood on their own except for dependence on a few system-provided classes such as `System.Console`. It is far more common, however, for real-world applications to consist of several different pieces, each compiled separately. For example, a corporate application might depend on several different components, including some developed internally and some purchased from independent software vendors.

Namespaces and assemblies enable this component-based system. Namespaces provide a logical organizational system. Namespaces are used both as an "internal" organization system for a program, and as an "external" organization system — a way of presenting program elements that are exposed to other programs.

Assemblies are used for physical packaging and deployment. An assembly may contain types, the executable code used to implement these types, and references to other assemblies.

There are two main kinds of assemblies: applications and libraries. Applications have a main entry point and usually have a file extension of `.exe`; libraries do not have a main entry point, and usually have a file extension of `.dll`.

To demonstrate the use of namespaces and assemblies, this section revisits the "hello, world" program presented earlier, and splits it into two pieces: a class library that provides messages and a console application that displays them.

The class library will contain a single class named `HelloMessage`. The example

```
// HelloLibrary.cs
namespace Microsoft.CSharp.Introduction
{
   public class HelloMessage
   {
      public string Message {
         get {
            return "hello, world";
         }
      }
   }
}
```

shows the `HelloMessage` class in a namespace named `Microsoft.CSharp.Introduction`. The `HelloMessage` class provides a read-only property named `Message`. Namespaces can nest, and the declaration

```
namespace Microsoft.CSharp.Introduction
{...}
```

is shorthand for several levels of namespace nesting:

```
namespace Microsoft
{
   namespace CSharp
   {
      namespace Introduction
      {...}
   }
}
```

The next step in the componentization of "hello, world" is to write a console application that uses the `HelloMessage` class. The fully qualified name for the class — `Microsoft.CSharp.Introduction.HelloMessage` — could be used, but this name is quite long and unwieldy. An easier way is to use a "using namespace directive," which makes it possible to use all of the types in a namespace without qualification. The example

```
// HelloApp.cs
using Microsoft.CSharp.Introduction;
class HelloApp
{
   static void Main() {
      HelloMessage m = new HelloMessage();
```

```
        System.Console.WriteLine(m.Message);
    }
}
```

shows a using namespace directive that refers to the `Microsoft.CSharp.Introduction` namespace. The occurrences of `HelloMessage` are shorthand for `Microsoft.CSharp.Introduction.HelloMessage`.

C# also enables the definition and use of aliases. A using alias directive defines an alias for a type. Such aliases can be useful in situation in which name collisions occur between two class libraries, or when a small number of types from a much larger namespace are being used. The example

```
using MessageSource = Microsoft.CSharp.Introduction.HelloMessage;
```

shows a using alias directive that defines `MessageSource` as an alias for the `HelloMessage` class.

The code we have written can be compiled into a class library containing the class `HelloMessage` and an application containing the class `HelloApp`. The details of this compilation step might differ based on the compiler or tool being used. Using the command-line compiler provided in Visual Studio .NET, the command-line invocations are

```
csc /target:library HelloLibrary.cs
csc /reference:HelloLibrary.dll HelloApp.cs
```

which produce a class library named `HelloLibrary.dll` and an application named `HelloApp.exe`.

1.13 Versioning

Versioning is the process of evolving a component over time in a compatible manner. A new version of a component is source compatible with a previous version if code that depends on the previous version can, when recompiled, work with the new version. In contrast, a new version of a component is binary compatible if an application that depended on the old version can, without recompilation, work with the new version.

Most languages do not support binary compatibility at all, and many do little to facilitate source compatibility. In fact, some languages contain flaws that make it impossible, in general, to evolve a class over time without breaking at least some client code.

As an example, consider the situation of a base class author who ships a class named `Base`. In the first version, `Base` contains no `F` method. A component named `Derived` derives from `Base`, and introduces an `F`. This `Derived` class, along with the class `Base` on which it depends, is released to customers, who deploy to numerous clients and servers.

```
// Author A
namespace A
{
   public class Base      // version 1
   {
   }
}
// Author B
namespace B
{
   class Derived: A.Base
   {
      public virtual void F() {
         System.Console.WriteLine("Derived.F");
      }
   }
}
```

From this point, the versioning trouble begins. The author of `Base` produces a new version, giving it its own `F` method.

```
// Author A
namespace A
{
   public class Base      // version 2
   {
      public virtual void F() {      // added in version 2
         System.Console.WriteLine("Base.F");
      }
   }
}
```

This new version of `Base` should be both source and binary compatible with the initial version. (If it weren't possible to simply add a method then a base class could never evolve.) Unfortunately, the new `F` in `Base` makes the meaning of `Derived`'s `F` unclear. Did `Derived` mean to override `Base`'s `F`? This seems unlikely, since when `Derived` was compiled, `Base` did not even have an `F`! Further, if `Derived`'s `F` does override `Base`'s `F`, then it must adhere to the contract specified by `Base` — a contract that was unspecified when `Derived` was written! In some cases, this is impossible. For example, `Base`'s `F` might require that overrides of it always call the base. `Derived`'s `F` could not possibly adhere to such a contract.

C# addresses this versioning problem by requiring developers to state their intent clearly . In the original code example, the code was clear, since `Base` did not even have an `F`. Clearly, `Derived`'s `F` is intended as a new method rather than an override of a base method, since no base method named `F` exists.

If `Base` adds an `F` and ships a new version, then the intent of a binary version of `Derived` is still clear — `Derived`'s `F` is semantically unrelated, and should not be treated as an override.

However, when `Derived` is recompiled, the meaning is unclear — the author of `Derived` may intend its `F` to override `Base`'s `F`, or to hide it. Since the intent is unclear, the compiler produces a warning, and by default makes `Derived`'s `F` hide `Base`'s `F`. This course of action duplicates the semantics for the case in which `Derived` is not recompiled. The warning that is generated alerts `Derived`'s author to the presence of the `F` method in `Base`.

If `Derived`'s `F` is semantically unrelated to `Base`'s `F`, then `Derived`'s author can express this intent — and, in effect, turn off the warning–by using the `new` keyword in the declaration of `F`.

```
// Author A
namespace A
{
   public class Base         // version 2
   {
      public virtual void F() { // added in version 2
         System.Console.WriteLine("Base.F");
      }
   }
}
// Author B
namespace B
{
   class Derived: A.Base   // version 2a: new
   {
      new public virtual void F() {
         System.Console.WriteLine("Derived.F");
      }
   }
}
```

On the other hand, `Derived`'s author might investigate further, and decide that `Derived`'s `F` should override `Base`'s `F`. This intent can be specified by using the `override` keyword, as shown below.

```
// Author A
namespace A
{
   public class Base               // version 2
   {
      public virtual void F() { // added in version 2
         System.Console.WriteLine("Base.F");
      }
   }
}
// Author B
namespace B
{
   class Derived: A.Base   // version 2b: override
   {
      public override void F() {
         base.F();
         System.Console.WriteLine("Derived.F");
      }
   }
}
```

The author of `Derived` has one other option, and that is to change the name of `F`, thus completely avoiding the name collision. Although this change would break source and binary compatibility for `Derived`, the importance of this compatibility varies depending on the scenario. If `Derived` is not exposed to other programs, then changing the name of `F` is likely a good idea, as it would improve the readability of the program — there would no longer be any confusion about the meaning of `F`.

1.14 Attributes

C# is an imperative language, but like all imperative languages it does have some declarative elements. For example, the accessibility of a method in a class is specified by declaring it `public`, `protected`, `internal`, `protected internal`, or `private`. C# generalizes this capability, so that programmers can invent new kinds of declarative information, attach this declarative information to various program entities, and retrieve this declarative information at run-time. Programs specify this additional declarative information by defining and using attributes (Section 17).

For instance, a framework might define a `HelpAttribute` attribute that can be placed on program elements such as classes and methods, enabling developers to provide a mapping from program elements to documentation for them. The example

```
using System;
[AttributeUsage(AttributeTargets.All)]
public class HelpAttribute: Attribute
{
   public HelpAttribute(string url) {
      this.url = url;
   }
   public string Topic = null;
   private string url;
   public string Url {
      get { return url; }
   }
}
```

defines an attribute class named `HelpAttribute` that has one positional parameter (`string url`) and one named parameter (`string Topic`). As explained in Section 17.1, this attribute may be referenced by its full name, `HelpAttribute`, or by its implicit short name, `Help`. Positional parameters are defined by the formal parameters for public instance constructors of the attribute class, and named parameters are defined by public non-static read-write fields and properties of the attribute class.

The example

```
[Help("http://www.microsoft.com/.../Class1.htm")]
public class Class1
{
   [Help("http://www.microsoft.com/.../Class1.htm", Topic = "F")]
   public void F() {}
}
```

shows several uses of the `Help` attribute.

Attribute information for a given program element can be retrieved at run time by using reflection support. The example

```
using System;
class Test
{
   static void Main() {
      Type type = typeof(Class1);
      object[] arr = type.GetCustomAttributes(typeof(HelpAttribute), true);
      if (arr.Length == 0)
         Console.WriteLine("Class1 has no Help attribute.");
      else {
         HelpAttribute ha = (HelpAttribute) arr[0];
         Console.WriteLine("Url = {0}, Topic = {1}", ha.Url, ha.Topic);
      }
   }
}
```

checks to see if `Class1` has a `Help` attribute, and writes out the associated `Topic` and `Url` values if the attribute is present.

2. Lexical structure

This chapter defines the lexical structure of C# programs.

2.1 Programs

A C# program consists of one or more source files, known formally as compilation units (Section 9.1). A source file is an ordered sequence of Unicode characters. Source files typically have a one-to-one correspondence with files in a file system, but this correspondence is not required. For maximal portability, it is recommended that files in a file system be encoded with the UTF-8 encoding.

Conceptually speaking, a program is compiled using three steps:

- Transformation, which converts a file from a particular character repertoire and encoding scheme into a sequence of Unicode characters.
- Lexical analysis, which translates a stream of Unicode input characters into a stream of tokens.
- Syntactic analysis, which translates the stream of tokens into executable code.

2.2 Grammars

This specification presents the syntax of the C# programming language using two grammars. The lexical grammar (Section 2.2.2) defines how Unicode characters are combined to form line terminators, white space, comments, tokens, and pre-processing directives. The syntactic grammar (Section 2.2.3) defines how the tokens resulting from the lexical grammar are combined to form C# programs.

2.2.1 Grammar notation

The lexical and syntactic grammars are presented using grammar productions. Each grammar production defines a non-terminal symbol and the possible expansions of that non-terminal symbol into sequences of non-terminal or terminal symbols. In grammar productions, non-terminal symbols are shown in italic type, and terminal symbols are shown in a fixed-width font.

The first line of a grammar production is the name of the non-terminal symbol being defined, followed by a colon. Each successive indented line contains a possible expansion of the non-terminal given as a sequence of non-terminal or terminal symbols. For example, the production:

while-statement:
  while (*boolean*-expression) *embedded-statement*

defines a *while-statement* to consist of the token `while`, followed by the token "`(`", followed by a *boolean-expression*, followed by the token "`)`", followed by an *embedded-statement*.

When there is more than one possible expansion of a non-terminal symbol, the alternatives are listed on separate lines. For example, the production:

statement-list:
  statement
  statement-list *statement*

defines a *statement-list* to either consist of a *statement* or consist of a *statement-list* followed by a *statement*. In other words, the definition is recursive and specifies that a statement list consists of one or more statements.

A subscripted suffix "*opt*" is used to indicate an optional symbol. The production:

block:
  { $statement\text{-}list_{opt}$ }

is shorthand for:

block:
  { }
  { *statement-list* }

and defines a *block* to consist of an optional *statement-list* enclosed in "`{`" and "`}`" tokens.

Alternatives are normally listed on separate lines, though in cases where there are many alternatives, the phrase "one of" may precede a list of expansions given on a single line. This is simply shorthand for listing each of the alternatives on a separate line. For example, the production:

real-type-suffix: one of
  F f D d M m

is shorthand for:

real-type-suffix:
  F
  f
  D
  d
  M
  m

2.2.2 Lexical grammar

The lexical grammar of C# is presented in Section 2.3, Section 2.4, and Section 2.5. The terminal symbols of the lexical grammar are the characters of the Unicode character set, and the lexical grammar specifies how characters are combined to form tokens (Section 2.4), white space (Section 2.3.2), comments (Section 2.3.3), and pre-processing directives (Section 2.5).

Every source file in a C# program must conform to the input production of the lexical grammar (Section 2.3).

2.2.3 Syntactic grammar

The syntactic grammar of C# is presented in the chapters and appendices that follow this chapter. The terminal symbols of the syntactic grammar are the tokens defined by the lexical grammar, and the syntactic grammar specifies how tokens are combined to form C# programs.

Every source file in a C# program must conform to the compilation-unit production of the syntactic grammar (Section 9.1).

2.3 Lexical analysis

The input production defines the lexical structure of a C# source file. Each source file in a C# program must conform to this lexical grammar production.

input:
  input-section$_{opt}$
input-section:
  input-section-part
  input-section *input-section-part*
input-section-part:
  input-elements$_{opt}$ *new-line*
  pp-directive
input-elements:
  input-element
  input-elements *input-element*
input-element:
  whitespace
  comment
  token

Five basic elements make up the lexical structure of a C# source file: Line terminators (Section 2.3.1), white space (Section 2.3.2), comments (Section 2.3.3), tokens (Section 2.4), and pre-processing directives (Section 2.5). Of these basic elements, only tokens are significant in the syntactic grammar of a C# program (Section 2.2.3).

The lexical processing of a C# source file consists of reducing the file into a sequence of tokens which becomes the input to the syntactic analysis. Line terminators, white space, and comments can serve to separate tokens, and pre-processing directives can cause sections of the source file to be skipped, but otherwise these lexical elements have no impact on the syntactic structure of a C# program.

When several lexical grammar productions match a sequence of characters in a source file, the lexical processing always forms the longest possible lexical element. For example, the character sequence `//` is processed as the beginning of a single-line comment because that lexical element is longer than a single `/` token.

2.3.1 Line terminators

Line terminators divide the characters of a C# source file into lines.

new-line:
  Carriage return character (U+000D)
  Line feed character (U+000A)
  Carriage return character (U+000D) followed by line feed character (U+000A)
  Line separator character (U+2028)
  Paragraph separator character (U+2029)

For compatibility with source code editing tools that add end-of-file markers, and to enable a source file to be viewed as a sequence of properly terminated lines, the following transformations are applied, in order, to every source file in a C# program:

- If the last character of the source file is a Control-Z character (`U+001A`), this character is deleted.
- A carriage-return character (`U+000D`) is added to the end of the source file if that source file is non-empty and if the last character of the source file is not a carriage return (`U+000D`), a line feed (`U+000A`), a line separator (`U+2028`), or a paragraph separator (`U+2029`).

2.3.2 White space

White space is defined as any character with Unicode class Zs (which includes the space character) as well as the horizontal tab character, the vertical tab character, and the form feed character.

whitespace:
  Any character with Unicode class Zs
  Horizontal tab character (U+0009)
  Vertical tab character (U+000B)
  Form feed character (U+000C)

2.3.3 Comments

Two forms of comments are supported: single-line comments and delimited comments. *Single-line comments* start with the characters `//` and extend to the end of the source line. *Delimited comments* start with the characters `/*` and end with the characters `*/`. Delimited comments may span multiple lines.

comment:
  single-line-comment
  delimited-comment
single-line-comment:
  // *input-characters*$_{opt}$
input-characters:
  input-character
  input-characters *input-character*
input-character:
  Any Unicode character except a *new-line-character*
new-line-character:
  Carriage return character (U+000D)
  Line feed character (U+000A)
  Line separator character (U+2028)
  Paragraph separator character (U+2029)
delimited-comment:
  */** *delimited-comment-characters*$_{opt}$ **/*
delimited-comment-characters:
  delimited-comment-character
  delimited-comment-characters *delimited-comment-character*
delimited-comment-character:
  not-asterisk
  *** *not-slash*
not-asterisk:
  Any Unicode character except *
not-slash:
  Any Unicode character except /

Comments do not nest. The character sequences `/*` and `*/` have no special meaning within a `//` comment, and the character sequences `//` and `/*` have no special meaning within a delimited comment.

Comments are not processed within character and string literals.

The example

```
/* Hello, world program
      This program writes "hello, world" to the console
*/
class Hello
{
   static void Main() {
      System.Console.WriteLine("hello, world");
   }
}
```

includes a delimited comment.

The example

```
// Hello, world program
// This program writes "hello, world" to the console
//
class Hello // any name will do for this class
{
   static void Main() { // this method must be named "Main"
      System.Console.WriteLine("hello, world");
   }
```

```
}
```

shows several single-line comments.

2.4 Tokens

There are several kinds of tokens: identifiers, keywords, literals, operators, and punctuators. White space and comments are not tokens, though they act as separators for tokens.

token:
  identifier
  keyword
  integer-literal
  real-literal
  character-literal
  string-literal
  operator-or-punctuator

2.4.1 Unicode character escape sequences

A Unicode character escape sequence represents a Unicode character. Unicode character escape sequences are processed in identifiers (Section 2.4.2), character literals (Section 2.4.4.4), and regular string literals (Section 2.4.4.5). A Unicode character escape is not processed in any other location (for example, to form an operator, punctuator, or keyword).

unicode-escape-sequence:
  \u *hex-digit* *hex-digit* *hex-digit* *hex-digit*
  \U *hex-digit* *hex-digit* *hex-digit* *hex-digit* *hex-digit* *hex-digit* *hex-digit* *hex-digit*

A Unicode escape sequence represents the single Unicode character formed by the hexadecimal number following the "`\u`" or "`\U`" characters. Since C# uses a 16-bit encoding of Unicode code points in characters and string values, a Unicode character in the range U+10000 to U+10FFFF is not permitted in a character literal and is represented using a Unicode surrogate pair in a string literal. Unicode characters with code points above 0x10FFFF are not supported.

Multiple translations are not performed. For instance, the string literal "`\u005Cu005C`" is equivalent to "`\u005C`" rather than "`\`". The Unicode value `\u005C` is the character "\".

The example

```
class Class1
{
   static void Test(bool \u0066) {
      char c = '\u0066';
      if (\u0066)
         System.Console.WriteLine(c.ToString());
   }
}
```

shows several uses of `\u0066`, which is the escape sequence for the letter "`f`". The program is equivalent to

```
class Class1
{
   static void Test(bool f) {
      char c = 'f';
      if (f)
         System.Console.WriteLine(c.ToString());
   }
}
```

2.4.2 Identifiers

The rules for identifiers given in this section correspond exactly to those recommended by the Unicode Standard Annex 15, except that underscore is allowed as an initial character (as is traditional in the C programming language), Unicode escape sequences are permitted in identifiers, and the "@" character is allowed as a prefix to enable keywords to be used as identifiers.

identifier:
  available-identifier
  @ *identifier-or-keyword*
available-identifier:
  An *identifier-or-keyword* that is not a *keyword*
identifier-or-keyword:
  identifier-start-character *identifier-part-characters*$_{opt}$
identifier-start-character:
  letter-character
  _ (the underscore character `U+005F`)
identifier-part-characters:
  identifier-part-character
  identifier-part-characters *identifier-part-character*
identifier-part-character:
  letter-character
  decimal-digit-character
  connecting-character
  combining-character
  formatting-character
letter-character:
  A Unicode character of classes Lu, Ll, Lt, Lm, Lo, or Nl
  A *unicode-escape-sequence* representing a character of classes Lu, Ll, Lt, Lm, Lo, or Nl
combining-character:
  A Unicode character of classes Mn or Mc
  A *unicode-escape-sequence* representing a character of classes Mn or Mc
decimal-digit-character:
  A Unicode character of the class Nd
  A *unicode-escape-sequence* representing a character of the class Nd
connecting-character:
  A Unicode character of the class Pc
  A *unicode-escape-sequence* representing a character of the class Pc
formatting-character:
  A Unicode character of the class Cf
  A *unicode-escape-sequence* representing a character of the class Cf

For information on the Unicode character classes mentioned above, see *The Unicode Standard, Version 3.0,* section 4.5.

Examples of valid identifiers include "`identifier1`", "`_identifier2`", and "`@if`".

An identifier in a conforming program must be in the canonical format defined by Unicode Normalization Form C, as defined by Unicode Standard Annex 15. The behavior when encountering an identifier not in Normalization Form C is implementation-defined; however, a diagnostic is not required.

The prefix "`@`" enables the use of keywords as identifiers, which is useful when interfacing with other programming languages. The character `@` is not actually part of the identifier, so the identifier might be seen in other languages as a normal identifier, without the prefix. An identifier with an `@` prefix is called a verbatim identifier. Use of the `@` prefix for identifiers that are not keywords is permitted, but strongly discouraged as a matter of style.

The example:

```
class @class
{
   public static void @static(bool @bool) {
      if (@bool)
         System.Console.WriteLine("true");
      else
         System.Console.WriteLine("false");
   }
```

```
}
class Class1
{
   static void M() {
      cl\u0061ss.st\u0061tic(true);
   }
}
```

defines a class named "`class`" with a static method named "`static`" that takes a parameter named "`bool`". Note that since Unicode escapes are not permitted in keywords, the token "`cl\u0061ss`" is an identifier, and is the same identifier as "`@class`".

Two identifiers are considered the same if they are identical after the following transformations are applied, in order:

- The prefix "`@`", if used, is removed.
- Each *unicode-escape-sequence* is transformed into its corresponding Unicode character.
- Any *formatting-characters* are removed.

Identifiers containing two consecutive underscore characters (`U+005F`) are reserved for use by the implementation. For example, an implementation might provide extended keywords that begin with two underscores.

2.4.3 Keywords

A keyword is an identifier-like sequence of characters that is reserved, and cannot be used as an identifier except when prefaced by the `@` character.

keyword: one of

abstract	as	base	bool	break	byte	case
catch	char	checked	class	const	continue	decimal
default	delegate	do	double	else	enum	event
explicit	extern	false	finally	fixed	float	for
foreach	goto	if	implicit	in	int	interface
internal	is	lock	long	namespace	new	null
object	operator	out	override	params	private	protected
public	readonly	ref	return	sbyte	sealed	short
sizeof	stackalloc	static	string	struct	switch	this
throw	true	try	typeof	uint	ulong	unchecked
unsafe	ushort	using	virtual	void	volatile	while

In some places in the grammar, specific identifiers have special meaning, but are not keywords. For example, within a property declaration, the "`get`" and "`set`" identifiers have special meaning (Section 10.6.2). An identifier other than `get` or `set` is never permitted in these locations, so this use does not conflict with a use of these words as identifiers.

2.4.4 Literals

A literal is a source code representation of a value.

literal:
  boolean-literal
  integer-literal
  real-literal
  character-literal
  string-literal
  null-literal

2.4.4.1 Boolean literals

There are two Boolean literal values: `true` and `false`.

boolean-literal:
  true
  false

The type of a *boolean-literal* is `bool`.

2.4.4.2 Integer literals

Integer literals are used to write values of types `int`, `uint`, `long`, and `ulong`. Integer literals have two possible forms: decimal and hexadecimal.

integer-literal:
  decimal-integer-literal
  hexadecimal-integer-literal
decimal-integer-literal:
  decimal-digits *integer-type-suffix*$_{opt}$
decimal-digits:
  decimal-digit
  decimal-digits *decimal-digit*
decimal-digit: one of
  0 1 2 3 4 5 6 7 8 9
integer-type-suffix: one of
  U u L l UL Ul uL ul LU Lu lU lu
hexadecimal-integer-literal:
  0x *hex-digits* *integer-type-suffix*$_{opt}$
  0X *hex-digits* *integer-type-suffix*$_{opt}$
hex-digits:
  hex-digit
  hex-digits *hex-digit*
hex-digit: one of
  0 1 2 3 4 5 6 7 8 9 A B C D E F a b c d e f

The type of an integer literal is determined as follows:

- If the literal has no suffix, it has the first of these types in which its value can be represented: `int`, `uint`, `long`, `ulong`.
- If the literal is suffixed by `U` or `u`, it has the first of these types in which its value can be represented: `uint`, `ulong`.
- If the literal is suffixed by `L` or `l`, it has the first of these types in which its value can be represented: `long`, `ulong`.
- If the literal is suffixed by `UL`, `Ul`, `uL`, `ul`, `LU`, `Lu`, `lU`, or `lu`, it is of type `ulong`.

If the value represented by an integer literal is outside the range of the `ulong` type, a compile-time error occurs.

As a matter of style, it is suggested that "`L`" be used instead of "`l`" when writing literals of type `long`, since it is easy to confuse the letter "`l`" with the digit "`1`".

To permit the smallest possible `int` and `long` values to be written as decimal integer literals, the following two rules exist:

- When a *decimal-integer-literal* with the value 2147483648 (2^{31}) and no *integer-type-suffix* appears as the token immediately following a unary minus operator token (Section 7.6.2), the result is a constant of type `int` with the value −2147483648 (-2^{31}). In all other situations, such a *decimal-integer-literal* is of type `uint`.
- When a *decimal-integer-literal* with the value 9223372036854775808 (2^{63}) and no *integer-type-suffix* or the *integer-type-suffix* `L` or `l` appears as the token immediately following a unary minus operator token (Section 7.6.2), the result is a constant of type `long` with the value −9223372036854775808 (-2^{63}). In all other situations, such a *decimal-integer-literal* is of type `ulong`.

2.4.4.3 Real literals

Real literals are used to write values of types `float`, `double`, and `decimal`.

real-literal:
  decimal-digits . *decimal-digits* *exponent-part*$_{opt}$ *real-type-suffix*$_{opt}$
  . *decimal-digits* *exponent-part*$_{opt}$ *real-type-suffix*$_{opt}$
  decimal-digits *exponent-part* *real-type-suffix*$_{opt}$
  decimal-digits *real-type-suffix*
exponent-part:
  e *sign*$_{opt}$ *decimal-digits*
  E *sign*$_{opt}$ *decimal-digits*
sign: one of
  + -
real-type-suffix: one of
  F f D d M m

If no real type suffix is specified, the type of the real literal is `double`. Otherwise, the real type suffix determines the type of the real literal, as follows:

- A real literal suffixed by `F` or `f` is of type `float`. For example, the literals `1f`, `1.5f`, `1e10f`, and `123.456F` are all of type `float`.
- A real literal suffixed by `D` or `d` is of type `double`. For example, the literals `1d`, `1.5d`, `1e10d`, and `123.456D` are all of type `double`.
- A real literal suffixed by `M` or `m` is of type `decimal`. For example, the literals `1m`, `1.5m`, `1e10m`, and `123.456M` are all of type `decimal`. This literal is converted to a `decimal` value by taking the exact value, and, if necessary, rounding to the nearest representable value using banker's rounding (Section 4.1.7). Any scale apparent in the literal is preserved unless the value is rounded or the value is zero (in which latter case the sign and scale will be 0). Hence, the literal `2.900m` will be parsed to form the decimal with sign `0`, coefficient `2900`, and scale `3`.

If the specified literal cannot be represented in the indicated type, then a compile-time error occurs.

The value of a real literal of type `float` or `double` is determined by using the IEEE "round to nearest" mode.

Note that in a real literal, decimal digits are always required after the decimal point. For example, 1.3F is a real literal but 1.F is not.

2.4.4.4 Character literals

A character literal represents a single character, and usually consists of a character in quotes, as in `'a'`.

character-literal:
  ' *character* '
character:
  single-character
  simple-escape-sequence
  hexadecimal-escape-sequence
  unicode-escape-sequence
single-character:
  Any character except ' (U+0027), \ (U+005C), and *new-line-character*
simple-escape-sequence: one of
  \' \" \\ \0 \a \b \f \n \r \t \v
hexadecimal-escape-sequence:
  \x *hex-digit* $\textit{hex-digit}_{opt}$ $\textit{hex-digit}_{opt}$ $\textit{hex-digit}_{opt}$

A character that follows a backslash character (`\`) in a *character* must be one of the following characters: `'`, `"`, `\`, `0`, `a`, `b`, `f`, `n`, `r`, `t`, `u`, `U`, `x`, `v`. Otherwise, a compile-time error occurs.

A hexadecimal escape sequence represents a single Unicode character, with the value formed by the hexadecimal number following "\x".

If the value represented by a character literal is greater than `U+FFFF`, a compile-time error occurs.

A Unicode character escape sequence (Section 2.4.1) in a character literal must be in the range `U+0000` to `U+FFFF`.

A simple escape sequence represents a Unicode character encoding, as described in the table below.

Escape sequence	Character name	Unicode encoding
`\'`	Single quote	`0x0027`
`\"`	Double quote	`0x0022`
`\\`	Backslash	`0x005C`
`\0`	Null	`0x0000`
`\a`	Alert	`0x0007`
`\b`	Backspace	`0x0008`
`\f`	Form feed	`0x000C`
`\n`	New line	`0x000A`
`\r`	Carriage return	`0x000D`
`\t`	Horizontal tab	`0x0009`

`\v`	Vertical tab	`0x000B`

The type of a *character-literal* is `char`.

2.4.4.5 String literals

C# supports two forms of string literals: regular string literals and verbatim string literals.

A regular string literal consists of zero or more characters enclosed in double quotes, as in `"hello"`, and may include both simple escape sequences (such as `\t` for the tab character) and hexadecimal and Unicode escape sequences.

A verbatim string literal consists of an `@` character followed by a double-quote character, zero or more characters, and a closing double-quote character. A simple example is `@"hello"`. In a verbatim string literal, the characters between the delimiters are interpreted verbatim, the only exception being a *quote-escape-sequence*. In particular, simple escape sequences and hexadecimal and Unicode escape sequences are not processed in verbatim string literals. A verbatim string literal may span multiple lines.

string-literal:
  regular-string-literal
  verbatim-string-literal
regular-string-literal:
  " *regular-string-literal-characters*$_{opt}$ "
regular-string-literal-characters:
  regular-string-literal-character
  regular-string-literal-characters *regular-string-literal-character*
regular-string-literal-character:
  single-regular-string-literal-character
  simple-escape-sequence
  hexadecimal-escape-sequence
  unicode-escape-sequence
single-regular-string-literal-character:
  Any character except " (U+0022), \ (U+005C), and *new-line-character*
verbatim-string-literal:
  @" *verbatim -string-literal-characters*$_{opt}$ "
verbatim-string-literal-characters:
  verbatim-string-literal-character
  verbatim-string-literal-characters *verbatim-string-literal-character*
verbatim-string-literal-character:
  single-verbatim-string-literal-character
  quote-escape-sequence
single-verbatim-string-literal-character:
  Any character except "
quote-escape-sequence:
  ""

A character that follows a backslash character (`\`) in a *regular-string-literal-character* must be one of the following characters: `'`, `"`, `\`, `0`, `a`, `b`, `f`, `n`, `r`, `t`, `u`, `U`, `x`, `v`. Otherwise, a compile-time error occurs.

The example

```
string a = "hello, world";                  // hello, world
string b = @"hello, world";               // hello, world
string c = "hello \t world";               // hello     world
string d = @"hello \t world";               // hello \t world
string e = "Joe said \"Hello\" to me";      // Joe said "Hello" to me
string f = @"Joe said ""Hello"" to me";   // Joe said "Hello" to me
string g = "\\\\server\\share\\file.txt";   // \\server\share\file.txt
string h = @"\\server\share\file.txt";      // \\server\share\file.txt
string i = "one\r\ntwo\r\nthree";
string j = @"one
two
three";
```

shows a variety of string literals. The last string literal, `j`, is a verbatim string literal that spans multiple lines. The characters between the quotation marks, including white space such as new line characters, are preserved verbatim.

Since a hexadecimal escape sequence can have a variable number of hex digits, the string literal `"\x123"` contains a single character with hex value 123. To create a string containing the character with hex value 12 followed by the character 3, you could

write `"\x00123"` or `"\x12" + "3"` instead.

The type of a *string-literal* is `string`.

Each string literal does not necessarily result in a new string instance. When two or more string literals that are equivalent according to the string equality operator (Section 7.9.7) appear in the same assembly, these string literals refer to the same string instance. For instance, the output produced by

```
class Test
{
   static void Main() {
      object a = "hello";
      object b = "hello";
      System.Console.WriteLine(a == b);
   }
}
```

is `True` because the two literals refer to the same string instance.

2.4.4.6 The null literal

null-literal:
  null

The type of a *null-literal* is the null type.

2.4.5 Operators and punctuators

There are several kinds of operators and punctuators. Operators are used in expressions to describe operations involving one or more operands. For example, the expression `a + b` uses the `+` operator to add the two operands `a` and `b`. Punctuators are for grouping and separating.

operator-or-punctuator: one of

```
{   }   [   ]   (   )   .   ,   :   ;

+   -   *   /   %   &   |   ^   !   ~

=   <   >   ?   ++  --  &&  ||  <<  >>

==  !=  <=  >=  +=  -=  *=  /=  %=  &=

|=  ^=  <<=  >>=  ->
```

2.5 Pre-processing directives

The pre-processing directives provide the ability to conditionally skip sections of source files, to report error and warning conditions, and to delineate distinct regions of source code. The term "pre-processing directives" is used only for consistency with the C and C++ programming languages. In C#, there is no separate pre-processing step; pre-processing directives are processed as part of the lexical analysis phase.

pp-directive:
  pp-declaration
  pp-conditional
  pp-line
  pp-diagnostic
  pp-region

The following pre-processing directives are available:

- `#define` and `#undef`, which are used to define and undefine, respectively, conditional compilation symbols (Section 2.5.3).
- `#if`, `#elif`, `#else`, and `#endif`, which are used to conditionally skip sections of source code (Section 2.5.4).
- `#line`, which is used to control line numbers emitted for errors and warnings (Section 2.5.7).
- `#error` and `#warning`, which are used to issue errors and warnings, respectively (Section 2.5.5).
- `#region` and `#endregion`, which are used to explicitly mark sections of source code (Section 2.5.6).

A pre-processing directive always occupies a separate line of source code and always begins with a `#` character and a pre-processing directive name. White space may occur before the `#` character and between the `#` character and the directive name.

A source line containing a `#define`, `#undef`, `#if`, `#elif`, `#else`, `#endif`, or `#line` directive may end with a single-line comment. Delimited comments (the `/* */` style of comments) are not permitted on source lines containing pre-processing directives.

Pre-processing directives are not tokens and are not part of the syntactic grammar of C#. However, pre-processing directives can be used to include or exclude sequences of tokens and can in that way affect the meaning of a C# program. For example, when compiled, the program:

```
#define A
#undef B
class C
{
#if A
   void F() {}
#else
   void G() {}
#endif
#if B
   void H() {}
#else
   void I() {}
#endif
}
```

results in the exact same sequence of tokens as the program:

```
class C
{
   void F() {}
   void I() {}
}
```

Thus, whereas the two programs are quite different lexically, they are identical syntactically.

2.5.1 Conditional compilation symbols

The conditional compilation functionality provided by the `#if`, `#elif`, `#else`, and `#endif` directives is controlled through pre-processing expressions (Section 2.5.1) and conditional compilation symbols.

conditional-symbol:
  Any *identifier-or-keyword* except true or false

A conditional compilation symbol has two possible states: defined or undefined. At the beginning of the lexical processing of a source file, a conditional compilation symbol is undefined unless it has been explicitly defined by an external mechanism (such as a command-line compiler option). When a `#define` directive is processed, the conditional compilation symbol named in that directive becomes defined in that source file. The symbol remains defined until an `#undef` directive for that same symbol is processed, or until the end of the source file is reached. An implication of this is that `#define` and `#undef` directives in one source file have no effect on other source files in the same program.

When referenced in a pre-processing expression, a defined conditional compilation symbol has the Boolean value `true`, and an undefined conditional compilation symbol has the Boolean value `false`. There is no requirement that conditional compilation symbols be explicitly declared before they are referenced in pre-processing expressions. Instead, undeclared symbols are simply undefined and thus have the value `false`.

The name space for conditional compilation symbols is distinct and separate from all other named entities in a C# program. Conditional compilation symbols can only be referenced in `#define` and `#undef` directives and in pre-processing expressions.

2.5.2 Pre-processing expressions

Pre-processing expressions can occur in `#if` and `#elif` directives. The operators `!`, `==`, `!=`, `&&` and `||` are permitted in pre-processing expressions, and parentheses may be used for grouping.

pp-expression:
  whitespace$_{opt}$ *pp-or-expression* *whitespace*$_{opt}$
pp-or-expression:
  pp-and-expression
  pp-or-expression *whitespace*$_{opt}$ || *whitespace*$_{opt}$ *pp-and-expression*
pp-and-expression:
  pp-equality-expression
  pp-and-expression *whitespace*$_{opt}$ && *whitespace*$_{opt}$ *pp-equality-expression*
pp-equality-expression:
  pp-unary-expression
  pp-equality-expression *whitespace*$_{opt}$ == *whitespace*$_{opt}$ *pp-unary-expression*
  pp-equality-expression *whitespace*$_{opt}$!= *whitespace*$_{opt}$ *pp-unary-expression*
pp-unary-expression:
  pp-primary-expression
  ! *whitespace*$_{opt}$ *pp-unary-expression*
pp-primary-expression:
  true
  false
  conditional-symbol
  (*whitespace*$_{opt}$ *pp-expression* *whitespace*$_{opt}$)

When referenced in a pre-processing expression, a defined conditional compilation symbol has the Boolean value `true`, and an undefined conditional compilation symbol has the Boolean value `false`.

Evaluation of a pre-processing expression always yields a Boolean value. The rules of evaluation for a pre-processing expression are the same as those for a constant expression (Section 7.15), except that the only user-defined entities that can be referenced are conditional compilation symbols.

2.5.3 Declaration directives

The declaration directives are used to define or undefine conditional compilation symbols.

pp-declaration:
  whitespace$_{opt}$ # *whitespace*$_{opt}$ define *whitespace* *conditional-symbol* *pp-new-line*
  whitespace$_{opt}$ # *whitespace*$_{opt}$ undef *whitespace* *conditional-symbol* *pp-new-line*
pp-new-line:
  whitespace$_{opt}$ *single-line-comment*$_{opt}$ *new-line*

The processing of a `#define` directive causes the given conditional compilation symbol to become defined, starting with the source line that follows the directive. Likewise, the processing of an `#undef` directive causes the given conditional compilation symbol to become undefined, starting with the source line that follows the directive.

Any `#define` and `#undef` directives in a source file must occur before the first *token* (Section 2.4) in the source file; otherwise, a compile-time error occurs. In intuitive terms, `#define` and `#undef` directives must precede any "real code" in the source file.

The example:

```
#define Enterprise
#if Professional || Enterprise
   #define Advanced
#endif
namespace Megacorp.Data
{
   #if Advanced
   class PivotTable {...}
   #endif
}
```

is valid because the `#define` directives precede the first token (the `namespace` keyword) in the source file.

The following example results in a compile-time error because a `#define` directive follows real code:

```
#define A
namespace N
{
   #define B
   #if B
   class Class1 {}
   #endif
}
```

A `#define` directive may define a conditional compilation symbol that is already defined, without there being any intervening `#undef for that` symbol. The example below defines a conditional compilation symbol `A` and then defines it again.

```
#define A
#define A
```

An `#undef` directive may undefine a conditional compilation symbol that is not defined. The example below defines a conditional compilation symbol `A` and then undefines it twice; although the second `#undef` has no effect, it is still valid.

```
#define A
#undef A
#undef A
```

2.5.4 Conditional compilation directives

The conditional compilation directives are used to conditionally include or exclude portions of a source file.

pp-conditional:
  pp-if-section *pp-elif-sections*$_{opt}$ *pp-else-section*$_{opt}$ *pp-endif*
pp-if-section:
  whitespace$_{opt}$ # *whitespace*$_{opt}$ if *whitespace* *pp-expression* *pp-new-line* *conditional-section*$_{opt}$
pp-elif-sections:
  pp-elif-section
  pp-elif-sections *pp-elif-section*
pp-elif-section:
  whitespace$_{opt}$ # *whitespace*$_{opt}$ elif *whitespace* *pp-expression* *pp-new-line* *conditional-section*$_{opt}$
pp-else-section:
  whitespace$_{opt}$ # *whitespace*$_{opt}$ else *pp-new-line* *conditional-section*$_{opt}$
pp-endif:
  whitespace$_{opt}$ # *whitespace*$_{opt}$ endif *pp-new-line*
conditional-section:
  input-section
  skipped-section
skipped-section:
  skipped-section-part
  skipped-section *skipped-section-part*
skipped-section-part:
  skipped-characters$_{opt}$ *new-line*
  pp-directive
skipped-characters:
  whitespace$_{opt}$ *not-number-sign* *input-characters*$_{opt}$
not-number-sign:
  Any *input-character* except #

As indicated by the syntax, conditional compilation directives must be written as sets consisting of, in order, an `#if` directive, zero or more `#elif` directives, zero or one `#else` directive, and an `#endif` directive. Between the directives are conditional sections of source code. Each section is controlled by the immediately preceding directive. A conditional section may itself contain nested conditional compilation directives provided these directives form complete sets.

A *pp-conditional* selects at most one of the contained *conditional-section*s for normal lexical processing:

- The *pp-expression*s of the `#if` and `#elif` directives are evaluated in order until one yields `true`. If an expression yields `true`, the *conditional-section* of the corresponding directive is selected.
- If all *pp-expression*s yield `false`, and if an `#else` directive is present, the *conditional-section* of the `#else` directive is selected.
- Otherwise, no *conditional-section* is selected.

The selected *conditional-section*, if any, is processed as a normal *input-section*: the source code contained in the section must adhere to the lexical grammar; tokens are generated from the source code in the section; and pre-processing directives in the section have the prescribed effects.

The remaining *conditional-section*s, if any, are processed as *skipped-section*s: except for pre-processing directives, the source code in the section need not adhere to the lexical grammar; no tokens are generated from the source code in the section; and pre-processing directives in the section must be lexically correct but are not otherwise processed. Within a *conditional-section* that is being processed as a *skipped-section*, any nested *conditional-section*s (contained in nested `#if`…`#endif` and `#region`…`#endregion` constructs) are also processed as *skipped-section*s.

The following example illustrates how conditional compilation directives can nest:

```
#define Debug      // Debugging on
#undef Trace      // Tracing off
class PurchaseTransaction
{
   void Commit() {
      #if Debug
         CheckConsistency();
```

```
         #if Trace
            WriteToLog(this.ToString());
         #endif
      #endif
      CommitHelper();
   }
}
```

Except for pre-processing directives, skipped source code is not subject to lexical analysis. For example, the following is valid despite the unterminated comment in the `#else` section:

```
#define Debug      // Debugging on
class PurchaseTransaction
{
   void Commit() {
      #if Debug
         CheckConsistency();
      #else
         /* Do something else
      #endif
   }
}
```

Note, however, that pre-processing directives are required to be lexically correct even in skipped sections of source code.

Pre-processing directives are not processed when they appear inside multi-line input elements. For example, the program:

```
class Hello
{
   static void Main() {
      System.Console.WriteLine(@"hello,
#if Debug
      world
#else
      Nebraska
#endif
      ");
   }
}
```

results in the output:

```
hello,
#if Debug
      world
#else
      Nebraska
#endif
```

In peculiar cases, the set of pre-processing directives that is processed might depend on the evaluation of the *pp-expression*. The example:

```
#if X
   /*
#else
   /* */ class Q { }
#endif
```

always produces the same token stream (`class Q { }`), regardless of whether or not `X` is defined. If `X` is defined, the only processed directives are `#if` and `#endif`, due to the multi-line comment. If `X` is undefined, then three directives (`#if`, `#else`, `#endif`) are part of the directive set.

2.5.5 Diagnostic directives

The diagnostic directives are used to explicitly generate error and warning messages that are reported in the same way as other compile-time errors and warnings.

pp-diagnostic:
  whitespace$_{opt}$ # *whitespace*$_{opt}$ error *pp-message*
  whitespace$_{opt}$ # *whitespace*$_{opt}$ warning *pp-message*
pp-message:
  new-line
  whitespace *input-characters*$_{opt}$ *new-line*

The example:

```
#warning Code review needed before check-in
#if Debug && Retail
   #error A build can't be both debug and retail
#endif
class Test {...}
```

always produces a warning ("Code review needed before check-in"), and produces a compile-time error ("A build can't be both debug and retail") if the conditional symbols `Debug` and `Retail` are both defined. Note that a *pp-message* can contain arbitrary text; specifically, it need not contain well-formed tokens, as shown by the single quote in the word `can't`.

2.5.6 Region directives

The region directives are used to explicitly mark regions of source code.

pp-region:
  pp-start-region *conditional-section*$_{opt}$ *pp-end-region*
pp-start-region:
  whitespace$_{opt}$ # *whitespace*$_{opt}$ region *pp-message*
pp-end-region:
  whitespace$_{opt}$ # *whitespace*$_{opt}$ endregion *pp-message*

No semantic meaning is attached to a region; regions are intended for use by the programmer or by automated tools to mark a section of source code. The message specified in a `#region` or `#endregion` directive likewise has no semantic meaning; it merely serves to identify the region. Matching `#region` and `#endregion` directives may have different *pp-messages*.

The lexical processing of a region:

```
#region
...
#endregion
```

corresponds exactly to the lexical processing of a conditional compilation directive of the form:

```
#if true
...
#endif
```

2.5.7 Line directives

Line directives may be used to alter the line numbers and source file names that are reported by the compiler in output such as warnings and errors.

Line directives are most commonly used in meta-programming tools that generate C# source code from some other text input.

pp-line:
  whitespace$_{opt}$ # *whitespace*$_{opt}$ line *whitespace* *line-indicator* *pp-new-line*
line-indicator:
  decimal-digits *whitespace* *file-name*

  decimal-digits

  default

  hidden

file-name:
  " *file-name-characters* "
file-name-characters:
  file-name-character
  file-name-characters *file-name-character*
file-name-character:
  Any *input-character* except "

When no `#line` directives are present, the compiler reports true line numbers and source file names in its output. The `#line` directive is most commonly used in meta-programming tools that generate C# source code from some other text input. When processing a `#line` directive that includes a line-indicator that is not `default`, the compiler treats the line *after* the directive as having the given line number (and file name, if specified).

A `#line default` directive reverses the effect of all preceding `#line` directives. The compiler reports true line information for subsequent lines, precisely as if no `#line` directives had been processed.

A `#line hidden` directive has no effect on the file and line numbers reported in error messages, but does affect source level debugging. When debugging, all lines between a `#line hidden` directive and the subsequent `#line` directive (that is not `#line hidden`) have no line number information. When stepping through code in the debugger, these lines will be skipped entirely.

Note that a *file-name* differs from a regular string literal in that escape characters are not processed; the '\' character simply designates an ordinary backslash character within a *file-name*.

3. Basic concepts

This chapter defines basic concepts that are required for understanding subsequent chapters.

3.1 Application startup

An assembly that has an entry point is called an application. When an application is run, a new application domain is created. Several different instantiations of an application may exist on the same machine at the same time, and each has its own application domain.

An application domain enables application isolation by acting as a container for application state. An application domain acts as a container and boundary for the types defined in the application and the class libraries it uses. Types loaded into one application domain are distinct from the same type loaded into another application domain, and instances of objects are not directly shared between application domains. For instance, each application domain has its own copy of static variables for these types, and a static constructor for a type is run at most once per application domain. Implementations are free to provide implementation-specific policy or mechanisms for the creation and destruction of application domains.

Application startup occurs when the execution environment calls a designated method, which is referred to as the application's entry point. This entry point method is always named `Main`, and can have one of the following signatures:

```
static void Main() {...}
static void Main(string[] args) {...}
static int Main() {...}
static int Main(string[] args) {...}
```

As shown, the entry point may optionally return an `int` value. This return value is used in application termination (Section 3.2).

The entry point may optionally have one formal parameter. The parameter may have any name, but the type of the parameter must be `string[]`. If the formal parameter is present, the execution environment creates and passes a `string[]` argument containing the command-line arguments that were specified when the application was started. The `string[]` argument is never null, but it may have a length of zero if no command-line arguments were specified.

Since C# supports method overloading, a class or struct may contain multiple definitions of some method, provided each has a different signature. However, within a single program, no class or struct may contain more than one method called `Main` whose definition qualifies it to be used as an application entry point. Other overloaded versions of `Main` are permitted, however, provided they have more than one parameter, or their only parameter is other than type `string[]`.

An application can be made up of multiple classes or structs. It is possible for more than one of these classes or structs to contain a method called `Main` whose definition qualifies it to be used as an application entry point. In such cases, an external mechanism (such as a command-line compiler option) must be used to select one of these `Main` methods as the entry point.

In C#, every method must be defined as a member of a class or struct. Ordinarily, the declared accessibility (Section 3.5.1) of a method is determined by the access modifiers (Section 10.2.3) specified in its declaration, and similarly the declared accessibility of a type is determined by the access modifiers specified in its declaration. In order for a given method of a given type to be callable, both the type and the member must be accessible. However, the application entry point is a special case. Specifically, the execution environment can access the application's entry point regardless of its declared accessibility and regardless of the declared accessibility of its enclosing type declarations.

In all other respects, entry point methods behave like those that are not entry points.

3.2 Application termination

Application termination returns control to the execution environment.

If the return type of the application's entry point method is `int`, the value returned serves as the application's termination status code. The purpose of this code is to allow communication of success or failure to the execution environment.

If the return type of the entry point method is `void`, reaching the right brace (`}`) that terminates that method, or executing a `return` statement that has no expression, results in a termination status code of `0`.

Prior to an application's termination, destructors for all of its objects that have not yet been garbage collected are called, unless such cleanup has been suppressed (by a call to the library method `GC.SuppressFinalize`, for example).

3.3 Declarations

Declarations in a C# program define the constituent elements of the program. C# programs are organized using namespaces (Section 9), which can contain type declarations and nested namespace declarations. Type declarations (Section 9.5) are used to define classes (Section 10), structs (Section 11), interfaces (Section 13), enums (Section 14), and delegates (Section 15). The kinds of members permitted in a type declaration depend on the form of the type declaration. For instance, class declarations can contain declarations for constants (Section 10.3), fields (Section 10.4), methods (Section 10.5), properties (Section 10.6), events (Section 10.7), indexers (Section 10.8), operators (Section 10.9), instance constructors (Section 10.10), static constructors (Section 10.11), destructors (Section 10.12), and nested types.

A declaration defines a name in the declaration space to which the declaration belongs. Except for overloaded members (Section 3.6), it is a compile-time error to have two or more declarations that introduce members with the same name in a declaration space. It is never possible for a declaration space to contain different kinds of members with the same name. For example, a declaration space can never contain a field and a method by the same name.

There are several different types of declaration spaces, as described in the following.

- Within all source files of a program, namespace-member-declarations with no enclosing namespace-declaration are members of a single combined declaration space called the global declaration space.
- Within all source files of a program, namespace-member-declarations within namespace-declarations that have the same fully qualified namespace name are members of a single combined declaration space.
- Each class, struct, or interface declaration creates a new declaration space. Names are introduced into this declaration space through class-member-declarations, struct-member-declarations, or interface-member-declarations. Except for overloaded instance constructor declarations and static constructor declarations, a class or struct member declaration cannot introduce a member by the same name as the class or struct. A class, struct, or interface permits the declaration of overloaded methods and indexers. Furthermore, a class or struct permits the declaration of overloaded instance constructors and operators. For example, a class, struct, or interface may contain multiple method declarations with the same name, provided these method declarations differ in their signature (Section 3.6). Note that base classes do not contribute to the declaration space of a class, and base interfaces do not contribute to the declaration space of an interface. Thus, a derived class or interface is allowed to declare a member with the same name as an inherited member. Such a member is said to hide the inherited member.
- Each enumeration declaration creates a new declaration space. Names are introduced into this declaration space through enum-member-declarations.
- Each block or switch-block creates a different declaration space for local variables and constants. Names are introduced into this declaration space through local-variable-declarations and local-constant-declarations. If a block is the body of an instance constructor, method, or operator declaration, or a get or set accessor for an indexer declaration, the parameters declared in such a declaration are members of the block's local variable declaration space. The local variable declaration space of a block includes any nested blocks. Thus, within a nested block it is not possible to declare a local variable with the same name as a local variable in an enclosing block.
- Each block or switch-block creates a separate declaration space for labels. Names are introduced into this declaration space through labeled-statements, and the names are referenced through *goto* statements. The label declaration space of a block includes any nested blocks. Thus, within a nested block it is not possible to declare a label with the same name as a label in an enclosing block.

The textual order in which names are declared is generally of no significance. In particular, textual order is not significant for the declaration and use of namespaces, constants, methods, properties, events, indexers, operators, instance constructors, destructors, static constructors, and types. Declaration order is significant in the following ways:

- Declaration order for field declarations and local variable declarations determines the order in which their initializers (if any) are executed.
- Local variables must be defined before they are used (Section 3.7).
- Declaration order for enum member declarations (Section 14.3) is significant when constant-expression values are omitted.

The declaration space of a namespace is "open ended", and two namespace declarations with the same fully qualified name contribute to the same declaration space. For example

```
namespace Megacorp.Data
{
   class Customer
   {
```

```
      ...
   }
}
namespace Megacorp.Data
{
   class Order
   {
      ...
   }
}
```

The two namespace declarations above contribute to the same declaration space, in this case declaring two classes with the fully qualified names `Megacorp.Data.Customer` and `Megacorp.Data.Order`. Because the two declarations contribute to the same declaration space, it would have caused a compile-time error if each contained a declaration of a class with the same name.

As previously specified, the declaration space of a block includes any nested blocks. Thus, in the following example, the `F` and `G` methods result in a compile-time error because the name `i` is declared in the outer block and cannot be redeclared in the inner block. However, the `H` and `I` methods are valid since the two `i`'s are declared in separate non-nested blocks.

```
class A
{
   void F() {
      int i = 0;
      if (true) {
         int i = 1;
      }
   }
   void G() {
      if (true) {
         int i = 0;
      }
      int i = 1;
   }
   void H() {
      if (true) {
         int i = 0;
      }
      if (true) {
         int i = 1;
      }
   }
   void I() {
      for (int i = 0; i < 10; i++)
         H();
      for (int i = 0; i < 10; i++)
         H();
   }
}
```

3.4 Members

Namespaces and types have members. The members of an entity are generally available through the use of a qualified name that starts with a reference to the entity, followed by a "`.`" token, followed by the name of the member.

Members of a type are either declared in the type or inherited from the base class of the type. When a type inherits from a base class, all members of the base class, except instance constructors, destructors and static constructors, become members of the derived type. The declared accessibility of a base class member does not control whether the member is inherited — inheritance extends to any member that is not an instance constructor, static constructor, or destructor. However, an inherited member may not be accessible in a derived type, either because of its declared accessibility (Section 3.5.1) or because it is hidden by a declaration in the type itself (Section 3.7.1.2).

3.4.1 Namespace members

Namespaces and types that have no enclosing namespace are members of the global namespace. This corresponds directly to the names declared in the global declaration space.

Namespaces and types declared within a namespace are members of that namespace. This corresponds directly to the names declared in the declaration space of the namespace.

Namespaces have no access restrictions. It is not possible to declare private, protected, or internal namespaces, and namespace names are always publicly accessible.

3.4.2 Struct members

The members of a struct are the members declared in the struct and the members inherited from class `object`.

The members of a simple type correspond directly to the members of the struct type aliased by the simple type:

- The members of `sbyte` are the members of the `System.SByte` struct.
- The members of `byte` are the members of the `System.Byte` struct.
- The members of `short` are the members of the `System.Int16` struct.
- The members of `ushort` are the members of the `System.UInt16` struct.
- The members of `int` are the members of the `System.Int32` struct.
- The members of `uint` are the members of the `System.UInt32` struct.
- The members of `long` are the members of the `System.Int64` struct.
- The members of `ulong` are the members of the `System.UInt64` struct.
- The members of `char` are the members of the `System.Char` struct.
- The members of `float` are the members of the `System.Single` struct.
- The members of `double` are the members of the `System.Double` struct.
- The members of `decimal` are the members of the `System.Decimal` struct.
- The members of `bool` are the members of the `System.Boolean` struct.

3.4.3 Enumeration members

The members of an enumeration are the constants declared in the enumeration and the members inherited from class `System.Enum`.

3.4.4 Class members

The members of a class are the members declared in the class and the members inherited from the base class (except for class `object` which has no base class). The members inherited from the base class include the constants, fields, methods, properties, events, indexers, operators, and types of the base class, but not the instance constructors, destructors and static constructors of the base class. Base class members are inherited without regard to their accessibility.

A class declaration may contain declarations of constants, fields, methods, properties, events, indexers, operators, instance constructors, destructors, static constructors and types.

The members of `object` and `string` correspond directly to the members of the class types they alias:

- The members of `object` are the members of the `System.Object` class.
- The members of `string` are the members of the `System.String` class.

3.4.5 Interface members

The members of an interface are the members declared in the interface and in all base interfaces of the interface, and the members inherited from class `object`.

3.4.6 Array members

The members of an array are the members inherited from class `System.Array`.

3.4.7 Delegate members

The members of a delegate are the members inherited from class `System.Delegate`.

3.5 Member access

Declarations of members allow control over member access. The accessibility of a member is established by the declared accessibility (Section 3.5.1) of the member combined with the accessibility of the immediately containing type, if any.

When access to a particular member is allowed, the member is said to be accessible. Conversely, when access to a particular member is disallowed, the member is said to be inaccessible. Access to a member is permitted when the textual location in which the access takes place is included in the accessibility domain (Section 3.5.2) of the member.

3.5.1 Declared accessibility

The declared accessibility of a member can be one of the following:

- Public, which is selected by including a `public` modifier in the member declaration. The intuitive meaning of `public` is "access not limited".
- Protected, which is selected by including a `protected` modifier in the member declaration. The intuitive meaning of `protected` is "access limited to the containing class or types derived from the containing class".
- Internal, which is selected by including an `internal` modifier in the member declaration. The intuitive meaning of `internal` is "access limited to this program".
- Protected internal (meaning protected or internal), which is selected by including both a `protected` and an `internal` modifier in the member declaration. The intuitive meaning of `protected internal` is "access limited to this program or types derived from the containing class".
- Private, which is selected by including a `private` modifier in the member declaration. The intuitive meaning of `private` is "access limited to the containing type".

Depending on the context in which a member declaration takes place, only certain types of declared accessibility are permitted. Furthermore, when a member declaration does not include any access modifiers, the context in which the declaration takes place determines the default declared accessibility.

- Namespaces implicitly have `public` declared accessibility. No access modifiers are allowed on namespace declarations.
- Types declared in compilation units or namespaces can have `public` or `internal` declared accessibility and default to `internal` declared accessibility.
- Class members can have any of the five kinds of declared accessibility and default to `private` declared accessibility. (Note that a type declared as a member of a class can have any of the five kinds of declared accessibility, whereas a type declared as a member of a namespace can have only `public` or `internal` declared accessibility.)
- Struct members can have `public`, `internal`, or `private` declared accessibility and default to `private` declared accessibility because structs are implicitly sealed. Struct members introduced in a struct (that is, not inherited by that struct) cannot have `protected` or `protected internal` declared accessibility. (Note that a type declared as a member of a struct can have `public`, `internal`, or `private` declared accessibility, whereas a type declared as a member of a namespace can have only `public` or `internal` declared accessibility.)
- Interface members implicitly have `public` declared accessibility. No access modifiers are allowed on interface member declarations.
- Enumeration members implicitly have `public` declared accessibility. No access modifiers are allowed on enumeration member declarations.

3.5.2 Accessibility domains

The accessibility domain of a member consists of the (possibly disjoint) sections of program text in which access to the member is permitted. For purposes of defining the accessibility domain of a member, a member is said to be top-level if it is not declared within a type, and a member is said to be nested if it is declared within another type. Furthermore, the program text of a program is defined as all program text contained in all source files of the program, and the program text of a type is defined as all program text contained between the opening and closing "`{`" and "`}`" tokens in the class-body, struct-body, interface-body, or enum-body of the type (including, possibly, types that are nested within the type).

The accessibility domain of a predefined type (such as `object`, `int`, or `double`) is unlimited.

The accessibility domain of a top-level type `T` that is declared in a program `P` is defined as follows:

- If the declared accessibility of `T` is `public`, the accessibility domain of `T` is the program text of `P` and any program that references `P`.
- If the declared accessibility of `T` is `internal`, the accessibility domain of `T` is the program text of `P`.

From these definitions it follows that the accessibility domain of a top-level type is always at least the program text of the program in which that type is declared.

The accessibility domain of a nested member `M` declared in a type `T` within a program `P` is defined as follows (noting that M itself may possibly be a type):

- If the declared accessibility of `M` is `public`, the accessibility domain of `M` is the accessibility domain of `T`.
- If the declared accessibility of `M` is `protected internal`, let `D` be the union of the program text of `P` and the program text of any type derived from `T`, which is declared outside `P`. The accessibility domain of `M` is the intersection of the accessibility domain of `T` with `D`.
- If the declared accessibility of `M` is `protected`, let `D` be the union of the program text of `T` and the program text of any type derived from `T`. The accessibility domain of `M` is the intersection of the accessibility domain of `T` with `D`.
- If the declared accessibility of `M` is `internal`, the accessibility domain of `M` is the intersection of the accessibility domain of `T` with the program text of `P`.
- If the declared accessibility of `M` is `private`, the accessibility domain of `M` is the program text of `T`.

From these definitions it follows that the accessibility domain of a nested member is always at least the program text of the type in which the member is declared. Furthermore, it follows that the accessibility domain of a member is never more inclusive than the accessibility domain of the type in which the member is declared.

In intuitive terms, when a type or member `M` is accessed, the following steps are evaluated to ensure that the access is permitted:

- First, if `M` is declared within a type (as opposed to a compilation unit or a namespace), a compile-time error occurs if that type is not accessible.
- Then, if `M` is `public`, the access is permitted.
- Otherwise, if `M` is `protected internal`, the access is permitted if it occurs within the program in which `M` is declared, or if it occurs within a class derived from the class in which `M` is declared and takes place through the derived class type (Section 3.5.3).
- Otherwise, if `M` is `protected`, the access is permitted if it occurs within the class in which `M` is declared, or if it occurs within a class derived from the class in which `M` is declared and takes place through the derived class type (Section 3.5.3).
- Otherwise, if `M` is `internal`, the access is permitted if it occurs within the program in which `M` is declared.
- Otherwise, if `M` is `private`, the access is permitted if it occurs within the type in which `M` is declared.
- Otherwise, the type or member is inaccessible, and a compile-time error occurs.

In the example

```
public class A
{
   public static int X;
   internal static int Y;
   private static int Z;
}
internal class B
```

```
{
   public static int X;
   internal static int Y;
   private static int Z;
   public class C
   {
      public static int X;
      internal static int Y;
      private static int Z;
   }
   private class D
   {
      public static int X;
      internal static int Y;
      private static int Z;
   }
}
```

the classes and members have the following accessibility domains:

- The accessibility domain of `A` and `A.X` is unlimited.
- The accessibility domain of `A.Y`, `B`, `B.X`, `B.Y`, `B.C`, `B.C.X`, and `B.C.Y` is the program text of the containing program.
- The accessibility domain of `A.Z` is the program text of `A`.
- The accessibility domain of `B.Z` and `B.D` is the program text of `B`, including the program text of `B.C` and `B.D`.
- The accessibility domain of `B.D.X` and `B.D.Y` is the program text of `B`, including the program text of `B.C` and `B.D`.
- The accessibility domain of `B.D.Z` is the program text of `B.D`.

As the example illustrates, the accessibility domain of a member is never larger than that of a containing type. For example, even though all `X` members have public declared accessibility, all but `A.X` have accessibility domains that are constrained by a containing type.

As described in Section 3.4, all members of a base class, except for instance constructors, destructors and static constructors, are inherited by derived types. This includes even private members of a base class. However, the accessibility domain of a private member includes only the program text of the type in which the member is declared. In the example

```
class A
{
   int x;
   static void F(B b) {
      b.x = 1;      // Ok
   }
}
class B: A
{
   static void F(B b) {
      b.x = 1;      // Error, x not accessible
   }
}
```

the `B` class inherits the private member `x` from the `A` class. Because the member is private, it is only accessible within the *class-body* of `A`. Thus, the access to `b.x` succeeds in the `A.F` method, but fails in the `B.F` method.

3.5.3 Protected access for instance members

When a `protected` instance member is accessed outside the program text of the class in which it is declared, and when a `protected internal` instance member is accessed outside the program text of the program in which it is declared, the access is required to take place through an instance of the derived class type in which the access occurs. Let `B` be a base class that declares a protected instance member `M`, and let `D` be a class that derives from `B`. Within the class-body of `D`, access to `M` can take one of the following forms:

- An unqualified type-name or primary-expression of the form `M`.
- A primary-expression of the form `E.M`, provided the type of `E` is `D` or a class derived from `D`.
- A primary-expression of the form `base.M`.

In addition to these forms of access, a derived class can access a protected instance constructor of a base class in a constructor-initializer (Section 10.10.1).

In the example

```
public class A
{
   protected int x;
   static void F(A a, B b) {
      a.x = 1;      // Ok
      b.x = 1;      // Ok
   }
}
public class B: A
{
   static void F(A a, B b) {
      a.x = 1;      // Error, must access through instance of B
      b.x = 1;      // Ok
   }
}
```

within `A`, it is possible to access `x` through instances of both `A` and `B`, since in either case the access takes place through an instance of `A` or a class derived from `A`. However, within `B`, it is not possible to access `x` through an instance of `A`, since `A` does not derive from `B`.

3.5.4 Accessibility constraints

Several constructs in the C# language require a type to be at least as accessible as a member or another type. A type `T` is said to be at least as accessible as a member or type `M` if the accessibility domain of `T` is a superset of the accessibility domain of `M`. In other words, `T` is at least as accessible as `M` if `T` is accessible in all contexts in which `M` is accessible.

The following accessibility constraints exist:

- The direct base class of a class type must be at least as accessible as the class type itself.
- The explicit base interfaces of an interface type must be at least as accessible as the interface type itself.
- The return type and parameter types of a delegate type must be at least as accessible as the delegate type itself.
- The type of a constant must be at least as accessible as the constant itself.
- The type of a field must be at least as accessible as the field itself.
- The return type and parameter types of a method must be at least as accessible as the method itself.
- The type of a property must be at least as accessible as the property itself.
- The type of an event must be at least as accessible as the event itself.
- The type and parameter types of an indexer must be at least as accessible as the indexer itself.
- The return type and parameter types of an operator must be at least as accessible as the operator itself.
- The parameter types of an instance constructor must be at least as accessible as the instance constructor itself.

In the example

```
class A {...}
public class B: A {...}
```

the `B` class results in a compile-time error because `A` is not at least as accessible as `B`.

Likewise, in the example

```
class A {...}
public class B
{
   A F() {...}
   internal A G() {...}
   public A H() {...}
}
```

the `H` method in `B` results in a compile-time error because the return type `A` is not at least as accessible as the method.

3.6 Signatures and overloading

Methods, instance constructors, indexers, and operators are characterized by their signatures:

- The signature of a method consists of the name of the method and the type and kind (value, reference, or output) of each of its formal parameters, considered in the order left to right. The signature of a method specifically does not include the return type, nor does it include the `params` modifier that may be specified for the right-most parameter.
- The signature of an instance constructor consists of the type and kind (value, reference, or output) of each of its formal parameters, considered in the order left to right. The signature of an instance constructor specifically does not include the `params` modifier that may be specified for the right-most parameter.
- The signature of an indexer consists of the type of each of its formal parameters, considered in the order left to right. The signature of an indexer specifically does not include the element type.
- The signature of an operator consists of the name of the operator and the type of each of its formal parameters, considered in the order left to right. The signature of an operator specifically does not include the result type.

Signatures are the enabling mechanism for overloading of members in classes, structs, and interfaces:

- Overloading of methods permits a class, struct, or interface to declare multiple methods with the same name, provided their signatures are unique within that class, struct, or interface.
- Overloading of instance constructors permits a class or struct to declare multiple instance constructors, provided their signatures are unique within that class or struct.
- Overloading of indexers permits a class, struct, or interface to declare multiple indexers, provided their signatures are unique within that class, struct, or interface.
- Overloading of operators permits a class or struct to declare multiple operators with the same name, provided their signatures are unique within that class or struct.

The following example shows a set of overloaded method declarations along with their signatures.

```
interface ITest
{
   void F();                     // F()
   void F(int x);                  // F(int)
   void F(ref int x);            // F(ref int)
   void F(int x, int y);         // F(int, int)
   int F(string s);               // F(string)
   int F(int x);                  // F(int)         error
   void F(string[] a);            // F(string[])
   void F(params string[] a);      // F(string[])      error
}
```

Note that any `ref` and `out` parameter modifiers (Section 10.5.1) are part of a signature. Thus, `F(int)` and `F(ref int)` are unique signatures. Also, note that the return type and the `params` modifier are not part of a signature, so it is not possible to overload solely based on return type or on the inclusion or exclusion of the `params` modifier. As such, the declarations of the methods `F(int)` and `F(params string[])` identified above result in a compile-time error.

3.7 Scopes

The scope of a name is the region of program text within which it is possible to refer to the entity declared by the name without qualification of the name. Scopes can be nested, and an inner scope may redeclare the meaning of a name from an outer scope. (This does not, however, remove the restriction imposed by Section 3.3 that within a nested block it is not possible to declare a local variable with the same name as a local variable in an enclosing block.) The name from the outer scope is then said to be hidden in the region of program text covered by the inner scope, and access to the outer name is only possible by qualifying the name.

- The scope of a namespace member declared by a *namespace-member-declaration* (Section 9.4) with no enclosing namespace-declaration is the entire program text.
- The scope of a namespace member declared by a namespace-member-declaration within a namespace-declaration whose fully qualified name is `N` is the namespace-body of every namespace-declaration whose fully qualified name is `N` or starts with `N`, followed by a period.
- The scope of a name defined or imported by a using-directive (Section 9.3) extends over the namespace-member-declarations of the compilation-unit or namespace-body in which the using-directive occurs. A using-directive may make zero or more namespace or type names available within a particular compilation-unit or namespace-body, but does not contribute any new members to the underlying declaration space. In other words, a using-directive is not transitive but rather affects only the compilation-unit or namespace-body in which it occurs.
- The scope of a member declared by a class-member-declaration (Section 10.2) is the class-body in which the declaration occurs. In addition, the scope of a class member extends to the *class-body* of those derived classes that are included in the accessibility domain (Section 3.5.2) of the member.
- The scope of a member declared by a struct-member-declaration (Section 11.2) is the struct-body in which the declaration occurs.
- The scope of a member declared by an enum-member-declaration (Section 14.3) is the enum-*body* in which the declaration occurs.
- The scope of a parameter declared in a method-declaration (Section 10.5) is the method-body of that method-declaration.
- The scope of a parameter declared in an indexer-declaration (Section 10.8) is the accessor-declarations of that indexer-declaration.
- The scope of a parameter declared in an operator-declaration (Section 10.9) is the block of that operator-declaration.
- The scope of a parameter declared in a constructor-declaration (Section 10.10) is the constructor-initializer and block of that constructor-declaration.
- The scope of a label declared in a labeled-statement (Section 8.4) is the block in which the declaration occurs.
- The scope of a local variable declared in a local-variable-declaration (Section 8.5.1) is the block in which the declaration occurs.
- The scope of a local variable declared in a switch-block of a `switch` statement (Section 8.7.2) is the switch-block.
- The scope of a local variable declared in a for-initializer of a `for` statement (Section 8.8.3) is the for-initializer, the for-condition, the for-iterator, and the contained statement of the `for` statement.
- The scope of a local constant declared in a local-constant-declaration (Section 8.5.2) is the block in which the declaration occurs. It is a compile-time error to refer to a local constant in a textual position that precedes its constant-declarator.

Within the scope of a namespace, class, struct, or enumeration member it is possible to refer to the member in a textual position that precedes the declaration of the member. For example

```
class A
{
   void F() {
      i = 1;
   }
   int i = 0;
}
```

Here, it is valid for `F` to refer to `i` before it is declared.

Within the scope of a local variable, it is a compile-time error to refer to the local variable in a textual position that precedes the *local*-variable-declarator of the local variable. For example

```
class A
{
   int i = 0;
   void F() {
      i = 1;               // Error, use precedes declaration
      int i;
      i = 2;
   }
   void G() {
      int j = (j = 1);      // Valid
   }
   void H() {
      int a = 1, b = ++a;   // Valid
   }
}
```

In the `F` method above, the first assignment to `i` specifically does not refer to the field declared in the outer scope. Rather, it refers to the local variable and it results in a compile-time error because it textually precedes the declaration of the variable. In the `G` method, the use of `j` in the initializer for the declaration of `j` is valid because the use does not precede the *local*-variable-declarator. In the `H` method, a subsequent *local*-variable-declarator correctly refers to a local variable declared in an earlier *local*-variable-declarator within the same local-variable-declaration.

The scoping rules for local variables are designed to guarantee that the meaning of a name used in an expression context is always the same within a block. If the scope of a local variable were to extend only from its declaration to the end of the block, then in the example above, the first assignment would assign to the instance variable and the second assignment would assign to the local variable, possibly leading to compile-time errors if the statements of the block were later to be rearranged.

The meaning of a name within a block may differ based on the context in which the name is used. In the example

```
using System;
class A {}
class Test
{
   static void Main() {
      string A = "hello, world";
      string s = A;                        // expression context
      Type t = typeof(A);                  // type context
      Console.WriteLine(s);               // writes "hello, world"
      Console.WriteLine(t);               // writes "A"
   }
}
```

the name `A` is used in an expression context to refer to the local variable `A` and in a type context to refer to the class `A`.

3.7.1 Name hiding

The scope of an entity typically encompasses more program text than the declaration space of the entity. In particular, the scope of an entity may include declarations that introduce new declaration spaces containing entities of the same name. Such declarations cause the original entity to become hidden. Conversely, an entity is said to be visible when it is not hidden.

Name hiding occurs when scopes overlap through nesting and when scopes overlap through inheritance. The characteristics of the two types of hiding are described in the following sections.

3.7.1.1 Hiding through nesting

Name hiding through nesting can occur as a result of nesting namespaces or types within namespaces, as a result of nesting types within classes or structs, and as a result of parameter and local variable declarations.

In the example

```
class A
{
   int i = 0;
   void F() {
      int i = 1;
   }
   void G() {
      i = 1;
   }
}
```

within the `F` method, the instance variable `i` is hidden by the local variable `i`, but within the `G` method, `i` still refers to the instance variable.

When a name in an inner scope hides a name in an outer scope, it hides all overloaded occurrences of that name. In the example

```
class Outer
{
   static void F(int i) {}
   static void F(string s) {}
   class Inner
   {
      void G() {
         F(1);            // Invokes Outer.Inner.F
         F("Hello");      // Error
      }
      static void F(long l) {}
   }
}
```

the call `F(1)` invokes the `F` declared in `Inner` because all outer occurrences of `F` are hidden by the inner declaration. For the same reason, the call `F("Hello")` results in a compile-time error.

3.7.1.2 Hiding through inheritance

Name hiding through inheritance occurs when classes or structs redeclare names that were inherited from base classes. This type of name hiding takes one of the following forms:

- A constant, field, property, event, or type introduced in a class or struct hides all base class members with the same name.
- A method introduced in a class or struct hides all non-method base class members with the same name, and all base class methods with the same signature (method name and parameter count, modifiers, and types).
- An indexer introduced in a class or struct hides all base class indexers with the same signature (parameter count and types).

The rules governing operator declarations (Section 10.9) make it impossible for a derived class to declare an operator with the same signature as an operator in a base class. Thus, operators never hide one another.

Contrary to hiding a name from an outer scope, hiding an accessible name from an inherited scope causes a warning to be reported. In the example

```
class Base
{
   public void F() {}
}
class Derived: Base
{
   public void F() {}      // Warning, hiding an inherited name
}
```

the declaration of `F` in `Derived` causes a warning to be reported. Hiding an inherited name is specifically not an error, since that would preclude separate evolution of base classes. For example, the above situation might have come about because a later version of `Base` introduced an `F` method that wasn't present in an earlier version of the class. Had the above situation been an error, then *any* change made to a base class in a separately versioned class library could potentially cause derived classes to become invalid.

The warning caused by hiding an inherited name can be eliminated through use of the `new` modifier:

```
class Base
{
   public void F() {}
}
class Derived: Base
{
   new public void F() {}
}
```

The `new` modifier indicates that the `F` in `Derived` is "new", and that it is indeed intended to hide the inherited member.

A declaration of a new member hides an inherited member only within the scope of the new member.

```
class Base
{
   public static void F() {}
}
class Derived: Base
{
   new private static void F() {}   // Hides Base.F in Derived only
}
class MoreDerived: Derived
{
   static void G() { F(); }         // Invokes Base.F
}
```

In the example above, the declaration of `F` in `Derived` hides the `F` that was inherited from `Base`, but since the new `F` in `Derived` has private access, its scope does not extend to `MoreDerived`. Thus, the call `F()` in `MoreDerived.G` is valid and will invoke `Base.F`.

3.8 Namespace and type names

Several contexts in a C# program require a namespace-name or a type-name to be specified. Either form of name is written as one or more identifiers separated by "`.`" tokens.

namespace-name:
  namespace-or-type-name
type-name:
  namespace-or-type-name
namespace-or-type-name:
  identifier
  namespace-or-type-name `.` *identifier*

A type-name is a namespace-or-type-name that refers to a type. Following resolution as described below, the namespace-or-type-name of a type-name must refer to a type, or otherwise a compile-time error occurs.

A namespace-name is a namespace-or-type-name that refers to a namespace. Following resolution as described below, the namespace-or-type-name of a namespace-name must refer to a namespace, or otherwise a compile-time error occurs.

The meaning of a namespace-or-type-name is determined as follows:

- If the namespace-or-type-name consists of a single identifier:
  - If the namespace-or-type-name appears within the body of a class or struct declaration, then starting with that class or struct declaration and continuing with each enclosing class or struct declaration (if any), if a member with the given name exists, is accessible, and denotes a type, then the namespace-or-type-name refers to that member. Note that non-type members (constants, fields, methods, properties, indexers, operators, instance constructors, destructors, and static constructors) are ignored when determining the meaning of a namespace-or-type-name.
  - Otherwise, starting with the namespace in which the namespace-or-type-name occurs, continuing with each enclosing namespace (if any), and ending with the global namespace, the following steps are evaluated until an entity is located:
  - If the namespace contains a namespace member with the given name, then the namespace-or-type-name refers to that member and, depending on the member, is classified as a namespace or a type.
  - Otherwise, if the namespace has a corresponding namespace declaration enclosing the location where the namespace-or-type-name occurs, then:

    If the namespace declaration contains a using-alias-directive that associates the given name with an imported namespace or type, then the namespace-or-type-name refers to that namespace or type.

    Otherwise, if the namespaces imported by the using-namespace-directives of the namespace declaration contain exactly one type with the given name, then the namespace-or-type-name refers to that type.

    Otherwise, if the namespaces imported by the using-namespace-directives of the namespace declaration contain more than one type with the given name, then the namespace-or-type-name is ambiguous and an error occurs.
  - Otherwise, the namespace-or-type-name is undefined and a compile-time error occurs.
- Otherwise, the namespace-or-type-name is of the form `N.I`, where `N` is a namespace-or-type-name consisting of all identifiers but the rightmost one, and `I` is the rightmost identifier. `N` is first resolved as a namespace-or-type-name. If the resolution of `N` is not successful, a compile-time error occurs. Otherwise, `N.I` is resolved as follows:
  - If `N` is a namespace and `I` is the name of an accessible member of that namespace, then `N.I` refers to that member and, depending on the member, is classified as a namespace or a type.
  - If `N` is a class or struct type and `I` is the name of an accessible type in `N`, then `N.I` refers to that type.
  - Otherwise, `N.I` is an invalid namespace-or-type-name, and a compile-time error occurs.

3.8.1 Fully qualified names

Every namespace and type has a fully qualified name, which uniquely identifies the namespace or type amongst all others. The fully qualified name of a namespace or type `N` is determined as follows:

- If `N` is a member of the global namespace, its fully qualified name is `N`.
- Otherwise, its fully qualified name is `S.N`, where `S` is the fully qualified name of the namespace or type in which `N` is declared.

In other words, the fully qualified name of `N` is the complete hierarchical path of identifiers that lead to `N`, starting from the global namespace. Because every member of a namespace or type must have a unique name, it follows that the fully qualified name of a namespace or type is always unique.

The example below shows several namespace and type declarations along with their associated fully qualified names.

```
class A {}            // A
namespace X            // X
{
   class B            // X.B
   {
      class C {}      // X.B.C
   }
   namespace Y         // X.Y
   {
      class D {}      // X.Y.D
   }
}
namespace X.Y         // X.Y
{
   class E {}         // X.Y.E
}
```

3.9 Automatic memory management

C# employs automatic memory management, which frees developers from manually allocating and freeing the memory occupied by objects. Automatic memory management policies are implemented by a garbage collector. The memory management life cycle of an object is as follows:

1. When the object is created, memory is allocated for it, the constructor is run, and the object is considered live.
2. If the object, or any part of it, cannot be accessed by any possible continuation of execution, other than the running of destructors, the object is considered no longer in use, and it becomes eligible for destruction. The C# compiler and the garbage collector may choose to analyze code to determine which references to an object may be used in the future. For instance, if a local variable that is in scope is the only existing reference to an object, but that local variable is never referred to in any possible continuation of execution from the current execution point in the procedure, the garbage collector may (but is not required to) treat the object as no longer in use.
3. Once the object is eligible for destruction, at some unspecified later time the destructor (Section 10.12) (if any) for the object is run. Unless overridden by explicit calls, the destructor for the object is run once only.
4. Once the destructor for an object is run, if that object, or any part of it, cannot be accessed by any possible continuation of execution, including the running of destructors, the object is considered inaccessible and the object becomes eligible for collection.
5. Finally, at some time after the object becomes eligible for collection, the garbage collector frees the memory associated with that object.

The garbage collector maintains information about object usage, and uses this information to make memory management decisions, such as where in memory to locate a newly created object, when to relocate an object, and when an object is no longer in use or inaccessible.

Like other languages that assume the existence of a garbage collector, C# is designed so that the garbage collector may implement a wide range of memory management policies. For instance, C# does not require that destructors be run or that objects be collected as soon as they are eligible, or that destructors be run in any particular order, or on any particular thread.

The behavior of the garbage collector can be controlled, to some degree, via static methods on the class `System.GC`. This class can be used to request a collection to occur, destructors to be run (or not run), and so forth.

Since the garbage collector is allowed wide latitude in deciding when to collect objects and run destructors, a conforming implementation may produce output that differs from that shown by the following code. The program

```
using System;
class A
{
   ~A() {
      Console.WriteLine("Destruct instance of A");
   }
}
class B
{
   object Ref;
   public B(object o) {
      Ref = o;
   }
   ~B() {
      Console.WriteLine("Destruct instance of B");
   }
}
class Test
{
   static void Main() {
      B b = new B(new A());
      b = null;
      GC.Collect();
      GC.WaitForPendingFinalizers();
   }
}
```

creates an instance of class `A` and an instance of class `B`. These objects become eligible for garbage collection when the variable `b`

is assigned the value `null`, since after this time it is impossible for any user-written code to access them. The output could be either

```
Destruct instance of A
Destruct instance of B
```

or

```
Destruct instance of B
Destruct instance of A
```

because the language imposes no constraints on the order in which objects are garbage collected.

In subtle cases, the distinction between "eligible for destruction" and "eligible for collection" can be important. For example,

```
using System;
class A
{
   ~A() {
      Console.WriteLine("Destruct instance of A");
   }
   public void F() {
      Console.WriteLine("A.F");
      Test.RefA = this;
   }
}
class B
{
   public A Ref;
   ~B() {
      Console.WriteLine("Destruct instance of B");
      Ref.F();
   }
}
class Test
{
   public static A RefA;
   public static B RefB;
   static void Main() {
      RefB = new B();
      RefA = new A();
      RefB.Ref = RefA;
      RefB = null;
      RefA = null;
      // A and B now eligible for destruction
      GC.Collect();
      GC.WaitForPendingFinalizers();
      // B now eligible for collection, but A is not
      if (RefA != null)
         Console.WriteLine("RefA is not null");
   }
}
```

In the above program, if the garbage collector chooses to run the destructor of `A` before the destructor of `B`, then the output of this program might be:

```
Destruct instance of A
Destruct instance of B
A.F
RefA is not null
```

Note that although the instance of `A` was not in use and `A`'s destructor was run, it is still possible for methods of `A` (in this case, `F`) to be called from another destructor. Also, note that running of a destructor may cause an object to become usable from the mainline program again. In this case, the running of `B`'s destructor caused an instance of `A` that was previously not in use to

become accessible from the live reference `Test.RefA`. After the call to `WaitForPendingFinalizers`, the instance of `B` is eligible for collection, but the instance of `A` is not, because of the reference `Test.RefA`.

To avoid confusion and unexpected behavior, it is generally a good idea for destructors to only perform cleanup on data stored in their object's own fields, and not to perform any actions on referenced objects or static fields.

3.10 Execution order

Execution of a C# program proceeds such that the side effects of each executing thread are preserved at critical execution points. A side effect is defined as a read or write of a volatile field, a write to a non-volatile variable, a write to an external resource, and the throwing of an exception. The critical execution points at which the order of these side effects must be preserved are references to volatile fields (Section 10.4.3), lock statements (Section 8.12), and thread creation and termination. The execution environment is free to change the order of execution of a C# program, subject to the following constraints:

- Data dependence is preserved within a thread of execution. That is, the value of each variable is computed as if all statements in the thread were executed in original program order.
- Initialization ordering rules are preserved (Section 10.4.4 and Section 10.4.5).
- The ordering of side effects is preserved with respect to volatile reads and writes (Section 10.4.3). Additionally, the execution environment need not evaluate part of an expression if it can deduce that that expression's value is not used and that no needed side effects are produced (including any caused by calling a method or accessing a volatile field). When program execution is interrupted by an asynchronous event (such as an exception thrown by another thread), it is not guaranteed that the observable side effects are visible in the original program order.

4. Types

The types of the C# language are divided into two main categories: value types and reference types.

type:
  value-type
  reference-type

A third category of types, pointers, is available only in unsafe code. This is discussed further in Section A.2.

Value types differ from reference types in that variables of the value types directly contain their data, whereas variables of the reference types store references to their data, the latter being known as objects. With reference types, it is possible for two variables to reference the same object, and thus possible for operations on one variable to affect the object referenced by the other variable. With value types, the variables each have their own copy of the data, and it is not possible for operations on one to affect the other.

C#'s type system is unified such that a value of any type can be treated as an object. Every type in C# directly or indirectly derives from the `object` class type, and `object` is the ultimate base class of all types. Values of reference types are treated as objects simply by viewing the values as type `object`. Values of value types are treated as objects by performing boxing and unboxing operations (Section 4.3).

4.1 Value types

A value type is either a struct type or an enumeration type. C# provides a set of predefined struct types called the simple types. The simple types are identified through reserved words.

value-type:
  struct-type
  enum-type
struct-type:
  type-name
  simple-type
simple-type:
  numeric-type
  bool
numeric-type:
  integral-type
  floating-point-type
  decimal
integral-type:
  sbyte
  byte
  short
  ushort
  int
  uint
  long
  ulong
  char
floating-point-type:
  float
  double
enum-type:
  type-name

A variable of a value type always contains a value of that type. Unlike reference types, it is not possible for a value of a value type to be `null` or to reference an object of a more derived type.

Assignment to a variable of a value type creates a copy of the value being assigned. This differs from assignment to a variable of a reference type, which copies the reference but not the object identified by the reference.

4.1.1 The System.ValueType type

All value types implicitly inherit from the class `System.ValueType`, which, in turn, inherits from class `object`. It is not possible for any type to derive from a value type, and value types are thus implicitly sealed (Section 10.1.1.2).

Note that `System.ValueType` is not itself a *value-type*. Rather, it is a *class-type* from which all *value-types* are automatically derived.

4.1.2 Default constructors

All value types implicitly declare a public parameterless instance constructor called the default constructor. The default constructor returns a zero-initialized instance known as the default value for the value type:

- For all *simple-types*, the default value is the value produced by a bit pattern of all zeros:
  - For `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, and `ulong`, the default value is `0`.
  - For `char`, the default value is `'\x0000'`.
  - For `float`, the default value is `0.0f`.
  - For `double`, the default value is `0.0d`.
  - For `decimal`, the default value is `0.0m`.
  - For `bool`, the default value is `false`.
- For an *enum-type* `E`, the default value is `0`.
- For a *struct-type*, the default value is the value produced by setting all value type fields to their default value and all reference type fields to `null`.

Like any other instance constructor, the default constructor of a value type is invoked using the `new` operator. For efficiency reasons, this requirement is not intended to actually have the implementation generate a constructor call. In the example below, variables `i` and `j` are both initialized to zero.

```
class A
{
   void F() {
      int i = 0;
      int j = new int();
   }
}
```

Because every value type implicitly has a public parameterless instance constructor, it is not possible for a struct type to contain an explicit declaration of a parameterless constructor. A struct type is however permitted to declare parameterized instance constructors (Section 11.3.8).

4.1.3 Struct types

A struct type is a value type that can declare constants, fields, methods, properties, indexers, operators, instance constructors, static constructors, and nested types. Struct types are described in Section 11.

4.1.4 Simple types

C# provides a set of predefined struct types called the simple types. The simple types are identified through reserved words, but these reserved words are simply aliases for predefined struct types in the `System` namespace, as described in the table below.

Reserved word	Aliased type
`sbyte`	`System.SByte`
`byte`	`System.Byte`
`short`	`System.Int16`
`ushort`	`System.UInt16`
`int`	`System.Int32`
`uint`	`System.UInt32`
`long`	`System.Int64`
`ulong`	`System.UInt64`
`char`	`System.Char`
`float`	`System.Single`
`double`	`System.Double`
`bool`	`System.Boolean`
`decimal`	`System.Decimal`

Because a simple type aliases a struct type, every simple type has members. For example, `int` has the members declared in `System.Int32` and the members inherited from `System.Object`, and the following statements are permitted:

```
int i = int.MaxValue;         // System.Int32.MaxValue constant
string s = i.ToString();      // System.Int32.ToString() instance method
string t = 123.ToString();      // System.Int32.ToString() instance method
```

The simple types differ from other struct types in that they permit certain additional operations:

- Most simple types permit values to be created by writing literals (Section 2.4.4). For example, `123` is a literal of type `int` and `'a'` is a literal of type `char`. C# makes no provision for literals of struct types in general, and non-default values of other struct types are ultimately always created through instance constructors of those struct types.
- When the operands of an expression are all simple type constants, it is possible for the compiler to evaluate the expression

at compile-time. Such an expression is known as a constant-expression (Section 7.15). Expressions involving operators defined by other struct types are not considered to be constant expressions.

- Through `const` declarations it is possible to declare constants of the simple types (Section 10.3). It is not possible to have constants of other struct types, but a similar effect is provided by `static readonly` fields.
- Conversions involving simple types can participate in evaluation of conversion operators defined by other struct types, but a user-defined conversion operator can never participate in evaluation of another user-defined operator (Section 6.4.2).

4.1.5 Integral types

C# supports nine integral types: `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, and `char`. The integral types have the following sizes and ranges of values:

- The `sbyte` type represents signed 8-bit integers with values between –128 and 127.
- The `byte` type represents unsigned 8-bit integers with values between 0 and 255.
- The `short` type represents signed 16-bit integers with values between –32768 and 32767.
- The `ushort` type represents unsigned 16-bit integers with values between 0 and 65535.
- The `int` type represents signed 32-bit integers with values between –2147483648 and 2147483647.
- The `uint` type represents unsigned 32-bit integers with values between 0 and 4294967295.
- The `long` type represents signed 64-bit integers with values between –9223372036854775808 and 9223372036854775807.
- The `ulong` type represents unsigned 64-bit integers with values between 0 and 18446744073709551615.
- The `char` type represents unsigned 16-bit integers with values between 0 and 65535. The set of possible values for the `char` type corresponds to the Unicode character set. Although `char` has the same representation as `ushort`, not all operations permitted on one type are permitted on the other.

The integral-type unary and binary operators always operate with signed 32-bit precision, unsigned 32-bit precision, signed 64-bit precision, or unsigned 64-bit precision:

- For the unary `+` and `~` operators, the operand is converted to type `T`, where `T` is the first of `int`, `uint`, `long`, and `ulong` that can fully represent all possible values of the operand. The operation is then performed using the precision of type `T`, and the type of the result is `T`.
- For the unary `-` operator, the operand is converted to type `T`, where `T` is the first of `int` and `long` that can fully represent all possible values of the operand. The operation is then performed using the precision of type `T`, and the type of the result is `T`. The unary `-` operator cannot be applied to operands of type `ulong`.
- For the binary `+`, `-`, `*`, `/`, `%`, `&`, `^`, `|`, `==`, `!=`, `>`, `<`, `>=`, and `<=` operators, the operands are converted to type `T`, where `T` is the first of `int`, `uint`, `long`, and `ulong` that can fully represent all possible values of both operands. The operation is then performed using the precision of type `T`, and the type of the result is `T` (or `bool` for the relational operators). It is not permitted for one operand to be of type `long` and the other to be of type `ulong` with the binary operators.
- For the binary `<<` and `>>` operators, the left operand is converted to type `T`, where `T` is the first of `int`, `uint`, `long`, and `ulong` that can fully represent all possible values of the operand. The operation is then performed using the precision of type `T`, and the type of the result is `T`.

The `char` type is classified as an integral type, but it differs from the other integral types in two ways:

- There are no implicit conversions from other types to the `char` type. In particular, even though the `sbyte`, `byte`, and `ushort` types have ranges of values that are fully representable using the `char` type, implicit conversions from `sbyte`, `byte`, or `ushort` to `char` do not exist.
- Constants of the `char` type must be written as character-literals or as integer-literals in combination with a cast to type `char`. For example, `(char)10` is the same as `'\x000A'`.

The `checked` and `unchecked` operators and statements are used to control overflow checking for integral-type arithmetic operations and conversions (Section 7.5.12). In a `checked` context, an overflow produces a compile-time error or causes a `System.OverflowException` to be thrown. In an `unchecked` context, overflows are ignored and any high-order bits that do not fit in the destination type are discarded.

4.1.6 Floating point types

C# supports two floating point types: `float` and `double`. The `float` and `double` types are represented using the 32-bit single-precision and 64-bit double-precision IEEE 754 formats, which provide the following sets of values:

- Positive zero and negative zero. In most situations, positive zero and negative zero behave identically as the simple value zero, but certain operations distinguish between the two (Section 7.7.2).
- Positive infinity and negative infinity. Infinities are produced by such operations as dividing a non-zero number by zero. For example, `1.0 / 0.0` yields positive infinity, and `-1.0 / 0.0` yields negative infinity.
- The Not-a-Number value, often abbreviated NaN. NaN's are produced by invalid floating-point operations, such as dividing zero by zero.
- The finite set of non-zero values of the form $s \times m \times 2^e$, where s is 1 or −1, and m and e are determined by the particular floating-point type: For `float`, $0 < m < 2^{24}$ and −149 ≤ e ≤ 104, and for `double`, $0 < m < 2^{53}$ and −1075 ≤ e ≤ 970. Denormalized floating-point numbers are considered valid non-zero values.

The `float` type can represent values ranging from approximately 1.5×10^{-45} to 3.4×10^{38} with a precision of 7 digits.

The `double` type can represent values ranging from approximately 5.0×10^{-324} to 1.7×10^{308} with a precision of 15-16 digits.

If one of the operands of a binary operator is of a floating-point type, then the other operand must be of an integral type or a floating-point type, and the operation is evaluated as follows:

- If one of the operands is of an integral type, then that operand is converted to the floating-point type of the other operand.
- Then, if either of the operands is of type `double`, the other operand is converted to `double`, the operation is performed using at least `double` range and precision, and the type of the result is `double` (or `bool` for the relational operators).
- Otherwise, the operation is performed using at least `float` range and precision, and the type of the result is `float` (or `bool` for the relational operators).

The floating-point operators, including the assignment operators, never produce exceptions. Instead, in exceptional situations, floating-point operations produce zero, infinity, or NaN, as described below:

- If the result of a floating-point operation is too small for the destination format, the result of the operation becomes positive zero or negative zero.
- If the result of a floating-point operation is too large for the destination format, the result of the operation becomes positive infinity or negative infinity.
- If a floating-point operation is invalid, the result of the operation becomes NaN.
- If one or both operands of a floating-point operation is NaN, the result of the operation becomes NaN.

Floating-point operations may be performed with higher precision than the result type of the operation. For example, some hardware architectures support an "extended" or "long double" floating-point type with greater range and precision than the `double` type, and implicitly perform all floating-point operations using this higher precision type. Only at excessive cost in performance can such hardware architectures be made to perform floating-point operations with *less* precision, and rather than require an implementation to forfeit both performance and precision, C# allows a higher precision type to be used for all floating-point operations. Other than delivering more precise results, this rarely has any measurable effects. However, in expressions of the form `x * y / z`, where the multiplication produces a result that is outside the `double` range, but the subsequent division brings the temporary result back into the `double` range, the fact that the expression is evaluated in a higher range format may cause a finite result to be produced instead of an infinity.

4.1.7 The decimal type

The `decimal` type is a 128-bit data type suitable for financial and monetary calculations. The `decimal` type can represent values ranging from $1.0 \times 10-^{28}$ to approximately 7.9×10^{28} with 28-29 significant digits.

The finite set of values of type `decimal` are of the form $(-1)^s$ x *c* x 10^{-e}, where the sign *s* is 0 or 1, the coefficient *c* is given by 0 ≤ *c* < 2^{96}, and the scale *e* is such that 0 ≤ *e ≤* 28. The decimal type does not support signed zeros, infinities, or NaN's. A `decimal` is represented as a 96-bit integer scaled by a power of ten. For `decimal`s with an absolute value less than `1.0m`, the value is exact to the 28th decimal place, but no further. For `decimal`s with an absolute value greater than or equal to `1.0m`, the value is exact to 28 or 29 digits. Contrary to the `float` and `double` data types, decimal fractional numbers such as 0.1 can be represented exactly in the `decimal` representation. In the `float` and `double` representations, such numbers are often infinite fractions, making those representations more prone to round-off errors.

If one of the operands of a binary operator is of type `decimal`, then the other operand must be of an integral type or of type `decimal`. If an integral type operand is present, it is converted to `decimal` before the operation is performed.

The result of an operation on values of type `decimal` is that which would result from calculating an exact result (preserving scale, as defined for each operator) and then rounding to fit the representation. Results are rounded to the nearest representable value, and, when a result is equally close to two representable values, to the value that has an even number in the least significant digit position (this is known as "banker's rounding"). A zero result always has a sign of 0 and a scale of 0.

If a decimal arithmetic operation produces a value less than or equal to 5 x 10^{-29} in absolute value, the result of the operation becomes zero. If a `decimal` arithmetic operation produces a result that is too large for the `decimal` format, a `System.OverflowException` is thrown.

The `decimal` type has greater precision but smaller range than the floating-point types. Thus, conversions from the floating-point types to `decimal` might produce overflow exceptions, and conversions from `decimal` to the floating-point types might cause loss of precision. For these reasons, no implicit conversions exist between the floating-point types and `decimal`, and without explicit casts, it is not possible to mix floating-point and `decimal` operands in the same expression.

4.1.8 The bool type

The `bool` type represents Boolean logical quantities. The possible values of type `bool` are `true` and `false`.

No standard conversions exist between `bool` and other types. In particular, the `bool` type is distinct and separate from the integral types, and a `bool` value cannot be used in place of an integral value, and vice versa.

In the C and C++ languages, a zero integral or floating-point value or a null pointer can be converted to the Boolean value `false`, and a non-zero integral or floating-point value, or a non-null pointer can be converted to the Boolean value `true`. In C#, such conversions are accomplished by explicitly comparing an integral or floating-point value to zero, or by explicitly comparing an object reference to `null`.

4.1.9 Enumeration types

An enumeration type is a distinct type with named constants. Every enumeration type has an underlying type, which must be `byte`, `sbyte`, `short`, `ushort`, `int`, `uint`, `long` or `ulong`. The set of values of the enumeration type is the same as the set of values of the underlying type. Values of the enumeration type are not restricted to the values of the named constants. Enumeration types are defined through enumeration declarations (Section 14.1).

4.2 Reference types

A reference type is a class type, an interface type, an array type, or a delegate type.

reference-type:
  class-type
  interface-type
  array-type
  delegate-type
class-type:
  type-name
  object
  string
interface-type:
  type-name
array-type:
  non-array-type *rank-specifiers*
non-array-type:
  type
rank-specifiers:
  rank-specifier
  rank-specifiers *rank-specifier*
rank-specifier:
  [*dim-separators*$_{opt}$]
dim-separators:
  ,
  dim-separators ,
delegate-type:
  type-name

A reference type value is a reference to an instance of the type, the latter known as an object. The special value `null` is compatible with all reference types and indicates the absence of an instance.

4.2.1 Class types

A class type defines a data structure that contains data members (constants and fields), function members (methods, properties, events, indexers, operators, instance constructors, destructors and static constructors), and nested types. Class types support inheritance, a mechanism whereby derived classes can extend and specialize base classes. Instances of class types are created using object-creation-expressions (Section 7.5.10.1).

Class types are described in Section 10.

Certain predefined class types have special meaning in the C# language, as described in the following table.

Class type	Description
`System.Object`	The ultimate base class of all other types (Section 4.2.2).
`System.String`	The string type of the C# language (Section 4.2.3).
`System.ValueType`	The base class of all value types (Section 4.1.1).
`System.Enum`	The base class of all enum types (Section 14).
`System.Array`	The base class of all array types (Section 12).
`System.Delegate`	The base class of all delegate types (Section 15).
`System.Exception`	The base class of all exception types (Section 16).

4.2.2 The object type

The `object` class type is the ultimate base class of all other types. Every type in C# directly or indirectly derives from the `object` class type.

The keyword `object` is simply an alias for the predefined class `System.Object`.

4.2.3 The string type

The `string` type is a sealed class type that inherits directly from `object`. Instances of the `string` class represent Unicode character strings.

Values of the `string` type can be written as string literals (Section 2.4.4).

The keyword `string` is simply an alias for the predefined class `System.String`.

4.2.4 Interface types

An interface defines a contract. A class or struct that implements an interface must adhere to its contract. An interface may inherit from multiple base interfaces, and a class or struct may implement multiple interfaces.

Interface types are described in Section 13.

4.2.5 Array types

An array is a data structure that contains zero or more variables that are accessed through computed indices. The variables contained in an array, also called the elements of the array, are all of the same type, and this type is called the element type of the array.

Array types are described in Section 12.

4.2.6 Delegate types

A delegate is a data structure that refers to one or more methods, and for instance methods, it also refers to their corresponding object instances.

The closest equivalent of a delegate in C or C++ is a function pointer, but whereas a function pointer can only reference static functions, a delegate can reference both static and instance methods. In the latter case, the delegate stores not only a reference to the method's entry point, but also a reference to the object instance on which to invoke the method.

Delegate types are described in Section 15.

4.3 Boxing and unboxing

The concept of boxing and unboxing is central to C#'s type system. It provides a bridge between *value-type*s and *reference-type*s by permitting any value of a *value-type* to be converted to and from type `object`. Boxing and unboxing enables a unified view of the type system wherein a value of any type can ultimately be treated as an object.

4.3.1 Boxing conversions

A boxing conversion permits a *value-type* to be implicitly converted to a *reference-type*. The following boxing conversions exist:

- From any *value-type* (including any *enum-type*) to the type `object`.
- From any *value-type* (including any *enum-type*) to the type `System.ValueType`.
- From any *value-type* to any *interface-type* implemented by the *value-type*.
- From any *enum-type* to the type `System.Enum`.

Boxing a value of a *value-type* consists of allocating an object instance and copying the *value-type* value into that instance.

The actual process of boxing a value of a *value-type* is best explained by imagining the existence of a boxing class for that type. For any *value-type* `T`, the boxing class behaves as if it were declared as follows:

```
sealed class T_Box: System.ValueType
{
   T value;
   public T_Box(T t) {
      value = t;
   }
}
```

Boxing of a value `v` of type `T` now consists of executing the expression `new T_Box(v)`, and returning the resulting instance as a value of type `object`. Thus, the statements

```
int i = 123;
object box = i;
```

conceptually correspond to

```
int i = 123;
object box = new int_Box(i);
```

Boxing classes like `T_Box` and `int_Box` above do not actually exist and the dynamic type of a boxed value is not actually a class type. Instead, a boxed value of type `T` has the dynamic type `T`, and a dynamic type check using the `is` operator can simply reference type `T`. For example,

```
int i = 123;
object box = i;
if (box is int) {
   Console.Write("Box contains an int");
}
```

will output the string "`Box contains an int`" on the console.

A boxing conversion implies making a copy of the value being boxed. This is different from a conversion of a *reference-type* to type `object`, in which the value continues to reference the same instance and simply is regarded as the less derived type `object`. For example, given the declaration

```
struct Point
{
   public int x, y;
   public Point(int x, int y) {
      this.x = x;
      this.y = y;
   }
}
```

the following statements

```
Point p = new Point(10, 10);
object box = p;
p.x = 20;
Console.Write(((Point)box).x);
```

will output the value 10 on the console because the implicit boxing operation that occurs in the assignment of `p` to `box` causes the value of `p` to be copied. Had `Point` been declared a `class` instead, the value 20 would be output because `p` and `box` would reference the same instance.

4.3.2 Unboxing conversions

An unboxing conversion permits a *reference-type* to be explicitly converted to a *value-type*. The following unboxing conversions exist:

- From the type `object` to any *value-type* (including any *enum-type*).
- From the type `System.ValueType` to any *value-type* (including any *enum-type*).
- From any *interface-type* to any *value-type* that implements the *interface-type*.
- From the type `System.Enum` to any *enum-type*.

An unboxing operation consists of first checking that the object instance is a boxed value of the given *value-type*, and then copying the value out of the instance.

Referring to the imaginary boxing class described in the previous section, an unboxing conversion of an object `box` to a *value-type* `T` consists of executing the expression `((T_Box)box).value`. Thus, the statements

```
object box = 123;
int i = (int)box;
```

conceptually correspond to

```
object box = new int_Box(123);
int i = ((int_Box)box).value;
```

For an unboxing conversion to a given *value-type* to succeed at run-time, the value of the source operand must be a reference to an object that was previously created by boxing a value of that *value-type*. If the source operand is `null`, a `System.NullReferenceException` is thrown. If the source operand is a reference to an incompatible object, a `System.InvalidCastException` is thrown.

5. Variables

Variables represent storage locations. Every variable has a type that determines what values can be stored in the variable. C# is a type-safe language, and the C# compiler guarantees that values stored in variables are always of the appropriate type. The value of a variable can be changed through assignment or through use of the ++ and -- operators.

A variable must be definitely assigned (Section 5.3) before its value can be obtained.

As described in the following sections, variables are either initially assigned or initially unassigned. An initially assigned variable has a well-defined initial value and is always considered definitely assigned. An initially unassigned variable has no initial value. For an initially unassigned variable to be considered definitely assigned at a certain location, an assignment to the variable must occur in every possible execution path leading to that location.

5.1 Variable categories

C# defines seven categories of variables: static variables, instance variables, array elements, value parameters, reference parameters, output parameters, and local variables. The sections that follow describe each of these categories.

In the example

```
class A
{
   public static int x;
   int y;
   void F(int[] v, int a, ref int b, out int c) {
      int i = 1;
      c = a + b++;
   }
}
```

`x` is a static variable, `y` is an instance variable, `v[0]` is an array element, `a` is a value parameter, `b` is a reference parameter, `c` is an output parameter, and `i` is a local variable.

5.1.1 Static variables

A field declared with the `static` modifier is called a static variable. A static variable comes into existence before execution of the static constructor (Section 10.11) for its containing type, and ceases to exist when the associated application domain ceases to exist.

The initial value of a static variable is the default value (Section 5.2) of the variable's type.

For purposes of definite assignment checking, a static variable is considered initially assigned.

5.1.2 Instance variables

A field declared without the `static` modifier is called an instance variable.

5.1.2.1 Instance variables in classes

An instance variable of a class comes into existence when a new instance of that class is created, and ceases to exist when there are no references to that instance and the instance's destructor (if any) has executed.

The initial value of an instance variable of a class is the default value (Section 5.2) of the variable's type.

For the purpose of definite assignment checking, an instance variable of a class is considered initially assigned.

5.1.2.2 Instance variables in structs

An instance variable of a struct has exactly the same lifetime as the struct variable to which it belongs. In other words, when a variable of a struct type comes into existence or ceases to exist, so too do the instance variables of the struct.

The initial assignment state of an instance variable of a struct is the same as that of the containing struct variable. In other words, when a struct variable is considered initially assigned, so too are its instance variables, and when a struct variable is considered initially unassigned, its instance variables are likewise unassigned.

5.1.3 Array elements

The elements of an array come into existence when an array instance is created, and cease to exist when there are no references to that array instance.

The initial value of each of the elements of an array is the default value (Section 5.2) of the type of the array elements.

For the purpose of definite assignment checking, an array element is considered initially assigned.

5.1.4 Value parameters

A parameter declared without a `ref` or `out` modifier is a value parameter.

A value parameter comes into existence upon invocation of the function member (method, instance constructor, accessor, or operator) (Section 7.4) to which the parameter belongs, and is initialized with the value of the argument given in the invocation. A value parameter ceases to exist upon return of the function member.

For the purpose of definite assignment checking, a value parameter is considered initially assigned.

5.1.5 Reference parameters

A parameter declared with a `ref` modifier is a reference parameter.

A reference parameter does not create a new storage location. Instead, a reference parameter represents the same storage location as the variable given as the argument in the function member invocation. Thus, the value of a reference parameter is always the same as the underlying variable.

The following definite assignment rules apply to reference parameters. Note the different rules for output parameters described in Section 5.1.6.

- A variable must be definitely assigned (Section 5.3) before it can be passed as a reference parameter in a function member invocation.
- Within a function member, a reference parameter is considered initially assigned.

Within an instance method or instance accessor of a struct type, the `this` keyword behaves exactly as a reference parameter of the struct type (Section 7.5.7).

5.1.6 Output parameters

A parameter declared with an `out` modifier is an output parameter.

An output parameter does not create a new storage location. Instead, an output parameter represents the same storage location as the variable given as the argument in the function member invocation. Thus, the value of an output parameter is always the same as the underlying variable.

The following definite assignment rules apply to output parameters. Note the different rules for reference parameters described in Section 5.1.5.

- A variable need not be definitely assigned before it can be passed as an output parameter in a function member invocation.
- Following the normal completion of a function member invocation, each variable that was passed as an output parameter is considered assigned in that execution path.
- Within a function member, an output parameter is considered initially unassigned.
- Every output parameter of a function member must be definitely assigned (Section 5.3) before the function member returns normally.

Within an instance constructor of a struct type, the `this` keyword behaves exactly as an output parameter of the struct type (Section 7.5.7).

5.1.7 Local variables

A local variable is declared by a local-variable-declaration, which may occur in a block, a for-statement, a switch-statement, or a using-statement.

The lifetime of a local variable is the portion of program execution during which storage is guaranteed to be reserved for it. This lifetime extends from entry into the block, for-statement, switch-statement, or using-statement with which it is associated, until execution of that block, for-statement, switch-statement, or using-statement ends in any way. (Entering an enclosed block or calling a method suspends, but does not end, execution of the current block, for-statement, switch-statement, or using-statement.) If the parent block, for-statement, switch-statement, or using-statement is entered recursively, a new instance of the local variable is created each time, and its local-variable-initializer, if any, is evaluated each time.

A local variable is not automatically initialized and thus has no default value. For the purpose of definite assignment checking, a local variable is considered initially unassigned. A *local-variable-declaration* may include a *local-variable-initializer*, in which case the variable is considered definitely assigned in its entire scope, except within the expression provided in the *local-variable-initializer*.

Within the scope of a local variable, it is a compile-time error to refer to that local variable in a textual position that precedes its *local-variable-declarator*.

The actual lifetime of a local variable is implementation-dependent. For example, a compiler might statically determine that a local variable in a block is only used for a small portion of that block. Using this analysis, the compiler could generate code that results in the variable's storage having a shorter lifetime than its containing block.

The storage referred to by a local reference variable is reclaimed independently of the lifetime of that local reference variable (Section 3.9).

A local variable is also declared by a foreach-statement and by a specific-catch-clause for a try-statement. For a foreach-statement, the local variable is an iteration variable (Section 8.8.4). For a specific-catch-clause, the local variable is an exception variable (Section 8.10). A local variable declared by a foreach-statement or specific-catch-clause is considered definitely assigned in its entire scope.

5.2 Default values

The following categories of variables are automatically initialized to their default values:

- Static variables.
- Instance variables of class instances.
- Array elements.

The default value of a variable depends on the type of the variable and is determined as follows:

- For a variable of a *value-type*, the default value is the same as the value computed by the *value-type*'s default constructor (Section 4.1.1).
- For a variable of a *reference-type*, the default value is `null`.

Initialization to default values is typically done by having the memory manager or garbage collector initialize memory to all-bits-zero before it is allocated for use. For this reason, it is convenient to use all-bits-zero to represent the null reference.

5.3 Definite assignment

At a given location in the executable code of a function member, a variable is said to be definitely assigned if the compiler can prove, by static flow analysis, that the variable has been automatically initialized or has been the target of at least one assignment. The rules of definite assignment are:

- An initially assigned variable (Section 5.3.1) is always considered definitely assigned.
- An initially unassigned variable (Section 5.3.2) is considered definitely assigned at a given location if all possible execution paths leading to that location contain at least one of the following:
    - A simple assignment (Section 7.13.1) in which the variable is the left operand.
    - An invocation expression (Section 7.5.5) or object creation expression (Section 7.5.10.1) that passes the variable as an output parameter.
    - For a local variable, a local variable declaration (Section 8.5) that includes a variable initializer.

The definite assignment states of instance variables of a struct-type variable are tracked individually as well as collectively. In additional to the rules above, the following rules apply to struct-type variables and their instance variables:

- An instance variable is considered definitely assigned if its containing struct-type variable is considered definitely assigned.
- A struct-type variable is considered definitely assigned if each of its instance variables is considered definitely assigned.

Definite assignment is a requirement in the following contexts:

- A variable must be definitely assigned at each location where its value is obtained. This ensures that undefined values never occur. The occurrence of a variable in an expression is considered to obtain the value of the variable, except when
    - the variable is the left operand of a simple assignment,
    - the variable is passed as an output parameter, or
    - the variable is a struct-type variable and occurs as the left operand of a member access.
- A variable must be definitely assigned at each location where it is passed as a reference parameter. This ensures that the function member being invoked can consider the reference parameter initially assigned.
- All output parameters of a function member must be definitely assigned at each location where the function member returns (through a `return` statement or through execution reaching the end of the function member body). This ensures that function members do not return undefined values in output parameters, thus enabling the compiler to consider a function member invocation that takes a variable as an output parameter equivalent to an assignment to the variable.
- The `this` variable of a *struct-type* instance constructor must be definitely assigned at each location where that instance constructor returns.

5.3.1 Initially assigned variables

The following categories of variables are classified as initially assigned:

- Static variables.
- Instance variables of class instances.
- Instance variables of initially assigned struct variables.
- Array elements.
- Value parameters.
- Reference parameters.
- Variables declared in a `catch` clause or a `foreach` statement.

5.3.2 Initially unassigned variables

The following categories of variables are classified as initially unassigned:

- Instance variables of initially unassigned struct variables.
- Output parameters, including the `this` variable of struct instance constructors.
- Local variables , except those declared in a catch clause or a foreach statement.

5.3.3 Precise rules for determining definite assignment

In order to determine that each used variable is definitely assigned, the compiler must use a process that is equivalent to the one described in this section.

The compiler processes the body of each function member that has one or more initially unassigned variables. For each initially unassigned variable *v*, the compiler determines a definite assignment state for *v* at each of the following points in the function member:

- At the beginning of each statement
- At the end point (Section 8.1) of each statement
- On each arc that transfers control to another statement or to the end point of a statement
- At the beginning of each expression
- At the end of each expression

The definite assignment state of *v* can be either:

- Definitely assigned. This indicates that on all possible control flows to this point, *v* has been assigned a value.
- Not definitely assigned. For the state of a variable at the end of an expression of type `bool`, the state of a variable that is not definitely assigned may (but does not necessarily) fall into one of the following sub-states:
- Definitely assigned after true expression. This state indicates that *v* is definitely assigned if the Boolean expression evaluated as true, but is not necessarily assigned if the Boolean expression evaluated as false.
- Definitely assigned after false expression. This state indicates that *v* is definitely assigned if the Boolean expression evaluated as false, but is not necessarily assigned if the Boolean expression evaluated as true.

The following rules govern how the state of a variable *v* is determined at each location.

5.3.3.1 General rules for statements

- *v* is not definitely assigned at the beginning of a function member body.
- *v* is definitely assigned at the beginning of any unreachable statement.
- The definite assignment state of *v* at the beginning of any other statement is determined by checking the definite assignment state of *v* on all control flow transfers that target the beginning of that statement. If (and only if) *v* is definitely assigned on all such control flow transfers, then *v* is definitely assigned at the beginning of the statement. The set of possible control flow transfers is determined in the same way as for checking statement reachability (Section 8.1).
- The definite assignment state of *v* at the end point of a block, `checked`, `unchecked`, `if`, `while`, `do`, `for`, `foreach`, `lock`, `using`, or `switch` statement is determined by checking the definite assignment state of *v* on all control flow transfers that target the end point of that statement. If *v* is definitely assigned on all such control flow transfers, then *v* is definitely assigned at the end point of the statement. Otherwise; *v* is not definitely assigned at the end point of the statement. The set of possible control flow transfers is determined in the same way as for checking statement reachability (Section 8.1).

5.3.3.2 Block statements, checked, and unchecked statements

The definite assignment state of *v* on the control transfer to the first statement of the statement list in the block (or to the end point of the block, if the statement list is empty) is the same as the definite assignment statement of *v* before the block, `checked`, or `unchecked` statement.

5.3.3.3 Expression statements

For an expression statement *stmt* that consists of the expression *expr*:

- *v* has the same definite assignment state at the beginning of *expr* as at the beginning of *stmt*.
- If *v* if definitely assigned at the end of *expr*, it is definitely assigned at the end point of *stmt*; otherwise; it is not definitely assigned at the end point of *stmt*.

5.3.3.4 Declaration statements

- If *stmt* is a declaration statement without initializers, then *v* has the same definite assignment state at the end point of *stmt* as at the beginning of *stmt*.
- If *stmt* is a declaration statement with initializers, then the definite assignment state for *v* is determined as if *stmt* were a statement list, with one assignment statement for each declaration with an initializer (in the order of declaration).

5.3.3.5 If statements

For an `if` statement *stmt* of the form:

```
if ( expr ) then-stmt else else-stmt
```

- *v* has the same definite assignment state at the beginning of *expr* as at the beginning of *stmt*.
- If *v* is definitely assigned at the end of *expr*, then it is definitely assigned on the control flow transfer to *then-stmt* and to either *else-stmt* or to the end-point of *stmt* if there is no else clause.
- If *v* has the state "definitely assigned after true expression" at the end of *expr*, then it is definitely assigned on the control flow transfer to *then-stmt*, and not definitely assigned on the control flow transfer to either *else-stmt* or to the end-point of *stmt* if there is no else clause.
- If *v* has the state "definitely assigned after false expression" at the end of *expr*, then it is definitely assigned on the control flow transfer to *else-stmt*, and not definitely assigned on the control flow transfer to *then-stmt*. It is definitely assigned at the end-point of *stmt* if and only if it is definitely assigned at the end-point of *then-stmt*.
- Otherwise, *v* is considered not definitely assigned on the control flow transfer to either the *then-stmt* or *else-stmt*, or to the end-point of *stmt* if there is no else clause.

5.3.3.6 Switch statements

In a `switch` statement *stmt* with a controlling expression *expr*:

- The definite assignment state of *v* at the beginning of *expr* is the same as the state of *v* at the beginning of *stmt*.
- The definite assignment state of *v* on the control flow transfer to a reachable switch block statement list is the same as the definite assignment state of *v* at the end of *expr*.

5.3.3.7 While statements

For a `while` statement *stmt* of the form:

```
while ( expr ) while-body
```

- *v* has the same definite assignment state at the beginning of *expr* as at the beginning of *stmt*.
- If *v* is definitely assigned at the end of *expr*, then it is definitely assigned on the control flow transfer to *while-body* and to the end point of *stmt*.
- If *v* has the state "definitely assigned after true expression" at the end of *expr*, then it is definitely assigned on the control flow transfer to *while-body*, but not definitely assigned at the end-point of *stmt*.
- If *v* has the state "definitely assigned after false expression" at the end of *expr*, then it is definitely assigned on the control flow transfer to the end point of *stmt*, but not definitely assigned on the control flow transfer to *while-body*.

5.3.3.8 Do statements

For a `do` statement *stmt* of the form:

```
do do-body while ( expr ) ;
```

- *v* has the same definite assignment state on the control flow transfer from the beginning of *stmt* to *do-body* as at the beginning of *stmt*.
- *v* has the same definite assignment state at the beginning of *expr* as at the end point of *do-body*.
- If *v* is definitely assigned at the end of *expr*, then it is definitely assigned on the control flow transfer to the end point of *stmt*.
- If *v* has the state "definitely assigned after false expression" at the end of *expr*, then it is definitely assigned on the control flow transfer to the end point of *stmt*.

5.3.3.9 For statements

Definite assignment checking for a `for` statement of the form:

```
for ( for-initializer ; for-condition ; for-iterator ) embedded-statement
```

is done as if the statement were written:

```
{
   for-initializer ;
   while ( for-condition ) {
      embedded-statement ;
      for-iterator ;
   }
}
```

If the *for-condition* is omitted from the `for` statement, then evaluation of definite assignment proceeds as if *for-condition* were replaced with `true` in the above expansion.

5.3.3.10 Break, continue, and goto statements

The definite assignment state of *v* on the control flow transfer caused by a `break`, `continue`, or `goto` statement is the same as the definite assignment state of *v* at the beginning of the statement.

5.3.3.11 Throw statements

For a statement *stmt* of the form

```
throw expr ;
```

The definite assignment state of *v* at the beginning of *expr* is the same as the definite assignment state of *v* at the beginning of *stmt*.

5.3.3.12 Return statements

For a statement *stmt* of the form

```
return expr ;
```

- The definite assignment state of *v* at the beginning of *expr* is the same as the definite assignment state of *v* at the beginning of *stmt*.
- If *v* is an output parameter, then it must be definitely assigned either:
- after *expr*
- or at the end of the `finally` block of a `try-finally` or `try-catch-finally` that encloses the `return` statement.

5.3.3.13 Try-catch statements

For a statement *stmt* of the form:

```
try try-block
catch(...) catch-block-1
...
catch(...) catch-block-n
```

- The definite assignment state of *v* at the beginning of *try-block* is the same as the definite assignment state of *v* at the beginning of *stmt*.
- The definite assignment state of *v* at the beginning of *catch-block-i* (for any *i*) is the same as the definite assignment state of *v* at the beginning of *stmt*.
- The definite assignment state of *v* at the end-point of *stmt* is definitely assigned if (and only if) *v* is definitely assigned at the end-point of *try-block* and every *catch-block-i* (for every *i* from 1 to *n*).

5.3.3.14 Try-finally statements

For a `try` statement *stmt* of the form:

```
try try-block finally finally-block
```

- The definite assignment state of *v* at the beginning of *try-block* is the same as the definite assignment state of *v* at the beginning of *stmt*.
- The definite assignment state of *v* at the beginning of *finally-block* is the same as the definite assignment state of *v* at the beginning of *stmt*.
- The definite assignment state of *v* at the end-point of *stmt* is definitely assigned if (and only if) at least one of the following is true:
    - *v* is definitely assigned at the end-point of *try-block*
    - *v* is definitely assigned at the end-point of *finally-block*

If a control flow transfer (for example, a `goto` statement) is made that begins within *try-block*, and ends outside of *try-block*, then *v* is also considered definitely assigned on that control flow transfer if *v* is definitely assigned at the end-point of *finally-block*. (This is not an only if — if *v* is definitely assigned for another reason on this control flow transfer, then it is still considered definitely assigned.)

5.3.3.15 Try-catch-finally statements

Definite assignment analysis for a `try-catch-finally` statement of the form:

```
try try-block
catch(...) catch-block-1
...
catch(...) catch-block-n
finally finally-block
```

is done as if the statement were a `try-finally` statement enclosing a `try-catch` statement:

```
try {
   try try-block
   catch(...) catch-block-1
   ...
   catch(...) catch-block-n
}
finally finally-block
```

The following example demonstrates how the different blocks of a `try` statement (Section 8.10) affect definite assignment.

```
class A
{
   static void F() {
      int i, j;
      try {
         goto LABEL;
         // neither i nor j definitely assigned
         i = 1;
         // i definitely assigned
      }
      catch {
         // neither i nor j definitely assigned
         i = 3;
         // i definitely assigned
      }
      finally {
         // neither i nor j definitely assigned
         j = 5;
         // j definitely assigned
      }
      // i and j definitely assigned
   LABEL;
      // j definitely assigned

   }
}
```

5.3.3.16 Foreach statements

For a `foreach` statement *stmt* of the form:

```
foreach ( type identifier in expr ) embedded-statement
```

- The definite assignment state of *v* at the beginning of *expr* is the same as the state of *v* at the beginning of *stmt*.
- The definite assignment state of *v* on the control flow transfer to *embedded-statement* or to the end point of *stmt* is the same as the state of *v* at the end of *expr*.

5.3.3.17 Using statements

For a `using` statement *stmt* of the form:

```
using ( resource-acquisition ) embedded-statement
```

- The definite assignment state of *v* at the beginning of *resource-acquisition* is the same as the state of *v* at the beginning of *stmt*.
- The definite assignment state of *v* on the control flow transfer to *embedded-statement* is the same as the state of *v* at the end of *resource-acquisition*.

5.3.3.18 Lock statements

For a `lock` statement *stmt* of the form:

```
lock ( expr ) embedded-statement
```

- The definite assignment state of *v* at the beginning of *expr* is the same as the state of *v* at the beginning of *stmt*.
- The definite assignment state of *v* on the control flow transfer to *embedded-statement* is the same as the state of *v* at the end of *expr*.

5.3.3.19 General rules for simple expressions

The following rule applies to these kinds of expressions: literals (Section 7.5.1), simple names (Section 7.5.2), member access expressions (Section 7.5.4), non-indexed base access expressions (Section 7.5.8), and `typeof` expressions (Section 7.5.11).

The definite assignment state of *v* at the end of such an expression is the same as the definite assignment state of *v* at the beginning of the expression.

5.3.3.20 General rules for expressions with embedded expressions

The following rules apply to these kinds of expressions: parenthesized expressions (Section 7.5.3), element access expressions (Section 7.5.6), base access expressions with indexing (Section 7.5.8), increment and decrement expressions (Section 7.5.9, Section 7.6.5), cast expressions (Section 7.6.6), unary `+`, `-`, `~`, `*` expressions, binary `+`, `-`, `*`, `/`, `%`, `<<`, `>>`, `<`, `<=`, `>`, `>=`, `==`, `!=`, `is`, `as`, `&`, `|`, `^` expressions (Section 7.7, Section 7.8, Section 7.9, Section 7.10), compound assignment expressions (Section 7.13.2), `checked` and `unchecked` expressions (Section 7.5.12), array and delegate creation expressions (Section 7.5.10).

Each of these expressions has one or more sub-expressions that are unconditionally evaluated in a fixed order. For example, the binary `%` operator evaluates the left hand side of the operator, then the right hand side. An indexing operation evaluates the indexed expression, and then evaluates each of the index expressions, in order from left to right. For an expression *expr*, which has sub-expressions *expr1*, *expr2*, ..., *exprn*, evaluated in that order:

- The definite assignment state of *v* at the beginning of *expr1* is the same as the definite assignment state at the beginning of *expr*.
- The definite assignment state of *v* at the beginning of *expri* (*i* greater than one) is the same as the definite assignment state at the end of *expri-1*.
- The definite assignment state of *v* at the end of *expr* is the same as the definite assignment state at the end of *exprn*.

5.3.3.21 Invocation expressions and object creation expressions

For an invocation expression *expr* of the form:

```
primary-expression ( arg1 , arg2 , ... , argn )
```

or an object creation expression of the form:

```
new type ( arg1 , arg2 , ... , argn )
```

- For an invocation expression, the definite assignment state of *v* before *primary-expression* is the same as the state of *v* before *expr*.
- For an invocation expression, the definite assignment state of *v* before *arg1* is the same as the state of *v* after *primary-expression*.
- For an object creation expression, the definite assignment state of *v* before *arg1* is the same as the state of *v* before *expr*.
- For each argument *argi*, the definite assignment state of *v* after *argi* is determined by the normal expression rules, ignoring any `ref` or `out` modifiers.
- For each argument *argi* for any *i* greater than one, the definite assignment state of *v* before *argi* is the same as the state of *v* after *argi-1*.
- If the variable *v* is passed as an `out` argument (i.e., an argument of the form "`out` *v*") in any of the arguments, then the state of *v* after *expr* is definitely assigned. Otherwise; the state of *v* after *expr* is the same as the state of *v* after *argn*.

5.3.3.22 Simple assignment expressions

For an expression *expr* of the form *w* = *expr-rhs*:

- The definite assignment state of *v* before *expr-rhs* is the same as the definite assignment state of *v* before *expr*.
- If *w* is the same variable as *v*, then the definite assignment state of *v* after *expr* is definitely assigned. Otherwise, the definite assignment state of *v* after *expr* is the same as the definite assignment state of *v* after *expr-rhs*.

5.3.3.23 && expressions

For an expression *expr* of the form *expr-first* `&&` *expr-second*:

- The definite assignment state of *v* before *expr-first* is the same as the definite assignment state of *v* before *expr*.
- The definite assignment state of *v* before *expr-second* is definitely assigned if the state of *v* after *expr-first* is either definitely assigned or "definitely assigned after true expression". Otherwise, it is not definitely assigned.
- The definite assignment state of *v* after *expr* is determined by:
- If the state of *v* after *expr-first* is definitely assigned, then the state of *v* after *expr* is definitely assigned.
- Otherwise, if the state of *v* after *expr-second* is definitely assigned, and the state of *v* after *expr-first* is "definitely assigned after false expression", then the state of *v* after *expr* is definitely assigned.
- Otherwise, if the state of *v* after *expr-second* is definitely assigned or "definitely assigned after true expression", then the state of *v* after *expr* is "definitely assigned after true expression".
- Otherwise, if the state of *v* after *expr-first* is "definitely assigned after false expression", and the state of *v* after *expr-second* is "definitely assigned after false expression", then the state of *v* after *expr* is "definitely assigned after false expression".
- Otherwise, the state of *v* after *expr* is not definitely assigned.

In the example

```
class A
{
   static void F(int x, int y) {
      int i;
      if (x >= 0 && (i = y) >= 0) {
         // i definitely assigned
      }
      else {
         // i not definitely assigned
      }
      // i not definitely assigned
   }
}
```

the variable `i` is considered definitely assigned in one of the embedded statements of an `if` statement but not in the other. In the `if` statement in method `F`, the variable `i` is definitely assigned in the first embedded statement because execution of the expression `(i = y)` always precedes execution of this embedded statement. In contrast, the variable `i` is not definitely assigned in the second embedded statement, since `x >= 0` might have tested false, resulting in the variable `i` being unassigned.

5.3.3.24 || expressions

For an expression *expr* of the form *expr-first* `||` *expr-second*:

- The definite assignment state of *v* before *expr-first* is the same as the definite assignment state of *v* before *expr*.
- The definite assignment state of *v* before *expr-second* is definitely assigned if the state of *v* after *expr-first* is either definitely assigned or "definitely assigned after false expression". Otherwise, it is not definitely assigned.
- The definite assignment statement of *v* after *expr* is determined by:
- If the state of *v* after *expr-first* is definitely assigned, then the state of *v* after *expr* is definitely assigned.
- Otherwise, if the state of *v* after *expr-second* is definitely assigned, and the state of *v* after *expr-first* is "definitely assigned after true expression", then the state of *v* after *expr* is definitely assigned.
- Otherwise, if the state of *v* after *expr-second* is definitely assigned or "definitely assigned after false expression", then the state of *v* after *expr* is "definitely assigned after false expression".
- Otherwise, if the state of *v* after *expr-first* is "definitely assigned after true expression", and the state of *v* after *expr-second* is "definitely assigned after true expression", then the state of *v* after *expr* is "definitely assigned after true expression".
- Otherwise, the state of *v* after *expr* is not definitely assigned.

In the example

```
class A
{
   static void G(int x, int y) {
      int i;
      if (x >= 0 || (i = y) >= 0) {
         // i not definitely assigned
      }
      else {
         // i definitely assigned
      }
      // i not definitely assigned
   }
}
```

the variable `i` is considered definitely assigned in one of the embedded statements of an `if` statement but not in the other. In the `if` statement in method `G`, the variable `i` is definitely assigned in the second embedded statement because execution of the expression `(i = y)` always precedes execution of this embedded statement. In contrast, the variable `i` is not definitely assigned in the first embedded statement, since `x >= 0` might have tested true, resulting in the variable `i` being unassigned.

5.3.3.25 ! expressions

For an expression *expr* of the form ！*expr-operand*:

- The definite assignment state of *v* before *expr-operand* is the same as the definite assignment state of *v* before *expr*.
- The definite assignment state of *v* after *expr* is determined by:
- If the state of *v* after *expr-operand* is definitely assigned, then the state of *v* after *expr* is definitely assigned.
- If the state of *v* after *expr-operand* is not definitely assigned, then the state of *v* after *expr* is not definitely assigned.
- If the state of *v* after *expr-operand* is "definitely assigned after false expression", then the state of *v* after *expr* is "definitely assigned after true expression".
- If the state of *v* after *expr-operand* is "definitely assigned after true expression", then the state of *v* after *expr* is "definitely assigned after false expression".

5.3.3.26 ?: expressions

For an expression *expr* of the form *expr-cond* ? *expr-true* : *expr-false*:

- The definite assignment state of *v* before *expr-cond* is the same as the state of *v* before *expr*.
- The definite assignment state of *v* before *expr-true* is definitely assigned if and only if the state of *v* after *expr-cond* is definitely assigned or "definitely assigned after true expression".

The definite assignment state of *v* before *expr-false* is definitely assigned if and only if the state of *v* after *expr-cond* is definitely assigned or "definitely assigned after false expression".

The definite assignment state of *v* after *expr* is determined by:

- If *expr-cond* is a constant expression (Section 7.15) with value true then the state of *v* after *expr* is the same as the state of *v* after *expr-true*.
- Otherwise, if *expr-cond* is a constant expression (Section 7.15) with value false then the state of *v* after *expr* is the same as the state of *v* after *expr-false*.
- Otherwise, if the state of *v* after *expr-true* is definitely assigned and the state of *v* after *expr-false* is definitely assigned, then the state of *v* after *expr* is definitely assigned.
- Otherwise, the state of *v* after *expr* is not definitely assigned.

5.4 Variable references

A *variable-reference* is an *expression* that is classified as a variable. A *variable-reference* denotes a storage location that can be accessed both to fetch the current value and to store a new value.

variable-reference:
  expression

In C and C++, a *variable-reference* is known as an *lvalue*.

5.5 Atomicity of variable references

Reads and writes of the following data types are atomic: `bool`, `char`, `byte`, `sbyte`, `short`, `ushort`, `uint`, `int`, `float`, and reference types. In addition, reads and writes of enum types with an underlying type in the previous list are also atomic. Reads and writes of other types, including `long`, `ulong`, `double`, and `decimal`, as well as user-defined types, are not guaranteed to be atomic. Aside from the library functions designed for that purpose, there is no guarantee of atomic read-modify-write, such as in the case of increment or decrement.

6. Conversions

A conversion enables an expression of one type to be treated as another type. Conversions can be implicit or explicit, and this determines whether an explicit cast is required. For instance, the conversion from type `int` to type `long` is implicit, so expressions of type `int` can implicitly be treated as type `long`. The opposite conversion, from type `long` to type `int`, is explicit and so an explicit cast is required.

```
int a = 123;
long b = a;         // implicit conversion from int to long
int c = (int) b;   // explicit conversion from long to int
```

Some conversions are defined by the language. Programs may also define their own conversions (Section 6.4).

6.1 Implicit conversions

The following conversions are classified as implicit conversions:

- Identity conversions
- Implicit numeric conversions
- Implicit enumeration conversions
- Implicit reference conversions
- Boxing conversions
- Implicit constant expression conversions
- User-defined implicit conversions

Implicit conversions can occur in a variety of situations, including function member invocations (Section 7.4.3), cast expressions (Section 7.6.6), and assignments (Section 7.13).

The pre-defined implicit conversions always succeed and never cause exceptions to be thrown. Properly designed user-defined implicit conversions should exhibit these characteristics as well.

6.1.1 Identity conversion

An identity conversion converts from any type to the same type. This conversion exists only such that an entity that already has a required type can be said to be convertible to that type.

6.1.2 Implicit numeric conversions

The implicit numeric conversions are:

- From `sbyte` to `short`, `int`, `long`, `float`, `double`, or `decimal`.
- From `byte` to `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `float`, `double`, or `decimal`.
- From `short` to `int`, `long`, `float`, `double`, or `decimal`.
- From `ushort` to `int`, `uint`, `long`, `ulong`, `float`, `double`, or `decimal`.
- From `int` to `long`, `float`, `double`, or `decimal`.
- From `uint` to `long`, `ulong`, `float`, `double`, or `decimal`.
- From `long` to `float`, `double`, or `decimal`.
- From `ulong` to `float`, `double`, or `decimal`.
- From `char` to `ushort`, `int`, `uint`, `long`, `ulong`, `float`, `double`, or `decimal`.
- From `float` to `double`.

Conversions from `int`, `uint`, `long`, or `ulong` to `float` and from `long` or `ulong` to `double` may cause a loss of precision, but will never cause a loss of magnitude. The other implicit numeric conversions never lose any information.

There are no implicit conversions to the `char` type, so values of the other integral types do not automatically convert to the `char` type.

6.1.3 Implicit enumeration conversions

An implicit enumeration conversion permits the *decimal-integer-literal* `0` to be converted to any *enum-type*.

6.1.4 Implicit reference conversions

The implicit reference conversions are:

- From any *reference-type* to `object`.
- From any *class-type* `S` to any *class-type* `T`, provided `S` is derived from `T`.
- From any *class-type* `S` to any *interface-type* `T`, provided `S` implements `T`.
- From any *interface-type* `S` to any *interface-type* `T`, provided `S` is derived from `T`.
- From an *array-type* `S` with an element type `SE` to an *array-type* `T` with an element type `TE`, provided all of the following are true:
    - `S` and `T` differ only in element type. In other words, `S` and `T` have the same number of dimensions.
    - Both `SE` and `TE` are *reference-type*s.
    - An implicit reference conversion exists from `SE` to `TE`.
- From any *array-type* to `System.Array`.
- From any *delegate-type* to `System.Delegate`.
- From the null type to any *reference-type*.

The implicit reference conversions are those conversions between *reference-type*s that can be proven to always succeed, and therefore require no checks at run-time.

Reference conversions, implicit or explicit, never change the referential identity of the object being converted. In other words, while a reference conversion may change the type of the reference, it never changes the type or value of the object being referred to.

6.1.5 Boxing conversions

A boxing conversion permits a *value-type* to be implicitly converted to a *reference-type*. Boxing a value of a *value-type* consists of allocating an object instance and copying the *value-type* value into that instance.

Boxing conversions are described further in Section 4.3.1.

6.1.6 Implicit constant expression conversions

An implicit constant expression conversion permits the following conversions:

- A *constant-expression* (Section 7.15) of type `int` can be converted to type `sbyte`, `byte`, `short`, `ushort`, `uint`, or `ulong`, provided the value of the *constant-expression* is within the range of the destination type.
- A *constant-expression* of type `long` can be converted to type `ulong`, provided the value of the *constant-expression* is not negative.

6.1.7 User-defined implicit conversions

A user-defined implicit conversion consists of an optional standard implicit conversion, followed by execution of a user-defined implicit conversion operator, followed by another optional standard implicit conversion. The exact rules for evaluating user-defined conversions are described in Section 6.4.3.

6.2 Explicit conversions

The following conversions are classified as explicit conversions:

- All implicit conversions.
- Explicit numeric conversions.
- Explicit enumeration conversions.
- Explicit reference conversions.
- Explicit interface conversions.
- Unboxing conversions.
- User-defined explicit conversions.

Explicit conversions can occur in cast expressions (Section 7.6.6).

The set of explicit conversions includes all implicit conversions. This means that redundant cast expressions are allowed.

The explicit conversions that are not implicit conversions are conversions that cannot be proven to always succeed, conversions that are known to possibly lose information, and conversions across domains of types sufficiently different to merit explicit notation.

6.2.1 Explicit numeric conversions

The explicit numeric conversions are the conversions from a numeric-type to another numeric-type for which an implicit numeric conversion (Section 6.1.2) does not already exist:

- From `sbyte` to `byte`, `ushort`, `uint`, `ulong`, or `char`.
- From `byte` to `sbyte` and `char`.
- From `short` to `sbyte`, `byte`, `ushort`, `uint`, `ulong`, or `char`.
- From `ushort` to `sbyte`, `byte`, `short`, or `char`.
- From `int` to `sbyte`, `byte`, `short`, `ushort`, `uint`, `ulong`, or `char`.
- From `uint` to `sbyte`, `byte`, `short`, `ushort`, `int`, or `char`.
- From `long` to `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `ulong`, or `char`.
- From `ulong` to `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, or `char`.
- From `char` to `sbyte`, `byte`, or `short`.
- From `float` to `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, or `decimal`.
- From `double` to `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, or `decimal`.
- From `decimal` to `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, or `double`.

Because the explicit conversions include all implicit and explicit numeric conversions, it is always possible to convert from any numeric-type to any other numeric-type using a cast expression (Section 7.6.6).

The explicit numeric conversions possibly lose information or possibly cause exceptions to be thrown. An explicit numeric conversion is processed as follows:

- For a conversion from an integral type to another integral type, the processing depends on the overflow checking context (Section 7.5.12) in which the conversion takes place:
  - In a `checked` context, the conversion succeeds if the value of the source operand is within the range of the destination type, but throws a `System.OverflowException` if the value of the source operand is outside the range of the destination type.
  - In an `unchecked` context, the conversion always succeeds, and proceeds as follows.

    If the source type is larger than the destination type, then the source value is truncated by discarding its "extra" most significant bits. The result is then treated as a value of the destination type.

    If the source type is smaller than the destination type, then the source value is either sign-extended or zero-extended so that it is the same size as the destination type. Sign-extension is used if the source type is signed; zero-extension is used if the source type is unsigned. The result is then treated as a value of the destination type.

    If the source type is the same size as the destination type, then the source value is treated as a value of the destination type.
- For a conversion from `decimal` to an integral type, the source value is rounded towards zero to the nearest integral value, and this integral value becomes the result of the conversion. If the resulting integral value is outside the range of the destination type, a `System.OverflowException` is thrown.
- For a conversion from `float` or `double` to an integral type, the processing depends on the overflow checking context (Section 7.5.12) in which the conversion takes place:
  - In a `checked` context, the conversion proceeds as follows:

    If the value of the operand is NaN or infinite, a `System.OverflowException` is thrown.

    Otherwise, the source operand is rounded towards zero to the nearest integral value. If this integral value is within the range of the destination type then this value is the result of the conversion.

    Otherwise, a `System.OverflowException` is thrown.
  - In an unchecked context, the conversion always succeeds, and proceeds as follows.

    If the value of the operand is NaN or infinite, the result of the conversion is an unspecified value of the destination type.

Otherwise, the source operand is rounded towards zero to the nearest integral value. If this integral value is within the range of the destination type then this value is the result of the conversion.

Otherwise, the result of the conversion is an unspecified value of the destination type.

- For a conversion from `double` to `float`, the `double` value is rounded to the nearest `float` value. If the `double` value is too small to represent as a `float`, the result becomes positive zero or negative zero. If the `double` value is too large to represent as a `float`, the result becomes positive infinity or negative infinity. If the `double` value is NaN, the result is also NaN.
- For a conversion from `float` or `double` to `decimal`, the source value is converted to `decimal` representation and rounded to the nearest number after the 28th decimal place if required (Section 4.1.7). If the source value is too small to represent as a `decimal`, the result becomes zero. If the source value is NaN, infinity, or too large to represent as a `decimal`, a `System.OverflowException` is thrown.
- For a conversion from `decimal` to `float` or `double`, the `decimal` value is rounded to the nearest `double` or `float` value. While this conversion may lose precision, it never causes an exception to be thrown.

6.2.2 Explicit enumeration conversions

The explicit enumeration conversions are:

- From `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, or `decimal` to any *enum-type*.
- From any *enum-type* to `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, or `decimal`.
- From any *enum-type* to any other *enum-type*.

An explicit enumeration conversion between two types is processed by treating any participating *enum-type* as the underlying type of that *enum-type*, and then performing an implicit or explicit numeric conversion between the resulting types. For example, given an *enum-type* `E` with and underlying type of `int`, a conversion from `E` to `byte` is processed as an explicit numeric conversion (Section 6.2.1) from `int` to `byte`, and a conversion from `byte` to `E` is processed as an implicit numeric conversion (Section 6.1.2) from `byte` to `int`.

6.2.3 Explicit reference conversions

The explicit reference conversions are:

- From `object` to any other *reference-type*.
- From any *class-type* `S` to any *class-type* `T`, provided `S` is a base class of `T`.
- From any *class-type* `S` to any *interface-type* `T`, provided `S` is not sealed and provided `S` does not implement `T`.
- From any *interface-type* `S` to any *class-type* `T`, provided `T` is not sealed or provided `T` implements `S`.
- From any *interface-type* `S` to any *interface-type* `T`, provided `S` is not derived from `T`.
- From an *array-type* `S` with an element type `SE` to an *array-type* `T` with an element type `TE`, provided all of the following are true:
- `S` and `T` differ only in element type. In other words, `S` and `T` have the same number of dimensions.
- Both `SE` and `TE` are *reference-types*.
- An explicit reference conversion exists from `SE` to `TE`.
- From `System.Array` and the interfaces it implements to any *array-type*.
- From `System.Delegate` and the interfaces it implements to any *delegate-type*.

The explicit reference conversions are those conversions between *reference-types* that require run-time checks to ensure they are correct.

For an explicit reference conversion to succeed at run-time, the value of the source operand must be `null`, or the actual type of the object referenced by the source operand must be a type that can be converted to the destination type by an implicit reference conversion (Section 6.1.4). If an explicit reference conversion fails, a `System.InvalidCastException` is thrown.

Reference conversions, implicit or explicit, never change the referential identity of the object being converted. In other words, while a reference conversion may change the type of the reference, it never changes the type or value of the object being referred to.

6.2.4 Unboxing conversions

An unboxing conversion permits a *reference-type* to be explicitly converted to a *value-type*. An unboxing operation consists of first checking that the object instance is a boxed value of the given *value-type*, and then copying the value out of the instance.

Unboxing conversions are described further in Section 4.3.2.

6.2.5 User-defined explicit conversions

A user-defined explicit conversion consists of an optional standard explicit conversion, followed by execution of a user-defined implicit or explicit conversion operator, followed by another optional standard explicit conversion. The exact rules for evaluating user-defined conversions are described in Section 6.4.4.

6.3 Standard conversions

The standard conversions are those pre-defined conversions that can occur as part of a user-defined conversion.

6.3.1 Standard implicit conversions

The following implicit conversions are classified as standard implicit conversions:

- Identity conversions (Section 6.1.1).
- Implicit numeric conversions (Section 6.1.2).
- Implicit reference conversions (Section 6.1.4).
- Boxing conversions (Section 6.1.5).
- Implicit constant expression conversions (Section 6.1.6).

The standard implicit conversions specifically exclude user-defined implicit conversions.

6.3.2 Standard explicit conversions

The standard explicit conversions are all standard implicit conversions plus the subset of the explicit conversions for which an opposite standard implicit conversion exists. In other words, if a standard implicit conversion exists from a type `A` to a type `B`, then a standard explicit conversion exists from type `A` to type `B` and from type `B` to type `A`.

6.4 User-defined conversions

C# allows the pre-defined implicit and explicit conversions to be augmented by user-defined conversions. User-defined conversions are introduced by declaring conversion operators (Section 10.9.3) in class and struct types.

6.4.1 Permitted user-defined conversions

C# permits only certain user-defined conversions to be declared. In particular, it is not possible to redefine an already existing implicit or explicit conversion. A class or struct is permitted to declare a conversion from a source type `S` to a target type `T` only if all of the following are true:

- `S` and `T` are different types.
- Either `S` or `T` is the class or struct type in which the operator declaration takes place.
- Neither `S` nor `T` is `object` or an *interface-type*.
- `T` is not a base class of `S`, and `S` is not a base class of `T`.

The restrictions that apply to user-defined conversions are discussed further in Section 10.9.3.

6.4.2 Evaluation of user-defined conversions

A user-defined conversion converts a value from its type, called the source type, to another type, called the target type. Evaluation of a user-defined conversion centers on finding the most specific user-defined conversion operator for the particular source and target types. This determination is broken into several steps:

- Finding the set of classes and structs from which user-defined conversion operators will be considered. This set consists of the source type and its base classes and the target type and its base classes (with the implicit assumptions that only classes and structs can declare user-defined operators, and that non-class types have no base classes).
- From that set of types, determining which user-defined conversion operators are applicable. For a conversion operator to be applicable, it must be possible to perform a standard conversion (Section 6.3) from the source type to the operand type of the operator, and it must be possible to perform a standard conversion from the result type of the operator to the target type.
- From the set of applicable user-defined operators, determining which operator is unambiguously the most specific. In general terms, the most specific operator is the operator whose operand type is "closest" to the source type and whose result type is "closest" to the target type. The exact rules for establishing the most specific user-defined conversion operator are defined in the following sections.

Once a most specific user-defined conversion operator has been identified, the actual execution of the user-defined conversion involves up to three steps:

- First, if required, performing a standard conversion from the source type to the operand type of the user-defined conversion operator.
- Next, invoking the user-defined conversion operator to perform the conversion.
- Finally, if required, performing a standard conversion from the result type of the user-defined conversion operator to the target type.

Evaluation of a user-defined conversion never involves more than one user-defined conversion operator. In other words, a conversion from type `S` to type `T` will never first execute a user-defined conversion from `S` to `X` and then execute a user-defined conversion from `X` to `T`.

Exact definitions of evaluation of user-defined implicit or explicit conversions are given in the following sections. The definitions make use of the following terms:

- If a standard implicit conversion (Section 6.3.1) exists from a type `A` to a type `B`, and if neither `A` nor `B` are *interface-types*, then `A` is said to be encompassed by `B`, and `B` is said to encompass `A`.
- The most encompassing type in a set of types is the one type that encompasses all other types in the set. If no single type encompasses all other types, then the set has no most encompassing type. In more intuitive terms, the most encompassing type is the "largest" type in the set — the one type to which each of the other types can be implicitly converted.
- The most encompassed type in a set of types is the one type that is encompassed by all other types in the set. If no single type is encompassed by all other types, then the set has no most encompassed type. In more intuitive terms, the most encompassed type is the "smallest" type in the set — the one type that can be implicitly converted to each of the other types.

6.4.3 User-defined implicit conversions

A user-defined implicit conversion from type `S` to type `T` is processed as follows:

- Find the set of types, `D`, from which user-defined conversion operators will be considered. This set consists of `S` (if `S` is a class or struct), the base classes of `S` (if `S` is a class), and `T` (if `T` is a class or struct).
- Find the set of applicable user-defined conversion operators, `U`. This set consists of the user-defined implicit conversion operators declared by the classes or structs in `D` that convert from a type encompassing `S` to a type encompassed by `T`. If `U` is empty, the conversion is undefined and a compile-time error occurs.
- Find the most specific source type, `SX`, of the operators in `U`:
  - If any of the operators in `U` convert from `S`, then `SX` is `S`.
  - Otherwise, `SX` is the most encompassed type in the combined set of target types of the operators in `U`. If no most encompassed type can be found, then the conversion is ambiguous and a compile-time error occurs.
- Find the most specific target type, `TX`, of the operators in `U`:
  - If any of the operators in `U` convert to `T`, then `TX` is `T`.
  - Otherwise, `TX` is the most encompassing type in the combined set of target types of the operators in `U`. If no most encompassing type can be found, then the conversion is ambiguous and a compile-time error occurs.
- If `U` contains exactly one user-defined conversion operator that converts from `SX` to `TX`, then this is the most specific conversion operator. If no such operator exists, or if more than one such operator exists, then the conversion is ambiguous and a compile-time error occurs. Otherwise, the user-defined conversion is applied:
  - If `S` is not `SX`, then a standard implicit conversion from `S` to `SX` is performed.
  - The most specific user-defined conversion operator is invoked to convert from `SX` to `TX`.
  - If `TX` is not `T`, then a standard implicit conversion from `TX` to `T` is performed.

6.4.4 User-defined explicit conversions

A user-defined explicit conversion from type `S` to type `T` is processed as follows:

- Find the set of types, `D`, from which user-defined conversion operators will be considered. This set consists of `S` (if `S` is a class or struct), the base classes of `S` (if `S` is a class), `T` (if `T` is a class or struct), and the base classes of `T` (if `T` is a class).
- Find the set of applicable user-defined conversion operators, `U`. This set consists of the user-defined implicit or explicit conversion operators declared by the classes or structs in `D` that convert from a type encompassing or encompassed by `S` to a type encompassing or encompassed by `T`. If `U` is empty, the conversion is undefined and a compile-time error occurs.
- Find the most specific source type, `SX`, of the operators in `U`:
  - If any of the operators in `U` convert from `S`, then `SX` is `S`.
  - Otherwise, if any of the operators in `U` convert from types that encompass `S`, then `SX` is the most encompassed type in the combined set of source types of those operators. If no most encompassed type can be found, then the conversion is ambiguous and a compile-time error occurs.
  - Otherwise, `SX` is the most encompassing type in the combined set of source types of the operators in `U`. If no most encompassing type can be found, then the conversion is ambiguous and a compile-time error occurs.
- Find the most specific target type, `TX`, of the operators in `U`:
  - If any of the operators in `U` convert to `T`, then `TX` is `T`.
  - Otherwise, if any of the operators in `U` convert to types that are encompassed by `T`, then `TX` is the most encompassing type in the combined set of source types of those operators. If no most encompassing type can be found, then the conversion is ambiguous and a compile-time error occurs.
  - Otherwise, `TX` is the most encompassed type in the combined set of target types of the operators in `U`. If no most encompassed type can be found, then the conversion is ambiguous and a compile-time error occurs.
- If `U` contains exactly one user-defined conversion operator that converts from `SX` to `TX`, then this is the most specific conversion operator. If no such operator exists, or if more than one such operator exists, then the conversion is ambiguous and a compile-time error occurs. Otherwise, the user-defined conversion is applied:
  - If `S` is not `SX`, then a standard explicit conversion from `S` to `SX` is performed.
  - The most specific user-defined conversion operator is invoked to convert from `SX` to `TX`.
  - If `TX` is not `T`, then a standard explicit conversion from `TX` to `T` is performed.

7. Expressions

An expression is a sequence of operators and operands. This chapter defines the syntax, order of evaluation of operands and operators, and meaning of expressions.

7.1 Expression classifications

An expression is classified as one of the following:

- A value. Every value has an associated type.
- A variable. Every variable has an associated type, namely the declared type of the variable.
- A namespace. An expression with this classification can only appear as the left hand side of a member-access (Section 7.5.4). In any other context, an expression classified as a namespace causes a compile-time error.
- A type. An expression with this classification can only appear as the left hand side of a member-access (Section 7.5.4), or as an operand for the `as` operator (Section 7.9.10), the `is` operator (Section 7.9.9), or the `typeof` operator (Section 7.5.11). In any other context, an expression classified as a type causes a compile-time error.
- A method group, which is a set of overloaded methods resulting from a member lookup (Section 7.3). A method group may have an associated instance expression. When an instance method is invoked, the result of evaluating the instance expression becomes the instance represented by `this` (Section 7.5.7). A method group is only permitted in an invocation-expression (Section 7.5.5) or a delegate-creation-expression (Section 7.5.10.3). In any other context, an expression classified as a method group causes a compile-time error.
- A property access. Every property access has an associated type, namely the type of the property. Furthermore, a property access may have an associated instance expression. When an accessor (the `get` or `set` block) of an instance property access is invoked, the result of evaluating the instance expression becomes the instance represented by `this` (Section 7.5.7).
- An event access. Every event access has an associated type, namely the type of the event. Furthermore, an event access may have an associated instance expression. An event access may appear as the left hand operand of the `+=` and `-=` operators (Section 7.13.3). In any other context, an expression classified as an event access causes a compile-time error.
- An indexer access. Every indexer access has an associated type, namely the element type of the indexer. Furthermore, an indexer access has an associated instance expression and an associated argument list. When an accessor (the `get` or `set` block) of an indexer access is invoked, the result of evaluating the instance expression becomes the instance represented by `this` (Section 7.5.7), and the result of evaluating the argument list becomes the parameter list of the invocation.
- Nothing. This occurs when the expression is an invocation of a method with a return type of `void`. An expression classified as nothing is only valid in the context of a statement-expression (Section 8.6).

The final result of an expression is never a namespace, type, method group, or event access. Rather, as noted above, these categories of expressions are intermediate constructs that are only permitted in certain contexts.

A property access or indexer access is always reclassified as a value by performing an invocation of the get-accessor or the set-accessor. The particular accessor is determined by the context of the property or indexer access: If the access is the target of an assignment, the set-accessor is invoked to assign a new value (Section 7.13.1). Otherwise, the get-accessor is invoked to obtain the current value (Section 7.1.1).

7.1.1 Values of expressions

Most of the constructs that involve an expression ultimately require the expression to denote a value. In such cases, if the actual expression denotes a namespace, a type, a method group, or nothing, a compile-time error occurs. However, if the expression denotes a property access, an indexer access, or a variable, the value of the property, indexer, or variable is implicitly substituted:

- The value of a variable is simply the value currently stored in the storage location identified by the variable. A variable must be considered definitely assigned (Section 5.3) before its value can be obtained, or otherwise a compile-time error occurs.
- The value of a property access expression is obtained by invoking the get-accessor of the property. If the property has no get-accessor, a compile-time error occurs. Otherwise, a function member invocation (Section 7.4.3) is performed, and the result of the invocation becomes the value of the property access expression.
- The value of an indexer access expression is obtained by invoking the get-accessor of the indexer. If the indexer has no get-accessor, a compile-time error occurs. Otherwise, a function member invocation (Section 7.4.3) is performed with the argument list associated with the indexer access expression, and the result of the invocation becomes the value of the indexer access expression.

7.2 Operators

Expressions are constructed from operands and operators. The operators of an expression indicate which operations to apply to the operands. Examples of operators include `+`, `-`, `*`, `/`, and `new`. Examples of operands include literals, fields, local variables, and expressions.

There are three kinds of operators:

- Unary operators. The unary operators take one operand and use either prefix notation (such as `-x`) or postfix notation (such as `x++`).
- Binary operators. The binary operators take two operands and all use infix notation (such as `x + y`).
- Ternary operator. Only one ternary operator, `?:`, exists; it takes three operands and uses infix notation (`c? x: y`).

The order of evaluation of operators in an expression is determined by the precedence and associativity of the operators (Section 7.2.1).

Operands in an expression are evaluated from left to right. For example, in `F(i) + G(i++) * H(i)`, method `F` is called using the old value of `i`, then method `G` is called with the old value of `i`, and, finally, method `H` is called with the new value of `i`. This is separate from and unrelated to operator precedence.

Certain operators can be overloaded. Operator overloading permits user-defined operator implementations to be specified for operations where one or both of the operands are of a user-defined class or struct type (Section 7.2.2).

7.2.1 Operator precedence and associativity

When an expression contains multiple operators, the precedence of the operators controls the order in which the individual operators are evaluated. For example, the expression `x + y * z` is evaluated as `x + (y * z)` because the `*` operator has higher precedence than the binary `+` operator. The precedence of an operator is established by the definition of its associated grammar production. For example, an additive-expression consists of a sequence of multiplicative-expressions separated by `+` or `-` operators, thus giving the `+` and `-` operators lower precedence than the `*`, `/`, and `%` operators.

The following table summarizes all operators in order of precedence from highest to lowest:

Section	Category	Operators
Section 7.5	Primary	`x.y  f(x)  a[x]  x++  x--  new  typeof  checked  unchecked`
Section 7.6	Unary	`+  -  !  ~  ++x  --x  (T)x`
Section 7.7	Multiplicative	`*  /  %`
Section 7.7	Additive	`+  -`
Section 7.8	Shift	`<<  >>`
Section 7.9	Relational and type testing	`<  >  <=  >=  is  as`
Section 7.9	Equality	`==  !=`
Section 7.10	Logical AND	`&`
Section 7.10	Logical XOR	`^`
Section 7.10	Logical OR	`|`
Section 7.11	Conditional AND	`&&`
Section 7.11	Conditional OR	`||`
Section 7.12	Conditional	`?:`
Section 7.13	Assignment	`=  *=  /=  %=  +=  -=  <<=  >>=  &=  ^=  |=`

When an operand occurs between two operators with the same precedence, the associativity of the operators controls the order in which the operations are performed:

- Except for the assignment operators, all binary operators are left-associative, meaning that operations are performed from

left to right. For example, `x + y + z` is evaluated as `(x + y) + z`.

- The assignment operators and the conditional operator (`?:`) are right-associative, meaning that operations are performed from right to left. For example, `x = y = z` is evaluated as `x = (y = z)`.

Precedence and associativity can be controlled using parentheses. For example, `x + y * z` first multiplies `y` by `z` and then adds the result to `x`, but `(x + y) * z` first adds `x` and `y` and then multiplies the result by `z`.

7.2.2 Operator overloading

All unary and binary operators have predefined implementations that are automatically available in any expression. In addition to the predefined implementations, user-defined implementations can be introduced by including `operator` declarations in classes and structs (Section 10.9). User-defined operator implementations always take precedence over predefined operator implementations: Only when no applicable user-defined operator implementations exist will the predefined operator implementations be considered.

The overloadable unary operators are:

```
+   -   !   ~   ++   --   true   false
```

Although `true` and `false` are not used explicitly in expressions, they are considered operators because they are invoked in several expression contexts: Boolean expressions (Section 7.16) and expressions involving the conditional (Section 7.12), and conditional logical operators (Section 7.11).

The overloadable binary operators are:

```
+   -   *   /   %   &   |   ^   <<   >>   ==   !=   >   <   >=   <=
```

Only the operators listed above can be overloaded. In particular, it is not possible to overload member access, method invocation, or the `=`, `&&`, `||`, `?:`, `checked`, `unchecked`, `new`, `typeof`, `as`, and `is` operators.

When a binary operator is overloaded, the corresponding assignment operator (if any) is also implicitly overloaded. For example, an overload of operator `*` is also an overload of operator `*=`. This is described further in Section 7.13. Note that the assignment operator itself (`=`) cannot be overloaded. An assignment always performs a simple bit-wise copy of a value into a variable.

Cast operations, such as `(T)x`, are overloaded by providing user-defined conversions (Section 6.4).

Element access, such as `a[x]`, is not considered an overloadable operator. Instead, user-defined indexing is supported through indexers (Section 10.8).

In expressions, operators are referenced using operator notation, and in declarations, operators are referenced using functional notation. The following table shows the relationship between operator and functional notations for unary and binary operators. In the first entry, *op* denotes any overloadable unary prefix operator. In the second entry, *op* denotes the unary postfix `++` and `--` operators. In the third entry, *op* denotes any overloadable binary operator.

Operator notation	Functional notation
op `x`	`operator` *op* `(x)`
`x` *op*	`operator` *op* `(x)`
`x` *op* `y`	`operator` *op* `(x, y)`

User-defined operator declarations always require at least one of the parameters to be of the class or struct type that contains the operator declaration. Thus, it is not possible for a user-defined operator to have the same signature as a predefined operator.

User-defined operator declarations cannot modify the syntax, precedence, or associativity of an operator. For example, the `/` operator is always a binary operator, always has the precedence level specified in Section 7.2.1, and is always left-associative.

While it is possible for a user-defined operator to perform any computation it pleases, implementations that produce results other than those that are intuitively expected are strongly discouraged. For example, an implementation of `operator ==` should compare the two operands for equality and return an appropriate `bool` result.

The descriptions of individual operators in Section 7.5 through Section 7.13 specify the predefined implementations of the operators and any additional rules that apply to each operator. The descriptions make use of the terms unary operator overload resolution, binary operator overload resolution, and numeric promotion, definitions of which are found in the following sections.

7.2.3 Unary operator overload resolution

An operation of the form *op* `x` or `x` *op*, where *op* is an overloadable unary operator, and `x` is an expression of type `X`, is processed as follows:

- The set of candidate user-defined operators provided by `X` for the operation `operator` *op* `(x)` is determined using the rules of Section 7.2.5.
- If the set of candidate user-defined operators is not empty, then this becomes the set of candidate operators for the operation. Otherwise, the predefined unary `operator` *op* implementations become the set of candidate operators for the operation. The predefined implementations of a given operator are specified in the description of the operator (Section 7.5 and Section 7.6).
- The overload resolution rules of Section 7.4.2 are applied to the set of candidate operators to select the best operator with respect to the argument list `(x)`, and this operator becomes the result of the overload resolution process. If overload resolution fails to select a single best operator, a compile-time error occurs.

7.2.4 Binary operator overload resolution

An operation of the form `x` *op* `y`, where *op* is an overloadable binary operator, `x` is an expression of type `X`, and `y` is an expression of type `Y`, is processed as follows:

- The set of candidate user-defined operators provided by `X` and `Y` for the operation `operator` *op* `(x, y)` is determined. The set consists of the union of the candidate operators provided by `X` and the candidate operators provided by `Y`, each determined using the rules of Section 7.2.5. If `X` and `Y` are the same type, or if `X` and `Y` are derived from a common base type, then shared candidate operators only occur in the combined set once.
- If the set of candidate user-defined operators is not empty, then this becomes the set of candidate operators for the operation. Otherwise, the predefined binary `operator` *op* implementations become the set of candidate operators for the operation. The predefined implementations of a given operator are specified in the description of the operator (Section 7.7 through Section 7.13).
- The overload resolution rules of Section 7.4.2 are applied to the set of candidate operators to select the best operator with respect to the argument list `(x, y)`, and this operator becomes the result of the overload resolution process. If overload resolution fails to select a single best operator, a compile-time error occurs.

7.2.5 Candidate user-defined operators

Given a type `T` and an operation `operator` *op* `(A)`, where *op* is an overloadable operator and `A` is an argument list, the set of candidate user-defined operators provided by `T` for `operator` *op* `(A)` is determined as follows:

- For all `operator` *op* declarations in `T`, if at least one operator is applicable (Section 7.4.2.1) with respect to the argument list `A`, then the set of candidate operators consists of all applicable `operator` *op* declarations in `T`.
- Otherwise, if `T` is `object`, the set of candidate operators is empty.
- Otherwise, the set of candidate operators provided by `T` is the set of candidate operators provided by the direct base class of `T`.

7.2.6 Numeric promotions

Numeric promotion consists of automatically performing certain implicit conversions of the operands of the predefined unary and binary numeric operators. Numeric promotion is not a distinct mechanism, but rather an effect of applying overload resolution to the predefined operators. Numeric promotion specifically does not affect evaluation of user-defined operators, although user-defined operators can be implemented to exhibit similar effects.

As an example of numeric promotion, consider the predefined implementations of the binary `*` operator:

```
int operator *(int x, int y);
uint operator *(uint x, uint y);
long operator *(long x, long y);
ulong operator *(ulong x, ulong y);
float operator *(float x, float y);
double operator *(double x, double y);
decimal operator *(decimal x, decimal y);
```

When overload resolution rules (Section 7.4.2) are applied to this set of operators, the effect is to select the first of the operators for which implicit conversions exist from the operand types. For example, for the operation `b * s`, where `b` is a `byte` and `s` is a `short`, overload resolution selects `operator *(int, int)` as the best operator. Thus, the effect is that `b` and `s` are converted to `int`, and the type of the result is `int`. Likewise, for the operation `i * d`, where `i` is an `int` and `d` is a `double`, overload resolution selects `operator *(double, double)` as the best operator.

7.2.6.1 Unary numeric promotions

Unary numeric promotion occurs for the operands of the predefined `+`, `–`, and `~` unary operators. Unary numeric promotion simply consists of converting operands of type `sbyte`, `byte`, `short`, `ushort`, or `char` to type `int`. Additionally, for the unary `–` operator, unary numeric promotion converts operands of type `uint` to type `long`.

7.2.6.2 Binary numeric promotions

Binary numeric promotion occurs for the operands of the predefined `+`, `-`, `*`, `/`, `%`, `&`, `|`, `^`, `==`, `!=`, `>`, `<`, `>=`, and `<=` binary operators. Binary numeric promotion implicitly converts both operands to a common type which, in case of the non-relational operators, also becomes the result type of the operation. Binary numeric promotion consists of applying the following rules, in the order they appear here:

- If either operand is of type `decimal`, the other operand is converted to type `decimal`, or a compile-time error occurs if the other operand is of type `float` or `double`.
- Otherwise, if either operand is of type `double`, the other operand is converted to type `double`.
- Otherwise, if either operand is of type `float`, the other operand is converted to type `float`.
- Otherwise, if either operand is of type `ulong`, the other operand is converted to type `ulong`, or a compile-time error occurs if the other operand is of type `sbyte`, `short`, `int`, or `long`.
- Otherwise, if either operand is of type `long`, the other operand is converted to type `long`.
- Otherwise, if either operand is of type `uint` and the other operand is of type `sbyte`, `short`, or `int`, both operands are converted to type `long`.
- Otherwise, if either operand is of type `uint`, the other operand is converted to type `uint`.
- Otherwise, both operands are converted to type `int`.

Note that the first rule disallows any operations that mix the `decimal` type with the `double` and `float` types. The rule follows from the fact that there are no implicit conversions between the `decimal` type and the `double` and `float` types.

Also note that it is not possible for an operand to be of type `ulong` when the other operand is of a signed integral type. The reason is that no integral type exists that can represent the full range of `ulong` as well as the signed integral types.

In both of the above cases, a cast expression can be used to explicitly convert one operand to a type that is compatible with the other operand.

In the example

```
decimal AddPercent(decimal x, double percent) {
   return x * (1.0 + percent / 100.0);
}
```

a compile-time error occurs because a `decimal` cannot be multiplied by a `double`. The error is resolved by explicitly converting the second operand to `decimal`, as follows:

```
decimal AddPercent(decimal x, double percent) {
   return x * (decimal)(1.0 + percent / 100.0);
}
```

7.3 Member lookup

A member lookup is the process whereby the meaning of a name in the context of a type is determined. A member lookup may occur as part of evaluating a simple-name (Section 7.5.2) or a member-access (Section 7.5.4) in an expression.

A member lookup of a name `N` in a type `T` is processed as follows:

- First, the set of all accessible (Section 3.5) members named `N` declared in `T` and the base types (Section 7.3.1) of `T` is constructed. Declarations that include an `override` modifier are excluded from the set. If no members named `N` exist and are accessible, then the lookup produces no match, and the following steps are not evaluated.
- Next, members that are hidden by other members are removed from the set. For every member `S.M` in the set, where `S` is the type in which the member `M` is declared, the following rules are applied:
    - If `M` is a constant, field, property, event, type, or enumeration member, then all members declared in a base type of `S` are removed from the set.
    - If `M` is a method, then all non-method members declared in a base type of `S` are removed from the set, and all methods with the same signature as `M` declared in a base type of `S` are removed from the set.
- Finally, having removed hidden members, the result of the lookup is determined:
    - If the set consists of a single non-method member, then this member is the result of the lookup.
    - Otherwise, if the set contains only methods, then this group of methods is the result of the lookup.
    - Otherwise, the lookup is ambiguous, and a compile-time error occurs (this situation can only occur for a member lookup in an interface that has multiple direct base interfaces).

For member lookups in types other than interfaces, and member lookups in interfaces that are strictly single-inheritance (each interface in the inheritance chain has exactly zero or one direct base interface), the effect of the lookup rules is simply that derived members hide base members with the same name or signature. Such single-inheritance lookups are never ambiguous. The ambiguities that can possibly arise from member lookups in multiple-inheritance interfaces are described in Section 13.2.5.

7.3.1 Base types

For purposes of member lookup, a type `T` is considered to have the following base types:

- If `T` is `object`, then `T` has no base type.
- If `T` is a *value-type*, the base type of `T` is the class type `object`.
- If `T` is a *class-type*, the base types of `T` are the base classes of `T`, including the class type `object`.
- If `T` is an *interface-type*, the base types of `T` are the base interfaces of `T` and the class type `object`.
- If `T` is an *array-type*, the base types of `T` are the class types `System.Array` and `object`.
- If `T` is a *delegate-type*, the base types of `T` are the class types `System.Delegate` and `object`.

7.4 Function members

Function members are members that contain executable statements. Function members are always members of types and cannot be members of namespaces. C# defines the following categories of function members:

- Methods
- Properties
- Events
- Indexers
- User-defined operators
- Instance constructors
- Static constructors
- Destructors

Except for destructors and static constructors (which cannot be invoked explicitly), the statements contained in function members are executed through function member invocations. The actual syntax for writing a function member invocation depends on the particular function member category.

The argument list (Section 7.4.1) of a function member invocation provides actual values or variable references for the parameters of the function member.

Invocations of methods, indexers, operators and instance constructors employ overload resolution to determine which of a candidate set of function members to invoke. This process is described in Section 7.4.2.

Once a particular function member has been identified at compile-time, possibly through overload resolution, the actual run-time process of invoking the function member is described in Section 7.4.3.

The following table summarizes the processing that takes place in constructs involving the six categories of function members that can be explicitly invoked. In the table, `e`, `x`, `y`, and `value` indicate expressions classified as variables or values, `T` indicates an expression classified as a type, `F` is the simple name of a method, and `P` is the simple name of a property.

Construct	Example	Description
Method invocation	`F(x, y)`	Overload resolution is applied to select the best method `F` in the containing class or struct. The method is invoked with the argument list `(x, y)`. If the method is not `static`, the instance expression is `this`.
Method invocation	`T.F(x, y)`	Overload resolution is applied to select the best method `F` in the class or struct `T`. A compile-time error occurs if the method is not `static`. The method is invoked with the argument list `(x, y)`.
Method invocation	`e.F(x, y)`	Overload resolution is applied to select the best method F in the class, struct, or interface given by the type of `e`. A compile-time error occurs if the method is `static`. The method is invoked with the instance expression `e` and the argument list `(x, y)`.
Property access	`P`	The `get` accessor of the property `P` in the containing class or struct is invoked. A compile-time error occurs if `P` is write-only. If `P` is not `static`, the instance expression is `this`.
Property access	`P = value`	The `set` accessor of the property `P` in the containing class or struct is invoked with the argument list `(value)`. A compile-time error occurs if `P` is read-only. If `P` is not `static`, the instance expression is `this`.
Property access	`T.P`	The `get` accessor of the property `P` in the class or struct `T` is invoked. A compile-time error occurs if `P` is not `static` or if `P` is write-only.
Property access	`T.P = value`	The `set` accessor of the property `P` in the class or struct `T` is invoked with the argument list `(value)`. A compile-time error occurs if `P` is not `static` or if `P` is read-only.

Property access	`e.P`	The `get` accessor of the property `P` in the class, struct, or interface given by the type of `e` is invoked with the instance expression `e`. A compile-time error occurs if `P` is `static` or if `P` is write-only.
Property access	`e.P = value`	The `set` accessor of the property `P` in the class, struct, or interface given by the type of `e` is invoked with the instance expression `e` and the argument list `(value)`. A compile-time error occurs if `P` is `static` or if `P` is read-only.
Event access	`E += value`	The `add` accessor of the event `E` in the containing class or struct is invoked. If `E` is not static, the instance expression is `this`.
Event access	`E -= value`	The `remove` accessor of the event `E` in the containing class or struct is invoked. If `E` is not static, the instance expression is `this`.
Event access	`T.E += value`	The `add` accessor of the event `E` in the class or struct `T` is invoked. A compile-time error occurs if `E` is not static.
Event access	`T.E -= value`	The remove accessor of the event `E` in the class or struct `T` is invoked. A compile-time error occurs if `E` is not static.
Event access	`e.E += value`	The `add` accessor of the event `E` in the class, struct, or interface given by the type of `e` is invoked with the instance expression `e`. A compile-time error occurs if `E` is static.
Event access	`e.E -= value`	The `remove` accessor of the event `E` in the class, struct, or interface given by the type of `e` is invoked with the instance expression `e`. A compile-time error occurs if `E` is static.
Indexer access	`e[x, y]`	Overload resolution is applied to select the best indexer in the class, struct, or interface given by the type of e. The `get` accessor of the indexer is invoked with the instance expression `e` and the argument list `(x, y)`. A compile-time error occurs if the indexer is write-only.
Indexer access	`e[x, y] = value`	Overload resolution is applied to select the best indexer in the class, struct, or interface given by the type of `e`. The `set` accessor of the indexer is invoked with the instance expression `e` and the argument list `(x, y, value)`. A compile-time error occurs if the indexer is read-only.
Operator invocation	`-x`	Overload resolution is applied to select the best unary operator in the class or struct given by the type of `x`. The selected operator is invoked with the argument list `(x)`.
Operator invocation	`x + y`	Overload resolution is applied to select the best binary operator in the classes or structs given by the types of x and y. The selected operator is invoked with the argument list `(x, y)`.
Instance constructor	`new T(x, y)`	Overload resolution is applied to select the best instance constructor in the class or struct `T`. The instance constructor is invoked with the argument list `(x, y)`.

7.4.1 Argument lists

Every function member invocation includes an argument list which provides actual values or variable references for the parameters of the function member. The syntax for specifying the argument list of a function member invocation depends on the function member category:

- For instance constructors, methods, and delegates, the arguments are specified as an argument-list, as described below.
- For properties, the argument list is empty when invoking the `get` accessor, and consists of the expression specified as the right operand of the assignment operator when invoking the `set` accessor.
- For events, the argument list consists of the expression specified as the right operand of the `+=` or `-=` operator.
- For indexers, the argument list consists of the expressions specified between the square brackets in the indexer access. When invoking the `set` accessor, the argument list additionally includes the expression specified as the right operand of the assignment operator.
- For user-defined operators, the argument list consists of the single operand of the unary operator or the two operands of the binary operator.

The arguments of properties (Section 10.6), events (Section 10.7), indexers (Section 10.8), and user-defined operators (Section 10.9) are always passed as value parameters (Section 10.5.1.1). Reference and output parameters are not supported for these categories of function members.

The arguments of an instance constructor, method, or delegate invocation are specified as an argument-list:

argument-list:
  argument
  argument-list , *argument*
argument:
  expression
  ref *variable-reference*
  out *variable-reference*

An argument-list consists of one or more arguments, separated by commas. Each argument can take one of the following forms:

- An expression, indicating that the argument is passed as a value parameter (Section 10.5.1.1).
- The keyword `ref` followed by a variable-reference (Section 5.3.3), indicating that the argument is passed as a reference parameter (Section 10.5.1.2). A variable must be definitely assigned (Section 5.3) before it can be passed as a reference parameter. A volatile field (Section 10.4.3) cannot be passed as a reference parameter.
- The keyword `out` followed by a variable-reference (Section 5.3.3), indicating that the argument is passed as an output parameter (Section 10.5.1.3). A variable is considered definitely assigned (Section 5.3) following a function member invocation in which the variable is passed as an output parameter. A volatile field (Section 10.4.3) cannot be passed as an output parameter.

During the run-time processing of a function member invocation (Section 7.4.3), the expressions or variable references of an argument list are evaluated in order, from left to right, as follows:

- For a value parameter, the argument expression is evaluated and an implicit conversion (Section 6.1) to the corresponding parameter type is performed. The resulting value becomes the initial value of the value parameter in the function member invocation.
- For a reference or output parameter, the variable reference is evaluated and the resulting storage location becomes the storage location represented by the parameter in the function member invocation. If the variable reference given as a reference or output parameter is an array element of a *reference-type*, a run-time check is performed to ensure that the element type of the array is identical to the type of the parameter. If this check fails, a `System.ArrayTypeMismatchException` is thrown.

Methods, indexers, and instance constructors may declare their right-most parameter to be a parameter array (Section 10.5.1.4). Such function members are invoked either in their normal form or in their expanded form depending on which is applicable (Section 7.4.2.1):

- When a function member with a parameter array is invoked in its normal form, the argument given for the parameter array must be a single expression of a type that is implicitly convertible (Section 6.1) to the parameter array type. In this case, the parameter array acts precisely like a value parameter.
- When a function member with a parameter array is invoked in its expanded form, the invocation must specify zero or more

arguments for the parameter array, where each argument is an expression of a type that is implicitly convertible (Section 6.1) to the element type of the parameter array. In this case, the invocation creates an instance of the parameter array type with a length corresponding to the number of arguments, initializes the elements of the array instance with the given argument values, and uses the newly created array instance as the actual argument.

The expressions of an argument list are always evaluated in the order they are written. Thus, the example

```
class Test
{
   static void F(int x, int y, int z) {
      System.Console.WriteLine("x = {0}, y = {1}, z = {2}", x, y, z);
   }
   static void Main() {
      int i = 0;
      F(i++, i++, i++);
   }
}
```

produces the output

```
x = 0, y = 1, z = 2
```

The array co-variance rules (Section 12.5) permit a value of an array type `A[]` to be a reference to an instance of an array type `B[]`, provided an implicit reference conversion exists from `B` to `A`. Because of these rules, when an array element of a *reference-type* is passed as a reference or output parameter, a run-time check is required to ensure that the actual element type of the array is identical to that of the parameter. In the example

```
class Test
{
   static void F(ref object x) {...}
   static void Main() {
      object[] a = new object[10];
      object[] b = new string[10];
      F(ref a[0]);      // Ok
      F(ref b[1]);      // ArrayTypeMismatchException
   }
}
```

the second invocation of `F` causes a `System.ArrayTypeMismatchException` to be thrown because the actual element type of `b` is `string` and not `object`.

When a function member with a parameter array is invoked in its expanded form, the invocation is processed exactly as if an array creation expression with an array initializer (Section 7.5.10.2) was inserted around the expanded parameters. For example, given the declaration

```
void F(int x, int y, params object[] args);
```

the following invocations of the expanded form of the method

```
F(10, 20);
F(10, 20, 30, 40);
F(10, 20, 1, "hello", 3.0);
```

correspond exactly to

```
F(10, 20, new object[] {});
F(10, 20, new object[] {30, 40});
F(10, 20, new object[] {1, "hello", 3.0});
```

In particular, note that an empty array is created when there are zero arguments given for the parameter array.

7.4.2 Overload resolution

Overload resolution is a compile-time mechanism for selecting the best function member to invoke given an argument list and a set of candidate function members. Overload resolution selects the function member to invoke in the following distinct contexts within C#:

- Invocation of a method named in an invocation-expression (Section 7.5.5).
- Invocation of an instance constructor named in an object-creation-expression (Section 7.5.10.1).
- Invocation of an indexer accessor through an element-access (Section 7.5.6).
- Invocation of a predefined or user-defined operator referenced in an expression (Section 7.2.3 and Section 7.2.4).

Each of these contexts defines the set of candidate function members and the list of arguments in its own unique way, as described in detail in the sections listed above. For example, the set of candidates for a method invocation does not include methods marked `override` (Section 7.3), and methods in a base class are not candidates if any method in a derived class is applicable (Section 7.5.5.1).

Once the candidate function members and the argument list have been identified, the selection of the best function member is the same in all cases:

- Given the set of applicable candidate function members, the best function member in that set is located.
- If the set contains only one function member, then that function member is the best function member.
- Otherwise, the best function member is the one function member that is better than all other function members with respect to the given argument list, provided that each function member is compared to all other function members using the rules in Section 7.4.2.2.
- If there is not exactly one function member that is better than all other function members, then the function member invocation is ambiguous and a compile-time error occurs.

The following sections define the exact meanings of the terms applicable function member and better function member.

7.4.2.1 Applicable function member

A function member is said to be an applicable function member with respect to an argument list `A` when all of the following are true:

- The number of arguments in `A` is identical to the number of parameters in the function member declaration.
- For each argument in `A`, the parameter passing mode of the argument (that is, value, `ref`, or `out`) is identical to the parameter passing mode of the corresponding parameter, and
  - for a value parameter or a parameter array, an implicit conversion (Section 6.1) exists from the type of the argument to the type of the corresponding parameter, or
  - for a `ref` or `out` parameter, the type of the argument is identical to the type of the corresponding parameter. After all, a `ref` or `out` parameter is an alias for the argument passed.

For a function member that includes a parameter array, if the function member is applicable by the above rules, it is said to be applicable in its normal form. If a function member that includes a parameter array is not applicable in its normal form, the function member may instead be applicable in its expanded form:

- The expanded form is constructed by replacing the parameter array in the function member declaration with zero or more value parameters of the element type of the parameter array such that the number of arguments in the argument list `A` matches the total number of parameters. If `A` has fewer arguments than the number of fixed parameters in the function member declaration, the expanded form of the function member cannot be constructed and is thus not applicable.
- If the class, struct, or interface in which the function member is declared already contains another applicable function member with the same signature as the expanded form, the expanded form is not applicable.
- Otherwise, the expanded form is applicable if for each argument in `A` the parameter passing mode of the argument is identical to the parameter passing mode of the corresponding parameter, and
  - for a fixed value parameter or a value parameter created by the expansion, an implicit conversion (Section 6.1) exists from the type of the argument to the type of the corresponding parameter, or
  - for a `ref` or `out` parameter, the type of the argument is identical to the type of the corresponding parameter.

7.4.2.2 Better function member

Given an argument list `A` with a set of argument types `{A1, A2, ..., AN}` and two applicable function members `MP` and `MQ` with parameter types `{P1, P2, ..., PN}` and `{Q1, Q2, ..., QN}`, `MP` is defined to be a better function member than `MQ` if

- for each argument, the implicit conversion from `AX` to `PX` is not worse than the implicit conversion from `AX` to `QX`, and
- for at least one argument, the conversion from `AX` to `PX` is better than the conversion from `AX` to `QX`.

When performing this evaluation, if `MP` or `MQ` is applicable in its expanded form, then `PX` or `QX` refers to a parameter in the expanded form of the parameter list.

7.4.2.3 Better conversion

Given an implicit conversion `C1` that converts from a type `S` to a type `T1`, and an implicit conversion `C2` that converts from a type `S` to a type `T2`, the better conversion of the two conversions is determined as follows:

- If `T1` and `T2` are the same type, neither conversion is better.
- If `S` is `T1`, `C1` is the better conversion.
- If `S` is `T2`, `C2` is the better conversion.
- If an implicit conversion from `T1` to `T2` exists, and no implicit conversion from `T2` to `T1` exists, `C1` is the better conversion.
- If an implicit conversion from `T2` to `T1` exists, and no implicit conversion from `T1` to `T2` exists, `C2` is the better conversion.
- If `T1` is `sbyte` and `T2` is `byte`, `ushort`, `uint`, or `ulong`, `C1` is the better conversion.
- If `T2` is `sbyte` and `T1` is `byte`, `ushort`, `uint`, or `ulong`, `C2` is the better conversion.
- If `T1` is `short` and `T2` is `ushort`, `uint`, or `ulong`, `C1` is the better conversion.
- If `T2` is `short` and `T1` is `ushort`, `uint`, or `ulong`, `C2` is the better conversion.
- If `T1` is `int` and `T2` is `uint`, or `ulong`, `C1` is the better conversion.
- If `T2` is `int` and `T1` is `uint`, or `ulong`, `C2` is the better conversion.
- If `T1` is `long` and `T2` is `ulong`, `C1` is the better conversion.
- If `T2` is `long` and `T1` is `ulong`, `C2` is the better conversion.
- Otherwise, neither conversion is better.

If an implicit conversion `C1` is defined by these rules to be a better conversion than an implicit conversion `C2`, then it is also the case that `C2` is a worse conversion than `C1`.

7.4.3 Function member invocation

This section describes the process that takes place at run-time to invoke a particular function member. It is assumed that a compile-time process has already determined the particular member to invoke, possibly by applying overload resolution to a set of candidate function members.

For purposes of describing the invocation process, function members are divided into two categories:

- Static function members. These are instance constructors, static methods, static property accessors, and user-defined operators. Static function members are always non-virtual.
- Instance function members. These are instance methods, instance property accessors, and indexer accessors. Instance function members are either non-virtual or virtual, and are always invoked on a particular instance. The instance is computed by an instance expression, and it becomes accessible within the function member as `this` (Section 7.5.7).

The run-time processing of a function member invocation consists of the following steps, where `M` is the function member and, if `M` is an instance member, `E` is the instance expression:

- If `M` is a static function member:
  - The argument list is evaluated as described in Section 7.4.1.
  - `M` is invoked.
- If `M` is an instance function member declared in a *value-type*:
  - `E` is evaluated. If this evaluation causes an exception, then no further steps are executed.
  - If `E` is not classified as a variable, then a temporary local variable of `E`'s type is created and the value of `E` is assigned to that variable. `E` is then reclassified as a reference to that temporary local variable. The temporary variable is accessible as `this` within `M`, but not in any other way. Thus, only when `E` is a true variable is it possible for the caller to observe the changes that `M` makes to `this`.
  - The argument list is evaluated as described in Section 7.4.1.
  - `M` is invoked. The variable referenced by `E` becomes the variable referenced by `this`.
- If `M` is an instance function member declared in a *reference-type*:
  - `E` is evaluated. If this evaluation causes an exception, then no further steps are executed.
  - The argument list is evaluated as described in Section 7.4.1.
  - If the type of `E` is a *value-type*, a boxing conversion (Section 4.3.1) is performed to convert `E` to type `object`, and `E` is considered to be of type `object` in the following steps. In this case, `M` could only be a member of `System.Object`.
  - The value of `E` is checked to be valid. If the value of `E` is `null`, a `System.NullReferenceException` is thrown and no further steps are executed.
  - The function member implementation to invoke is determined:

    If the compile-time type of `E` is an interface, the function member to invoke is the implementation of `M` provided by the run-time type of the instance referenced by `E`. This function member is determined by applying the interface mapping rules (Section 13.4.2) to determine the implementation of `M` provided by the run-time type of the instance referenced by `E`.

    Otherwise, if `M` is a virtual function member, the function member to invoke is the implementation of `M` provided by the run-time type of the instance referenced by `E`. This function member is determined by applying the rules for determining the most derived implementation (Section 10.5.3) of `M` with respect to the run-time type of the instance referenced by `E`.

    Otherwise, `M` is a non-virtual function member, and the function member to invoke is `M` itself.

  - The function member implementation determined in the step above is invoked. The object referenced by `E` becomes the object referenced by `this`.

7.4.3.1 Invocations on boxed instances

A function member implemented in a *value-type* can be invoked through a boxed instance of that *value-type* in the following situations:

- When the function member is an `override` of a method inherited from type `object` and is invoked through an instance expression of type `object`.
- When the function member is an implementation of an interface function member and is invoked through an instance expression of an *interface-type*.
- When the function member is invoked through a delegate.

In these situations, the boxed instance is considered to contain a variable of the *value-type*, and this variable becomes the variable referenced by `this` within the function member invocation. In particular, this means that when a function member is invoked on a boxed instance, it is possible for the function member to modify the value contained in the boxed instance.

7.5 Primary expressions

Primary expressions include the simplest forms of expressions.

primary-expression:
  primary-no-array-creation-expression
  array-creation-expression
primary-no-array-creation-expression:
  literal
  simple-name
  parenthesized-expression
  member-access
  invocation-expression
  element-access
  this-access
  base-access
  post-increment-expression
  post-decrement-expression
  object-creation-expression
  delegate-creation-expression
  typeof-expression
  checked-expression
  unchecked-expression

Primary expressions are divided between array-creation-expressions and primary-no-array-creation-expressions. Treating array-creation-expression in this way, rather than listing it along with the other simple expression forms, enables the grammar to disallow potentially confusing code such as

```
object o = new int[3][1];
```

which would otherwise be interpreted as

```
object o = (new int[3])[1];
```

7.5.1 Literals

A primary-expression that consists of a literal (Section 2.4.4) is classified as a value.

7.5.2 Simple names

A simple-name consists of a single identifier.

simple-name:
  identifier

A simple-name is evaluated and classified as follows:

- If the simple-name appears within a block and if the block's (or an enclosing block's) local variable declaration space (Section 3.3) contains a local variable or parameter with the given name, then the simple-name refers to that local variable or parameter and is classified as a variable.
- Otherwise, for each type `T`, starting with the immediately enclosing class, struct, or enumeration declaration and continuing with each enclosing outer class or struct declaration (if any), if a member lookup of the simple-name in `T` produces a match:
- If `T` is the immediately enclosing class or struct type and the lookup identifies one or more methods, the result is a method group with an associated instance expression of `this`.
- If `T` is the immediately enclosing class or struct type, if the lookup identifies an instance member, and if the reference occurs within the block of an instance constructor, an instance method, or an instance accessor, the result is the same as a member access (Section 7.5.4) of the form `this.E`, where `E` is the simple-name.
- Otherwise, the result is the same as a member access (Section 7.5.4) of the form `T.E`, where `E` is the simple-name. In this case, it is a compile-time error for the simple-name to refer to an instance member.
- Otherwise, starting with the namespace in which the simple-name occurs, continuing with each enclosing namespace (if any), and ending with the global namespace, the following steps are evaluated until an entity is located:
- If the namespace contains a namespace member with the given name, then the simple-name refers to that member and, depending on the member, is classified as a namespace or a type.
- Otherwise, if the namespace has a corresponding namespace declaration enclosing the location where the simple-name occurs, then:
- If the namespace declaration contains a using-alias-directive that associates the given name with an imported namespace or type, then the simple-name refers to that namespace or type.
- Otherwise, if the namespaces imported by the using-namespace-directives of the namespace declaration contain exactly one type with the given name, then the simple-name refers to that type.
- Otherwise, if the namespaces imported by the using-namespace-directives of the namespace declaration contain more than one type with the given name, then the simple-name is ambiguous and a compile-time error occurs.
- Otherwise, the name given by the simple-name is undefined and a compile-time error occurs.

7.5.2.1 Invariant meaning in blocks

For each occurrence of a given identifier as a simple-name in an expression, every other occurrence of the same identifier as a simple-name in an expression within the immediately enclosing block (Section 8.2) or switch-block (Section 8.7.2) must refer to the same entity. This rule ensures that the meaning of a name in the context of an expression is always the same within a block.

The example

```
class Test
{
   double x;
   void F(bool b) {
      x = 1.0;
      if (b) {
         int x;
         x = 1;
      }
   }
}
```

results in a compile-time error because `x` refers to different entities within the outer block (the extent of which includes the nested block in the `if` statement). In contrast, the example

```
class Test
{
   double x;
   void F(bool b) {
      if (b) {
         x = 1.0;
      }
      else {
         int x ;
         x = 1;
      }
   }
}
```

is permitted because the name `x` is never used in the outer block.

> **Note** The rule of *invariant meaning* applies only to simple names. It is perfectly valid for the same identifier to have one meaning as a simple name and another meaning as right operand of a member access (Section 7.5.4). For example:

```
struct Point
{
   int x, y;
   public Point(int x, int y) {
      this.x = x;
      this.y = y;
   }
}
```

The example above illustrates a common pattern of using the names of fields as parameter names in an instance constructor. In the example, the simple names `x` and `y` refer to the parameters, but that does not prevent the member access expressions `this.x` and `this.y` from accessing the fields.

7.5.3 Parenthesized expressions

A parenthesized-expression consists of an expression enclosed in parentheses.

parenthesized-expression:
(*expression*)

A parenthesized-expression is evaluated by evaluating the *expression* within the parentheses. If the expression within the parentheses denotes a namespace, type, or method group, a compile-time error occurs. Otherwise, the result of the parenthesized-expression is the result of the evaluation of the contained expression.

7.5.4 Member access

A member-access consists of a primary-expression or a predefined-type, followed by a "`.`" token, followed by an identifier.

member-access:
  primary-expression `.` *identifier*
  predefined-type `.` *identifier*
predefined-type: one of
  bool byte char decimal double float int long
  object sbyte short string uint ulong ushort

A member-access of the form `E.I`, where `E` is a primary-expression or a predefined-type and `I` is an identifier, is evaluated and classified as follows:

- If `E` is a namespace and `I` is the name of an accessible member of that namespace, then the result is that member and, depending on the member, is classified as a namespace or a type.
- If `E` is a predefined-type or a primary-expression classified as a type, and a member lookup (Section 7.3) of `I` in `E` produces a match, then `E.I` is evaluated and classified as follows:
  - If `I` identifies a type, then the result is that type.
  - If `I` identifies one or more methods, then the result is a method group with no associated instance expression.
  - If `I` identifies a `static` property, then the result is a property access with no associated instance expression.
  - If `I` identifies a `static` field:

    If the field is `readonly` and the reference occurs outside the static constructor of the class or struct in which the field is declared, then the result is a value, namely the value of the static field `I` in `E`.

    Otherwise, the result is a variable, namely the static field `I` in `E`.

  - If `I` identifies a `static` event:

    If the reference occurs within the class or struct in which the event is declared, and the event was declared without event-accessor-declarations (Section 10.7), then `E.I` is processed exactly as if `I` was a static field.

    Otherwise, the result is an event access with no associated instance expression.

  - If `I` identifies a constant, then the result is a value, namely the value of that constant.
  - If `I` identifies an enumeration member, then the result is a value, namely the value of that enumeration member.
  - Otherwise, `E.I` is an invalid member reference, and a compile-time error occurs.
- If `E` is a property access, indexer access, variable, or value, the type of which is `T`, and a member lookup (Section 7.3) of `I` in `T` produces a match, then `E.I` is evaluated and classified as follows:
  - First, if `E` is a property or indexer access, then the value of the property or indexer access is obtained (Section 7.1.1) and `E` is reclassified as a value.
  - If `I` identifies one or more methods, then the result is a method group with an associated instance expression of `E`.
  - If `I` identifies an instance property, then the result is a property access with an associated instance expression of `E`.
  - If `T` is a *class-type* and `I` identifies an instance field of that *class-type*:

    If the value of `E` is `null`, then a `System.NullReferenceException` is thrown.

    Otherwise, if the field is `readonly` and the reference occurs outside an instance constructor of the class in which the field is declared, then the result is a value, namely the value of the field `I` in the object referenced by `E`.

    Otherwise, the result is a variable, namely the field `I` in the object referenced by `E`.

  - If `T` is a struct-type and `I` identifies an instance field of that struct-type:

    If `E` is a value, or if the field is `readonly` and the reference occurs outside an instance constructor of the struct in which the field is declared, then the result is a value, namely the value of the field `I` in the struct instance given by `E`.

    Otherwise, the result is a variable, namely the field `I` in the struct instance given by `E`.

  - If `I` identifies an instance event:

If the reference occurs within the class or struct in which the event is declared, and the event was declared without event-accessor-declarations (Section 10.7), then `E.I` is processed exactly as if `I` was an instance field.

Otherwise, the result is an event access with an associated instance expression of `E`.

- Otherwise, `E.I` is an invalid member reference, and a compile-time error occurs.

7.5.4.1 Identical simple names and type names

In a member access of the form `E.I`, if `E` is a single identifier, and if the meaning of `E` as a simple-name (Section 7.5.2) is a constant, field, property, local variable, or parameter with the same type as the meaning of `E` as a type-name (Section 3.8), then both possible meanings of `E` are permitted. The two possible meanings of `E.I` are never ambiguous, since `I` must necessarily be a member of the type `E` in both cases. In other words, the rule simply permits access to the static members and nested types of `E` where a compile-time error would otherwise have occurred. For example:

```
struct Color
{
   public static readonly Color White = new Color(...);
   public static readonly Color Black = new Color(...);
   public Color Complement() {...}
}
class A
{
   public Color Color;               // Field Color of type Color
   void F() {
      Color = Color.Black;          // References Color.Black static member
      Color = Color.Complement();   // Invokes Complement() on Color field
   }
   static void G() {
      Color c = Color.White;         // References Color.White static member
   }
}
```

Within the `A` class, those occurrences of the `Color` identifier that reference the `Color` type are underlined, and those that reference the `Color` field are not underlined.

7.5.5 Invocation expressions

An invocation-expression is used to invoke a method.

invocation-expression:
  primary-expression (*argument-list*$_{opt}$)

The primary-expression of an invocation-expression must be a method group or a value of a *delegate-type*. If the primary-expression is a method group, the invocation-expression is a method invocation (Section 7.5.5.1). If the primary-expression is a value of a *delegate-type*, the invocation-expression is a delegate invocation (Section 7.5.5.2). If the primary-expression is neither a method group nor a value of a *delegate-type*, a compile-time error occurs.

The optional argument-list (Section 7.4.1) provides values or variable references for the parameters of the method.

The result of evaluating an invocation-expression is classified as follows:

- If the invocation-expression invokes a method or delegate that returns `void`, the result is nothing. An expression that is classified as nothing cannot be an operand of any operator, and is permitted only in the context of a statement-expression (Section 8.6).
- Otherwise, the result is a value of the type returned by the method or delegate.

7.5.5.1 Method invocations

For a method invocation, the primary-expression of the invocation-expression must be a method group. The method group identifies the one method to invoke or the set of overloaded methods from which to choose a specific method to invoke. In the latter case, determination of the specific method to invoke is based on the context provided by the types of the arguments in the argument-list.

The compile-time processing of a method invocation of the form `M(A)`, where `M` is a method group and `A` is an optional argument-list, consists of the following steps:

- The set of candidate methods for the method invocation is constructed. Starting with the set of methods associated with `M`, which were found by a previous member lookup (Section 7.3), the set is reduced to those methods that are applicable with respect to the argument list `A`. The set reduction consists of applying the following rules to each method `T.N` in the set, where `T` is the type in which the method `N` is declared:
  - If `N` is not applicable with respect to `A` (Section 7.4.2.1), then `N` is removed from the set.
  - If `N` is applicable with respect to `A` (Section 7.4.2.1), then all methods declared in a base type of `T` are removed from the set.
- If the resulting set of candidate methods is empty, then no applicable methods exist, and a compile-time error occurs. If the candidate methods are not all declared in the same type, the method invocation is ambiguous, and a compile-time error occurs (this latter situation can only occur for an invocation of a method in an interface that has multiple direct base interfaces, as described in Section 13.2.5).
- The best method of the set of candidate methods is identified using the overload resolution rules of Section 7.4.2. If a single best method cannot be identified, the method invocation is ambiguous, and a compile-time error occurs.
- Given a best method, the invocation of the method is validated in the context of the method group: If the best method is a static method, the method group must have resulted from a simple-name or a member-access through a type. If the best method is an instance method, the method group must have resulted from a simple-name, a member-access through a variable or value, or a base-access. If neither requirement is true, a compile-time error occurs.

Once a method has been selected and validated at compile-time by the above steps, the actual run-time invocation is processed according to the rules of function member invocation described in Section 7.4.3.

The intuitive effect of the resolution rules described above is as follows: To locate the particular method invoked by a method invocation, start with the type indicated by the method invocation and proceed up the inheritance chain until at least one applicable, accessible, non-override method declaration is found. Then perform overload resolution on the set of applicable, accessible, non-override methods declared in that type and invoke the method thus selected.

7.5.5.2 Delegate invocations

For a delegate invocation, the primary-expression of the invocation-expression must be a value of a *delegate-type*. Furthermore, considering the *delegate-type* to be a function member with the same parameter list as the *delegate-type*, the *delegate-type* must be applicable (Section 7.4.2.1) with respect to the argument-list of the invocation-expression.

The run-time processing of a delegate invocation of the form `D(A)`, where `D` is a primary-expression of a *delegate-type* and `A` is an optional argument-list, consists of the following steps:

- `D` is evaluated. If this evaluation causes an exception, no further steps are executed.
- The value of `D` is checked to be valid. If the value of `D` is `null`, a `System.NullReferenceException` is thrown and no further steps are executed.
- Otherwise, `D` is a reference to a delegate instance. Function member invocations (Section 7.4.3) are performed on each of the callable entities in the invocation list of the delegate. For callable entities consisting of an instance and instance method, the instance for the invocation is the instance contained in the callable entity.

7.5.6 Element access

An element-access consists of a primary-no-array-creation-expression, followed by a "[" token, followed by an expression-list, followed by a "]" token. The expression-list consists of one or more expressions, separated by commas.

element-access:
  primary-no-array-creation-expression [*expression-list*]
expression-list:
  expression
  expression-list , *expression*

If the primary-no-array-creation-expression of an element-access is a value of an *array-type*, the element-access is an array access (Section 7.5.6.1). Otherwise, the primary-no-array-creation-expression must be a variable or value of a class, struct, or interface type that has one or more indexer members, in which case the element-access is an indexer access (Section 7.5.6.2).

7.5.6.1 Array access

For an array access, the primary-no-array-creation-expression of the element-access must be a value of an *array-type*. The number of expressions in the expression-list must be the same as the rank of the *array-type*, and each expression must be of type `int`, `uint`, `long`, `ulong`, or of a type that can be implicitly converted to one or more of these types.

The result of evaluating an array access is a variable of the element type of the array, namely the array element selected by the value(s) of the expression(s) in the expression-list.

The run-time processing of an array access of the form `P[A]`, where `P` is a primary-no-array-creation-expression of an *array-type* and `A` is an expression-list, consists of the following steps:

- `P` is evaluated. If this evaluation causes an exception, no further steps are executed.
- The index expressions of the expression-list are evaluated in order, from left to right. Following evaluation of each index expression, an implicit conversion (Section 6.1) to one of the following types is performed: `int`, `uint`, `long`, `ulong`. The first type in this list for which an implicit conversion exists is chosen. For instance, if the index expression is of type `short` then an implicit conversion to `int` is performed, since implicit conversions from `short` to `int` and from `short` to `long` are possible. If evaluation of an index expression or the subsequent implicit conversion causes an exception, then no further index expressions are evaluated and no further steps are executed.
- The value of `P` is checked to be valid. If the value of `P` is `null`, a `System.NullReferenceException` is thrown and no further steps are executed.
- The value of each expression in the *expression-list* is checked against the actual bounds of each dimension of the array instance referenced by `P`. If one or more values are out of range, a `System.IndexOutOfRangeException` is thrown and no further steps are executed.
- The location of the array element given by the index expression(s) is computed, and this location becomes the result of the array access.

7.5.6.2 Indexer access

For an indexer access, the primary-no-array-creation-expression of the element-access must be a variable or value of a class, struct, or interface type, and this type must implement one or more indexers that are applicable with respect to the expression-list of the element-access.

The compile-time processing of an indexer access of the form `P[A]`, where `P` is a primary-no-array-creation-expression of a class, struct, or interface type `T`, and `A` is an expression-list, consists of the following steps:

- The set of indexers provided by `T` is constructed. The set consists of all indexers declared in `T` or a base type of `T` that are not `override` declarations and are accessible in the current context (Section 3.5).
- The set is reduced to those indexers that are applicable and not hidden by other indexers. The following rules are applied to each indexer `S.I` in the set, where `S` is the type in which the indexer `I` is declared:
  - If `I` is not applicable with respect to `A` (Section 7.4.2.1), then `I` is removed from the set.
  - If `I` is applicable with respect to `A` (Section 7.4.2.1), then all indexers declared in a base type of `S` are removed from the set.
- If the resulting set of candidate indexers is empty, then no applicable indexers exist, and a compile-time error occurs. If the candidate indexers are not all declared in the same type, the indexer access is ambiguous, and a compile-time error occurs (this latter situation can only occur for an indexer access on an instance of an interface that has multiple direct base interfaces).
- The best indexer of the set of candidate indexers is identified using the overload resolution rules of Section 7.4.2. If a single best indexer cannot be identified, the indexer access is ambiguous, and a compile-time error occurs.
- The index expressions of the expression-list are evaluated in order, from left to right. The result of processing the indexer access is an expression classified as an indexer access. The indexer access expression references the indexer determined in the step above, and has an associated instance expression of `P` and an associated argument list of `A`.

Depending on the context in which it is used, an indexer access causes invocation of either the get-accessor or the set-accessor of the indexer. If the indexer access is the target of an assignment, the set-accessor is invoked to assign a new value (Section 7.13.1). In all other cases, the get-accessor is invoked to obtain the current value (Section 7.1.1).

7.5.7 This access

A this-access consists of the reserved word `this`.

this-access:
  this

A this-access is permitted only in the *block* of an instance constructor, an instance method, or an instance accessor. It has one of the following meanings:

- When `this` is used in a *primary-expression* within an instance constructor of a class, it is classified as a value. The type of the value is the class within which the usage occurs, and the value is a reference to the object being constructed.
- When `this` is used in a *primary-expression* within an instance method or instance accessor of a class, it is classified as a value. The type of the value is the class within which the usage occurs, and the value is a reference to the object for which the method or accessor was invoked.
- When `this` is used in a *primary-expression* within an instance constructor of a struct, it is classified as a variable. The type of the variable is the struct within which the usage occurs, and the variable represents the struct being constructed. The `this` variable of an instance constructor of a struct behaves exactly the same as an `out` parameter of the struct type — in particular, this means that the variable must be definitely assigned in every execution path of the instance constructor.
- When `this` is used in a *primary-expression* within an instance method or instance accessor of a struct, it is classified as a variable. The type of the variable is the struct within which the usage occurs, and the variable represents the struct for which the method or accessor was invoked. The `this` variable of an instance method of a struct behaves exactly the same as a `ref` parameter of the struct type.

Use of `this` in a *primary-expression* in a context other than the ones listed above is a compile-time error. In particular, it is not possible to refer to `this` in a static method, a static property accessor, or in a *variable-initializer* of a field declaration.

7.5.8 Base access

A *base-access* consists of the reserved word `base` followed by either a "`.`" token and an identifier or an expression-list enclosed in square brackets:

base-access:
  base . *identifier*
  base [*expression-list*]

A *base-access* is used to access base class members that are hidden by similarly named members in the current class or struct. A *base-access* is permitted only in the block of an instance constructor, an instance method, or an instance accessor. When `base.I` occurs in a class or struct, `I` must denote a member of the base class of that class or struct. Likewise, when `base[E]` occurs in a class, an applicable indexer must exist in the base class.

At compile-time, *base-access* expressions of the form `base.I` and `base[E]` are evaluated exactly as if they were written `((B)this).I` and `((B)this)[E]`, where `B` is the base class of the class or struct in which the construct occurs. Thus, `base.I` and `base[E]` correspond to `this.I` and `this[E]`, except `this` is viewed as an instance of the base class.

When a *base-access* references a virtual function member (a method, property, or indexer), the determination of which function member to invoke at run-time (Section 7.4.3) is changed. The function member that is invoked is determined by finding the most derived implementation (Section 10.5.3) of the function member with respect to `B` (instead of with respect to the run-time type of `this`, as would be usual in a non-base access). Thus, within an `override` of a `virtual` function member, a *base-access* can be used to invoke the inherited implementation of the function member. If the function member referenced by a *base-access* is abstract, a compile-time error occurs.

7.5.9 Postfix increment and decrement operators

post-increment-expression:
  primary-expression `++`
post-decrement-expression:
  primary-expression `--`

The operand of a postfix increment or decrement operation must be an expression classified as a variable, a property access, or an indexer access. The result of the operation is a value of the same type as the operand.

If the operand of a postfix increment or decrement operation is a property or indexer access, the property or indexer must have both a `get` and a `set` accessor. If this is not the case, a compile-time error occurs.

Unary operator overload resolution (Section 7.2.3) is applied to select a specific operator implementation. Predefined `++` and `--` operators exist for the following types: `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, `decimal`, and any enum type. The predefined `++` operators return the value produced by adding 1 to the operand, and the predefined `--` operators return the value produced by subtracting 1 from the operand.

The run-time processing of a postfix increment or decrement operation of the form `x++` or `x--` consists of the following steps:

- If `x` is classified as a variable:
  - `x` is evaluated to produce the variable.
  - The value of `x` is saved.
  - The selected operator is invoked with the saved value of `x` as its argument.
  - The value returned by the operator is stored in the location given by the evaluation of `x`.
  - The saved value of `x` becomes the result of the operation.
- If `x` is classified as a property or indexer access:
  - The instance expression (if `x` is not `static`) and the argument list (if `x` is an indexer access) associated with `x` are evaluated, and the results are used in the subsequent `get` and `set` accessor invocations.
  - The `get` accessor of `x` is invoked and the returned value is saved.
  - The selected operator is invoked with the saved value of `x` as its argument.
  - The `set` accessor of `x` is invoked with the value returned by the operator as its `value` argument.
  - The saved value of `x` becomes the result of the operation.

The `++` and `--` operators also support prefix notation, as described in Section 7.6.5. The result of `x++` or `x--` is the value of `x` *before* the operation, whereas the result of `++x` or `--x` is the value of `x` *after* the operation. In either case, `x` itself has the same value after the operation.

An `operator ++` or `operator --` implementation can be invoked using either postfix or prefix notation. It is not possible to have separate operator implementations for the two notations.

7.5.10 The new operator

The `new` operator is used to create new instances of types.

There are three forms of `new` expressions:

- Object creation expressions are used to create new instances of class types and value types.
- Array creation expressions are used to create new instances of array types.
- Delegate creation expressions are used to create new instances of delegate types.

The `new` operator implies creation of an instance of a type, but does not necessarily imply dynamic allocation of memory. In particular, instances of value types require no additional memory beyond the variables in which they reside, and no dynamic allocations occur when `new` is used to create instances of value types.

7.5.10.1 Object creation expressions

An *object-creation-expression* is used to create a new instance of a *class-type* or a *value-type*.

object-creation-expression:
  new *type* (*argument-list*$_{opt}$)

The type of an object-creation-expression must be a *class-type* or a *value-type*. The type cannot be an `abstract` *class-type*.

The optional argument-list (Section 7.4.1) is permitted only if the type is a *class-type* or a struct-type.

The compile-time processing of an object-creation-expression of the form `new T(A)`, where `T` is a *class-type* or a *value-type* and `A` is an optional argument-list, consists of the following steps:

- If `T` is a *value-type* and `A` is not present:
  - The object-creation-expression is a default constructor invocation. The result of the object-creation-expression is a value of type `T`, namely the default value for `T` as defined in Section 4.1.1.
- Otherwise, if `T` is a *class-type* or a struct-type:
  - If `T` is an `abstract` *class-type*, a compile-time error occurs.
  - The instance constructor to invoke is determined using the overload resolution rules of Section 7.4.2. The set of candidate instance constructors consists of all accessible instance constructors declared in `T`, which are applicable with respect to `A` (Section 7.4.2.1). If the set of candidate instance constructors is empty, or if a single best instance constructor cannot be identified, a compile-time error occurs.
  - The result of the object-creation-expression is a value of type `T`, namely the value produced by invoking the instance constructor determined in the step above.
- Otherwise, the object-creation-expression is invalid, and a compile-time error occurs.

The run-time processing of an object-creation-expression of the form `new T(A)`, where `T` is *class-type* or a struct-type and `A` is an optional argument-list, consists of the following steps:

- If `T` is a *class-type*:
  - A new instance of class `T` is allocated. If there is not enough memory available to allocate the new instance, a `System.OutOfMemoryException` is thrown and no further steps are executed.
  - All fields of the new instance are initialized to their default values (Section 5.2).
  - The instance constructor is invoked according to the rules of function member invocation (Section 7.4.3). A reference to the newly allocated instance is automatically passed to the instance constructor and the instance can be accessed from within that constructor as `this`.
- If `T` is a struct-type:
  - An instance of type `T` is created by allocating a temporary local variable. Since an instance constructor of a struct-type is required to definitely assign a value to each field of the instance being created, no initialization of the temporary variable is necessary.
  - The instance constructor is invoked according to the rules of function member invocation (Section 7.4.3). A reference to the newly allocated instance is automatically passed to the instance constructor and the instance can be accessed from within that constructor as `this`.

7.5.10.2 Array creation expressions

An array-creation-expression is used to create a new instance of an *array-type*.

array-creation-expression:
  new *non-array-type* [*expression-list*] *rank-specifiers*$_{opt}$ *array-initializer*$_{opt}$
  new *array-type* *array-initializer*

An array creation expression of the first form allocates an array instance of the type that results from deleting each of the individual expressions from the expression list. For example, the array creation expression `new int[10, 20]` produces an array instance of type `int[,]`, and the array creation expression `new int[10][,]` produces an array of type `int[][,]`. Each expression in the expression list must be of type `int`, `uint`, `long`, or `ulong`, or of a type that can be implicitly converted to one or more of these types. The value of each expression determines the length of the corresponding dimension in the newly allocated array instance. Since the length of an array dimension must be nonnegative, it is a compile-time error to have a constant-expression with a negative value in the expression list.

Except in an unsafe context (Section A.1), the layout of arrays is unspecified.

If an array creation expression of the first form includes an array initializer, each expression in the expression list must be a constant and the rank and dimension lengths specified by the expression list must match those of the array initializer.

In an array creation expression of the second form, the rank of the specified array type must match that of the array initializer. The individual dimension lengths are inferred from the number of elements in each of the corresponding nesting levels of the array initializer. Thus, the expression

```
new int[,] {{0, 1}, {2, 3}, {4, 5}}
```

exactly corresponds to

```
new int[3, 2] {{0, 1}, {2, 3}, {4, 5}}
```

Array initializers are described further in Section 12.6.

The result of evaluating an array creation expression is classified as a value, namely a reference to the newly allocated array instance. The run-time processing of an array creation expression consists of the following steps:

- The dimension length expressions of the expression-list are evaluated in order, from left to right. Following evaluation of each expression, an implicit conversion (Section 6.1) to one of the following types is performed: `int`, `uint`, `long`, `ulong`. The first type in this list for which an implicit conversion exists is chosen. If evaluation of an expression or the subsequent implicit conversion causes an exception, then no further expressions are evaluated and no further steps are executed.
- The computed values for the dimension lengths are validated as follows. If one or more of the values are less than zero, a `System.OverflowException` is thrown and no further steps are executed.
- An array instance with the given dimension lengths is allocated. If there is not enough memory available to allocate the new instance, a `System.OutOfMemoryException` is thrown and no further steps are executed.
- All elements of the new array instance are initialized to their default values (Section 5.2).
- If the array creation expression contains an array initializer, then each expression in the array initializer is evaluated and assigned to its corresponding array element. The evaluations and assignments are performed in the order the expressions are written in the array initializer — in other words, elements are initialized in increasing index order, with the rightmost dimension increasing first. If evaluation of a given expression or the subsequent assignment to the corresponding array element causes an exception, then no further elements are initialized (and the remaining elements will thus have their default values).

An array creation expression permits instantiation of an array with elements of an array type, but the elements of such an array must be manually initialized. For example, the statement

```
int[][] a = new int[100][];
```

creates a single-dimensional array with 100 elements of type `int[]`. The initial value of each element is `null`. It is not possible for the same array creation expression to also instantiate the sub-arrays, and the statement

```
int[][] a = new int[100][5];      // Error
```

results in a compile-time error. Instantiation of the sub-arrays must instead be performed manually, as in

```
int[][] a = new int[100][];
for (int i = 0; i < 100; i++) a[i] = new int[5];
```

When an array of arrays has a "rectangular" shape, that is when the sub-arrays are all of the same length, it is more efficient to use a multi-dimensional array. In the example above, instantiation of the array of arrays creates 101 objects — one outer array and 100 sub-arrays. In contrast,

```
int[,] = new int[100, 5];
```

creates only a single object, a two-dimensional array, and accomplishes the allocation in a single statement.

7.5.10.3 Delegate creation expressions

A delegate-creation-expression is used to create a new instance of a *delegate-type*.

delegate-creation-expression:
  new *delegate-type* (*expression*)

The argument of a delegate creation expression must be a method group (Section 7.1) or a value of a *delegate-type*. If the argument is a method group, it identifies the method and, for an instance method, the object for which to create a delegate. If the argument is a value of a *delegate-type*, it identifies a delegate instance of which to create a copy.

The compile-time processing of a delegate-creation-expression of the form `new D(E)`, where `D` is a *delegate-type* and `E` is an expression, consists of the following steps:

- If `E` is a method group:
  - The set of methods identified by `E` must include exactly one method that is compatible (Section 15.1) with `D`, and this method becomes the one to which the newly created delegate refers. If no matching method exists, or if more than one matching method exists, a compile-time error occurs. If the selected method is an instance method, the instance expression associated with `E` determines the target object of the delegate.
  - As in a method invocation, the selected method must be compatible with the context of the method group: If the method is a static method, the method group must have resulted from a simple-name or a member-access through a type. If the method is an instance method, the method group must have resulted from a simple-name or a member-access through a variable or value. If the selected method does not match the context of the method group, a compile-time error occurs.
  - The result is a value of type `D`, namely a newly created delegate that refers to the selected method and target object.
- Otherwise, if `E` is a value of a *delegate-type*:
  - `D` and `E` must be compatible (Section 15.1); otherwise, a compile-time error occurs.
  - The result is a value of type `D`, namely a newly created delegate that refers to the same invocation list as `E`.
- Otherwise, the delegate creation expression is invalid, and a compile-time error occurs.

The run-time processing of a delegate-creation-expression of the form `new D(E)`, where `D` is a *delegate-type* and `E` is an expression, consists of the following steps:

- If `E` is a method group:
  - If the method selected at compile-time is a static method, the target object of the delegate is `null`. Otherwise, the selected method is an instance method, and the target object of the delegate is determined from the instance expression associated with `E`:

    The instance expression is evaluated. If this evaluation causes an exception, no further steps are executed.

    If the instance expression is of a *reference-type*, the value computed by the instance expression becomes the target object. If the target object is `null`, a `System.NullReferenceException` is thrown and no further steps are executed.

    If the instance expression is of a *value-type*, a boxing operation (Section 4.3.1) is performed to convert the value to an object, and this object becomes the target object.
  - A new instance of the delegate type `D` is allocated. If there is not enough memory available to allocate the new instance, a `System.OutOfMemoryException` is thrown and no further steps are executed.
  - The new delegate instance is initialized with a reference to the method that was determined at compile-time and a reference to the target object computed above.
- If `E` is a value of a *delegate-type*:
  - `E` is evaluated. If this evaluation causes an exception, no further steps are executed.
  - If the value of `E` is `null`, a `System.NullReferenceException` is thrown and no further steps are executed.
  - A new instance of the delegate type `D` is allocated. If there is not enough memory available to allocate the new instance, a `System.OutOfMemoryException` is thrown and no further steps are executed.
  - The new delegate instance is initialized with the same invocation list as the delegate instance given by `E`.

The invocation list of a delegate is determined when the delegate is instantiated and then remains constant for the entire lifetime of the delegate. In other words, it is not possible to change the target callable entities of a delegate once it has been created. When

two delegates are combined or one is removed from another (Section 15.1), a new delegate results; no existing delegate has its contents changed.

It is not possible to create a delegate that refers to a property, indexer, user-defined operator, instance constructor, destructor, or static constructor.

As described above, when a delegate is created from a method group, the formal parameter list and return type of the delegate determine which of the overloaded methods to select. In the example

```
delegate double DoubleFunc(double x);
class A
{
   DoubleFunc f = new DoubleFunc(Square);
   static float Square(float x) {
      return x * x;
   }
   static double Square(double x) {
      return x * x;
   }
}
```

the `A.f` field is initialized with a delegate that refers to the second `Square` method because that method exactly matches the formal parameter list and return type of `DoubleFunc`. Had the second `Square` method not been present, a compile-time error would have occurred.

7.5.11 The typeof operator

The `typeof` operator is used to obtain the `System.Type` object for a type.

typeof-expression:
  typeof (*type*)
  typeof (void)

The first form of typeof-expression consists of a `typeof` keyword followed by a parenthesized *type*. The result of an expression of this form is the `System.Type` object for the indicated type. There is only one `System.Type` object for any given type. This means that for type `T`, `typeof(T) == typeof(T)` is always true.

The second form of typeof-expression consists of a `typeof` keyword followed by a parenthesized `void` keyword. The result of an expression of this form is the `System.Type` object that represents the absence of a type. The type object returned by `typeof(void)` is distinct from the type object returned for any type. This special type object is useful in class libraries that allow reflection onto methods in the language, where those methods wish to have a way to represent the return type of any method, including void methods, with an instance of `System.Type`.

The example

```
using System;
class Test
{
   static void Main() {
      Type[] t = {
         typeof(int),
         typeof(System.Int32),
         typeof(string),
         typeof(double[]),
         typeof(void)
      };
      for (int i = 0; i < t.Length; i++) {
         Console.WriteLine(t[i].FullName);
      }
   }
}
```

produces the following output:

```
System.Int32
System.Int32
System.String
System.Double[]
System.Void
```

Note that `int` and `System.Int32` are the same type.

7.5.12 The checked and unchecked operators

The `checked` and `unchecked` operators are used to control the overflow checking context for integral-type arithmetic operations and conversions.

checked-expression:
  checked (*expression*)
unchecked-expression:
  unchecked (*expression*)

The `checked` operator evaluates the contained expression in a checked context, and the `unchecked` operator evaluates the contained expression in an unchecked context. A checked-expression or unchecked-expression corresponds exactly to a parenthesized-expression (Section 7.5.3), except that the contained expression is evaluated in the given overflow checking context.

The overflow checking context can also be controlled through the `checked` and `unchecked` statements (Section 8.11).

The following operations are affected by the overflow checking context established by the `checked` and `unchecked` operators and statements:

- The predefined `++` and `--` unary operators (Section 7.5.9 and Section 7.6.5), when the operand is of an integral type.
- The predefined `-` unary operator (Section 7.6.2), when the operand is of an integral type.
- The predefined `+`, `-`, `*`, and `/` binary operators (Section 7.7), when both operands are of integral types.
- Explicit numeric conversions (Section 6.2.1) from one integral type to another integral type, or from `float` or `double` to an integral type.

When one of the above operations produce a result that is too large to represent in the destination type, the context in which the operation is performed controls the resulting behavior:

- In a `checked` context, if the operation is a constant expression (Section 7.15), a compile-time error occurs. Otherwise, when the operation is performed at run-time, a `System.OverflowException` is thrown.
- In an `unchecked` context, the result is truncated by discarding any high-order bits that do not fit in the destination type.

For non-constant expressions (expressions that are evaluated at run-time) that are not enclosed by any `checked` or `unchecked` operators or statements, the default overflow checking context is `unchecked` unless external factors (such as compiler switches and execution environment configuration) call for `checked` evaluation.

For constant expressions (expressions that can be fully evaluated at compile-time), the default overflow checking context is always `checked`. Unless a constant expression is explicitly placed in an `unchecked` context, overflows that occur during the compile-time evaluation of the expression always cause compile-time errors.

In the example

```
class Test
{
   static readonly int x = 1000000;
   static readonly int y = 1000000;
   static int F() {
      return checked(x * y);      // Throws OverflowException
   }
   static int G() {
      return unchecked(x * y);   // Returns -727379968
   }
   static int H() {
      return x * y;               // Depends on default
   }
}
```

no compile-time errors are reported since neither of the expressions can be evaluated at compile-time. At run-time, the `F` method throws a `System.OverflowException`, and the `G` method returns –727379968 (the lower 32 bits of the out-of-range result). The behavior of the `H` method depends on the default overflow checking context for the compilation, but it is either the same as `F` or the same as `G`.

In the example

```
class Test
{
   const int x = 1000000;
   const int y = 1000000;
   static int F() {
      return checked(x * y);      // Compile error, overflow
   }
   static int G() {
      return unchecked(x * y);   // Returns -727379968
   }
   static int H() {
      return x * y;               // Compile error, overflow
   }
}
```

the overflows that occur when evaluating the constant expressions in `F` and `H` cause compile-time errors to be reported because the expressions are evaluated in a `checked` context. An overflow also occurs when evaluating the constant expression in `G`, but since the evaluation takes place in an `unchecked` context, the overflow is not reported.

The `checked` and `unchecked` operators only affect the overflow checking context for those operations that are textually contained within the "`(`" and "`)`" tokens. The operators have no effect on function members that are invoked as a result of evaluating the contained expression. In the example

```
class Test
{
   static int Multiply(int x, int y) {
      return x * y;
   }
   static int F() {
      return checked(Multiply(1000000, 1000000));
   }
}
```

the use of `checked` in `F` does not affect the evaluation of `x * y` in `Multiply`, so `x * y` is evaluated in the default overflow checking context.

The `unchecked` operator is convenient when writing constants of the signed integral types in hexadecimal notation. For example:

```
class Test
{
   public const int AllBits = unchecked((int)0xFFFFFFFF);
   public const int HighBit = unchecked((int)0x80000000);
}
```

Both of the hexadecimal constants above are of type `uint`. Because the constants are outside the `int` range, without the `unchecked` operator, the casts to `int` would produce compile-time errors.

The `checked` and `unchecked` operators and statements allow programmers to control certain aspects of some numeric calculations. However, the behavior of some numeric operators depends on their operands' data types. For example, multiplying two decimals always results in an exception on overflow even within an explicitly `unchecked` construct. Similarly, multiplying two floats never results in an exception on overflow even within an explicitly `checked` construct. In addition, other operators are *never* affected by the mode of checking, whether default or explicit.

7.6 Unary operators

The +, –, !, ~, ++, ––, and cast operators are called the unary operators.

unary-expression:
  primary-expression
  + *unary-expression*
  - *unary-expression*
  ! *unary-expression*
  ~ *unary-expression*
  pre-increment-expression
  pre-decrement-expression
  cast-expression

7.6.1 Unary plus operator

For an operation of the form `+x`, unary operator overload resolution (Section 7.2.3) is applied to select a specific operator implementation. The operand is converted to the parameter type of the selected operator, and the type of the result is the return type of the operator. The predefined unary plus operators are:

```
int operator +(int x);
uint operator +(uint x);
long operator +(long x);
ulong operator +(ulong x);
float operator +(float x);
double operator +(double x);
decimal operator +(decimal x);
```

For each of these operators, the result is simply the value of the operand.

7.6.2 Unary minus operator

For an operation of the form `-x`, unary operator overload resolution (Section 7.2.3) is applied to select a specific operator implementation. The operand is converted to the parameter type of the selected operator, and the type of the result is the return type of the operator. The predefined negation operators are:

- Integer negation:

```
int operator -(int x);
long operator -(long x);
```

The result is computed by subtracting `x` from zero. If the value of `x` is the smallest representable value of the operand type (-2^{31} for `int` or -2^{63} for `long`), then the mathematical negation of x is not representable within the operand type. If this occurs within a `checked` context, a `System.OverflowException` is thrown; if it occurs within an `unchecked` context, the result is the value of the operand and the overflow is not reported.

If the operand of the negation operator is of type `uint`, it is converted to type `long`, and the type of the result is `long`. An exception is the rule that permits the `int` value −2147483648 (-2^{31}) to be written as a decimal integer literal (Section 2.4.4.2).

If the operand of the negation operator is of type `ulong`, a compile-time error occurs. An exception is the rule that permits the `long` value −9223372036854775808 (-2^{63}) to be written as a decimal integer literal (Section 2.4.4.2).

- Floating-point negation:

```
float operator -(float x);
double operator -(double x);
```

The result is the value of `x` with its sign inverted. If `x` is NaN, the result is also NaN.

- Decimal negation:

```
decimal operator -(decimal x);
```

The result is computed by subtracting `x` from zero. Decimal negation is equivalent to using the unary minus operator of type `System.Decimal`.

7.6.3 Logical negation operator

For an operation of the form `!x`, unary operator overload resolution (Section 7.2.3) is applied to select a specific operator implementation. The operand is converted to the parameter type of the selected operator, and the type of the result is the return type of the operator. Only one predefined logical negation operator exists:

```
bool operator !(bool x);
```

This operator computes the logical negation of the operand: If the operand is `true`, the result is `false`. If the operand is `false`, the result is `true`.

7.6.4 Bitwise complement operator

For an operation of the form `~x`, unary operator overload resolution (Section 7.2.3) is applied to select a specific operator implementation. The operand is converted to the parameter type of the selected operator, and the type of the result is the return type of the operator. The predefined bitwise complement operators are:

```
int operator ~(int x);
uint operator ~(uint x);
long operator ~(long x);
ulong operator ~(ulong x);
```

For each of these operators, the result of the operation is the bitwise complement of `x`.

Every enumeration type `E` implicitly provides the following bitwise complement operator:

```
E operator ~(E x);
```

The result of evaluating `~x`, where `x` is an expression of an enumeration type `E` with an underlying type `U`, is exactly the same as evaluating (E) `(~(U)x)`.

7.6.5 Prefix increment and decrement operators

pre-increment-expression:
  `++` *unary-expression*
pre-decrement-expression:
  `--` *unary-expression*

The operand of a prefix increment or decrement operation must be an expression classified as a variable, a property access, or an indexer access. The result of the operation is a value of the same type as the operand.

If the operand of a prefix increment or decrement operation is a property or indexer access, the property or indexer must have both a `get` and a `set` accessor. If this is not the case, a compile-time error occurs.

Unary operator overload resolution (Section 7.2.3) is applied to select a specific operator implementation. Predefined `++` and `--` operators exist for the following types: `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, `decimal`, and any enum type. The predefined `++` operators return the value produced by adding 1 to the operand, and the predefined `--` operators return the value produced by subtracting 1 from the operand.

The run-time processing of a prefix increment or decrement operation of the form `++x` or `--x` consists of the following steps:

- If `x` is classified as a variable:
  - `x` is evaluated to produce the variable.
  - The selected operator is invoked with the value of `x` as its argument.
  - The value returned by the operator is stored in the location given by the evaluation of `x`.
  - The value returned by the operator becomes the result of the operation.
- If `x` is classified as a property or indexer access:
  - The instance expression (if `x` is not `static`) and the argument list (if `x` is an indexer access) associated with `x` are evaluated, and the results are used in the subsequent `get` and `set` accessor invocations.
  - The `get` accessor of `x` is invoked.
  - The selected operator is invoked with the value returned by the `get` accessor as its argument.
  - The `set` accessor of `x` is invoked with the value returned by the operator as its `value` argument.
  - The value returned by the operator becomes the result of the operation.

The `++` and `--` operators also support postfix notation, (Section 7.5.9). The result of `x++` or `x--` is the value of `x` *before* the operation, whereas the result of `++x` or `--x` is the value of `x` *after* the operation. In either case, `x` itself has the same value after the operation.

An `operator ++` or `operator --` implementation can be invoked using either postfix or prefix notation. It is not possible to have separate operator implementations for the two notations.

7.6.6 Cast expressions

A cast-expression is used to explicitly convert an expression to a given type.

cast-expression:
  (*type*) *unary-expression*

A cast-expression of the form `(T)E`, where `T` is a *type* and `E` is a unary-expression, performs an explicit conversion (Section 6.2) of the value of `E` to type `T`. If no explicit conversion exists from the type of `E` to `T`, a compile-time error occurs. Otherwise, the result is the value produced by the explicit conversion. The result is always classified as a value, even if `E` denotes a variable.

The grammar for a cast-expression leads to certain syntactic ambiguities. For example, the expression `(x)-y` could either be interpreted as a cast-expression (a cast of `-y` to type `x`) or as an additive-expression combined with a parenthesized-expression (which computes the value `x - y)`.

To resolve cast-expression ambiguities, the following rule exists: A sequence of one or more tokens (Section 2.4) enclosed in parentheses is considered the start of a cast-expression only if at least one of the following are true:

- The sequence of tokens is correct grammar for a type, but not for an expression.
- The sequence of tokens is correct grammar for a type, and the token immediately following the closing parentheses is the token "`~`", the token "`!`", the token "`(`", an identifier (Section 2.4.1), a literal (Section 2.4.4), or any keyword (Section 2.4.3) except `as` and `is`.

The term "correct grammar" above means only that the sequence of tokens must conform to the particular grammatical production. It specifically does not consider the actual meaning of any constituent identifiers. For example, if `x` and `y` are identifiers, then `x.y` is correct grammar for a type, even if `x.y` doesn't actually denote a type.

From the disambiguation rule it follows that, if `x` and `y` are identifiers, `(x)y`, `(x)(y)`, and `(x)(-y)` are cast-expressions, but `(x)-y` is not, even if `x` identifies a type. However, if `x` is a keyword that identifies a predefined type (such as `int`), then all four forms are cast-expressions (because such a keyword could not possibly be an expression by itself).

7.7 Arithmetic operators

The `*`, `/`, `%`, `+`, and `–` operators are called the arithmetic operators.

multiplicative-expression:
  unary-expression
  multiplicative-expression `*` *unary-expression*
  multiplicative-expression `/` *unary-expression*
  multiplicative-expression `%` *unary-expression*
additive-expression:
  multiplicative-expression
  additive-expression `+` *multiplicative-expression*
  additive-expression `–` *multiplicative-expression*

7.7.1 Multiplication operator

For an operation of the form `x * y`, binary operator overload resolution (Section 7.2.4) is applied to select a specific operator implementation. The operands are converted to the parameter types of the selected operator, and the type of the result is the return type of the operator.

The predefined multiplication operators are listed below. The operators all compute the product of `x` and `y`.

- Integer multiplication:

```
int operator *(int x, int y);
uint operator *(uint x, uint y);
long operator *(long x, long y);
ulong operator *(ulong x, ulong y);
```

In a `checked` context, if the product is outside the range of the result type, a `System.OverflowException` is thrown. In an `unchecked` context, overflows are not reported and any significant high-order bits outside the range of the result type are discarded.

- Floating-point multiplication:

```
float operator *(float x, float y);
double operator *(double x, double y);
```

The product is computed according to the rules of IEEE 754 arithmetic. The following table lists the results of all possible combinations of nonzero finite values, zeros, infinities, and NaN's. In the table, `x` and `y` are positive finite values. `z` is the result of `x * y`. If the result is too large for the destination type, `z` is infinity. If the result is too small for the destination type, `z` is zero.

	+y	**–y**	**+0**	**–0**	**+∞**	**–∞**	**NaN**
+x	+z	-z	+0	-0	+8	-8	NaN
–x	-z	+z	-0	+0	-8	+8	NaN
+0	+0	-0	+0	-0	NaN	NaN	NaN
–0	-0	+0	-0	+0	NaN	NaN	NaN
+∞	+8	-8	NaN	NaN	+8	-8	NaN
–∞	-8	+8	NaN	NaN	-8	+8	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

- Decimal multiplication:

```
decimal operator *(decimal x, decimal y);
```

If the resulting value is too large to represent in the `decimal` format, a `System.OverflowException` is thrown. If the result value is too small to represent in the `decimal` format, the result is zero. The scale of the result, before any rounding, is the sum of the scales of the two operands.

Decimal multiplication is equivalent to using the multiplication operator of type `System.Decimal`.

7.7.2 Division operator

For an operation of the form `x / y`, binary operator overload resolution (Section 7.2.4) is applied to select a specific operator implementation. The operands are converted to the parameter types of the selected operator, and the type of the result is the return type of the operator.

The predefined division operators are listed below. The operators all compute the quotient of `x` and `y`.

- Integer division:

```
int operator /(int x, int y);
uint operator /(uint x, uint y);
long operator /(long x, long y);
ulong operator /(ulong x, ulong y);
```

If the value of the right operand is zero, a `System.DivideByZeroException` is thrown.

The division rounds the result towards zero, and the absolute value of the result is the largest possible integer that is less than the absolute value of the quotient of the two operands. The result is zero or positive when the two operands have the same sign and zero or negative when the two operands have opposite signs.

If the left operand is the smallest representable `int` or `long` value and the right operand is `-1`, an overflow occurs. A `System.OverflowException` is always thrown in this situation, regardless of whether the operation occurs in a `checked` or an `unchecked` context.

- Floating-point division:

```
float operator /(float x, float y);
double operator /(double x, double y);
```

The quotient is computed according to the rules of IEEE 754 arithmetic. The following table lists the results of all possible combinations of nonzero finite values, zeros, infinities, and NaN's. In the table, `x` and `y` are positive finite values. `z` is the result of `x / y`. If the result is too large for the destination type, `z` is infinity. If the result is too small for the destination type, `z` is zero.

	+y	**–y**	**+0**	**–0**	**+∞**	**–∞**	**NaN**
+x	+z	-z	+8	-8	+0	-0	NaN
–x	-z	+z	-8	+8	-0	+0	NaN
+0	+0	-0	NaN	NaN	+0	-0	NaN
–0	-0	+0	NaN	NaN	-0	+0	NaN
+∞	+8	-8	+8	-8	NaN	NaN	NaN
–∞	-8	+8	-8	+8	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

- Decimal division:

```
decimal operator /(decimal x, decimal y);
```

If the value of the right operand is zero, a `System.DivideByZeroException` is thrown. If the resulting value is too large to represent in the `decimal` format, a `System.OverflowException` is thrown. If the result value is too small to represent in the `decimal` format, the result is zero. The scale of the result is the smallest scale that will preserve a result equal to the nearest representable decimal value to the true mathematical result.

Decimal division is equivalent to using the division operator of type `System.Decimal`.

7.7.3 Remainder operator

For an operation of the form `x % y`, binary operator overload resolution (Section 7.2.4) is applied to select a specific operator implementation. The operands are converted to the parameter types of the selected operator, and the type of the result is the return type of the operator.

The predefined remainder operators are listed below. The operators all compute the remainder of the division between `x` and `y`.

- Integer remainder:

```
int operator %(int x, int y);
uint operator %(uint x, uint y);
long operator %(long x, long y);
ulong operator %(ulong x, ulong y);
```

The result of `x % y` is the value produced by `x - (x / y) * y`. If `y` is zero, a `System.DivideByZeroException` is thrown. The remainder operator never causes an overflow.

- Floating-point remainder:

```
float operator %(float x, float y);
double operator %(double x, double y);
```

The following table lists the results of all possible combinations of nonzero finite values, zeros, infinities, and NaN's. In the table, `x` and `y` are positive finite values. `z` is the result of `x % y` and is computed as `x - n * y`, where `n` is the largest possible integer that is less than or equal to `x / y`. This method of computing the remainder is analogous to that used for integer operands, but differs from the IEEE 754 definition (in which `n` is the integer closest to `x / y`).

	+y	**–y**	**+0**	**–0**	**+∞**	**–∞**	**NaN**
+x	+z	+z	NaN	NaN	x	x	NaN
–x	-z	-z	NaN	NaN	-x	-x	NaN
+0	+0	+0	NaN	NaN	+0	+0	NaN
–0	-0	-0	NaN	NaN	-0	-0	NaN
+∞	NaN	NaN	NaN	NaN	NaN	NaN	NaN
–∞	NaN	NaN	NaN	NaN	NaN	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

- Decimal remainder:

```
decimal operator %(decimal x, decimal y);
```

If the value of the right operand is zero, a `System.DivideByZeroException` is thrown. The scale of the result, before any rounding, is the larger of the scales of the two operands, and the sign of the result, if non-zero, is the same as that of `x`.

Decimal remainder is equivalent to using the remainder operator of type `System.Decimal`.

7.7.4 Addition operator

For an operation of the form `x + y`, binary operator overload resolution (Section 7.2.4) is applied to select a specific operator implementation. The operands are converted to the parameter types of the selected operator, and the type of the result is the return type of the operator.

The predefined addition operators are listed below. For numeric and enumeration types, the predefined addition operators compute the sum of the two operands. When one or both operands are of type string, the predefined addition operators concatenate the string representation of the operands.

- Integer addition:

```
int operator +(int x, int y);
uint operator +(uint x, uint y);
long operator +(long x, long y);
ulong operator +(ulong x, ulong y);
```

In a `checked` context, if the sum is outside the range of the result type, a `System.OverflowException` is thrown. In an `unchecked` context, overflows are not reported and any significant high-order bits outside the range of the result type are discarded.

- Floating-point addition:

```
float operator +(float x, float y);
double operator +(double x, double y);
```

The sum is computed according to the rules of IEEE 754 arithmetic. The following table lists the results of all possible combinations of nonzero finite values, zeros, infinities, and NaN's. In the table, `x` and `y` are nonzero finite values, and `z` is the result of `x + y`. If `x` and `y` have the same magnitude but opposite signs, `z` is positive zero. If `x + y` is too large to represent in the destination type, `z` is an infinity with the same sign as `x + y`.

	y	**+0**	**–0**	**+∞**	**–∞**	**NaN**
x	z	x	x	+∞	–∞	NaN
+0	y	+0	+0	+∞	–∞	NaN
–0	y	+0	–0	+∞	–∞	NaN
+∞	+∞	+∞	+∞	+∞	NaN	NaN
–∞	–∞	–∞	–∞	NaN	–∞	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN

- Decimal addition:

```
decimal operator +(decimal x, decimal y);
```

If the resulting value is too large to represent in the `decimal` format, a `System.OverflowException` is thrown. The scale of the result, before any rounding, is the larger of the scales of the two operands.

Decimal addition is equivalent to using the addition operator of type `System.Decimal`.

- Enumeration addition. Every enumeration type implicitly provides the following predefined operators, where `E` is the enum type, and `U` is the underlying type of `E`:

```
E operator +(E x, U y);
E operator +(U x, E y);
```

The operators are evaluated exactly as `(E)((U)x + (U)y)`.

- String concatenation:

```
string operator +(string x, string y);
string operator +(string x, object y);
string operator +(object x, string y);
```

The binary `+` operator performs string concatenation when one or both operands are of type `string`. If an operand of string concatenation is `null`, an empty string is substituted. Otherwise, any non-string argument is converted to its string representation by invoking the virtual `ToString` method inherited from type `object`. If `ToString` returns `null`, an empty string is substituted.

```
using System;
class Test
{
   static void Main() {
      string s = null;
      Console.WriteLine("s = >" + s + "<"); // displays s = ><
      int i = 1;
      Console.WriteLine("i = " + i);        // displays i = 1
      float f = 1.2300E+15F;
      Console.WriteLine("f = " + f);        // displays f = 1.23+E15
      decimal d = 2.900m;
      Console.WriteLine("d = " + d);        // displays d = 2.900
   }
}
```

The result of the string concatenation operator is a string that consists of the characters of the left operand followed by the characters of the right operand. The string concatenation operator never returns a `null` value. A `System.OutOfMemoryException` may be thrown if there is not enough memory available to allocate the resulting string.

- Delegate combination. Every delegate type implicitly provides the following predefined operator, where `D` is the delegate type:

```
D operator +(D x, D y);
```

The binary `+` operator performs delegate combination when both operands are of some delegate type `D`. (If the operands have different delegate types, a compile-time error occurs.) If the first operand is `null`, the result of the operation is the value of the second operand (even if that is also `null`). Otherwise, if the second operand is `null`, then the result of the operation is the value of the first operand. Otherwise, the result of the operation is a new delegate instance that, when invoked, invokes the first operand and then invokes the second operand. For examples of delegate combination, see Section 7.7.5 and Section 15.3. Since `System.Delegate` is not a delegate type, `operator +` is not defined for it.

7.7.5 Subtraction operator

For an operation of the form `x - y`, binary operator overload resolution (Section 7.2.4) is applied to select a specific operator implementation. The operands are converted to the parameter types of the selected operator, and the type of the result is the return type of the operator.

The predefined subtraction operators are listed below. The operators all subtract `y` from `x`.

- Integer subtraction:

```
int operator -(int x, int y);
uint operator -(uint x, uint y);
long operator -(long x, long y);
ulong operator -(ulong x, ulong y);
```

In a `checked` context, if the difference is outside the range of the result type, a `System.OverflowException` is thrown. In an `unchecked` context, overflows are not reported and any significant high-order bits outside the range of the result type are discarded.

- Floating-point subtraction:

```
float operator -(float x, float y);
double operator -(double x, double y);
```

The difference is computed according to the rules of IEEE 754 arithmetic. The following table lists the results of all possible combinations of nonzero finite values, zeros, infinities, and NaN's. In the table, `x` and `y` are nonzero finite values, and `z` is the result of `x - y`. If `x` and `y` are equal, `z` is positive zero. If `x - y` is too large to represent in the destination type, `z` is an infinity with the same sign as `x - y`.

	y	**+0**	**–0**	**+∞**	**–∞**	**NaN**
x	z	x	x	-8	+8	NaN
+0	-y	+0	+0	-8	+8	NaN
–0	-y	-0	+0	-8	+8	NaN
+∞	+8	+8	+8	NaN	+8	NaN
–∞	-8	-8	-8	-8	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN

- Decimal subtraction:

```
decimal operator -(decimal x, decimal y);
```

If the resulting value is too large to represent in the `decimal` format, a `System.OverflowException` is thrown. The scale of the result, before any rounding, is the larger of the scales of the two operands.

Decimal subtraction is equivalent to using the subtraction operator of type `System.Decimal`.

- Enumeration subtraction. Every enumeration type implicitly provides the following predefined operator, where `E` is the enum type, and `U` is the underlying type of `E`:

```
U operator -(E x, E y);
```

This operator is evaluated exactly as `(U)((U)x - (U)y)`. In other words, the operator computes the difference between the ordinal values of `x` and `y`, and the type of the result is the underlying type of the enumeration.

```
E operator -(E x, U y);
```

This operator is evaluated exactly as `(E)((U)x - y)`. In other words, the operator subtracts a value from the underlying type of the enumeration, yielding a value of the enumeration.

- Delegate removal. Every delegate type implicitly provides the following predefined operator, where `D` is the delegate type:

```
D operator -(D x, D y);
```

The binary `-` operator performs delegate removal when both operands are of some delegate type `D`. If the operands have different delegate types, a compile-time error occurs. If the first operand is `null`, the result of the operation is `null`. Otherwise, if the second operand is `null`, then the result of the operation is the value of the first operand. Otherwise, both operands represent invocation lists (Section 15.1) having one or more entries, and the result is a new invocation list consisting of the first operand's list with the second operand's entries removed from it, provided the second operand's list is a proper contiguous sublist of the first's. (To determine sublist equality, corresponding entries are compared as for the delegate equality operator; see Section 7.9.8). Otherwise, the result is the value of the left operand. Neither of the operands' lists is changed in the process. If the second operand's list matches multiple sublists of contiguous entries in the first operand's list, the right-most matching sublist of contiguous entries is removed. If removal results in an empty list, the result is `null`. For example:

```
delegate void D(int x);
class C
{
   public static void M1(int i) { /* ... */ }
   public static void M2(int i) { /* ... */ }
}
class Test
{
   static void Main() {
      D cd1 = new D(C.M1);
      D cd2 = new D(C.M2);
      D cd3 = cd1 + cd2 + cd2 + cd1;   // M1 + M2 + M2 + M1
      cd3 -= cd1;                        // => M1 + M2 + M2
      cd3 = cd1 + cd2 + cd2 + cd1;      // M1 + M2 + M2 + M1
      cd3 -= cd1 + cd2;                  // => M2 + M1
      cd3 = cd1 + cd2 + cd2 + cd1;      // M1 + M2 + M2 + M1
      cd3 -= cd2 + cd2;                  // => M1 + M1
      cd3 = cd1 + cd2 + cd2 + cd1;      // M1 + M2 + M2 + M1
      cd3 -= cd2 + cd1;                  // => M1 + M2
      cd3 = cd1 + cd2 + cd2 + cd1;      // M1 + M2 + M2 + M1
      cd3 -= cd1 + cd1;                  // => M1 + M2 + M2 + M1
   }
}
```

7.8 Shift operators

The << and >> operators are used to perform bit shifting operations.

shift-expression:
  additive-expression
  shift-expression << *additive-expression*
  shift-expression >> *additive-expression*

For an operation of the form `x << count` or `x >> count`, binary operator overload resolution (Section 7.2.4) is applied to select a specific operator implementation. The operands are converted to the parameter types of the selected operator, and the type of the result is the return type of the operator.

When declaring an overloaded shift operator, the type of the first operand must always be the class or struct containing the operator declaration, and the type of the second operand must always be `int`.

The predefined shift operators are listed below.

- Shift left:

```
int operator <<(int x, int count);
uint operator <<(uint x, int count);
long operator <<(long x, int count);
ulong operator <<(ulong x, int count);
```

The << operator shifts `x` left by a number of bits computed as described below.

The high-order bits outside the range of the result type of `x` are discarded, the remaining bits are shifted left, and the low-order empty bit positions are set to zero.

- Shift right:

```
int operator >>(int x, int count);
uint operator >>(uint x, int count);
long operator >>(long x, int count);
ulong operator >>(ulong x, int count);
```

The >> operator shifts `x` right by a number of bits computed as described below.

When `x` is of type `int` or `long`, the low-order bits of `x` are discarded, the remaining bits are shifted right, and the high-order empty bit positions are set to zero if `x` is non-negative and set to one if `x` is negative.

When `x` is of type `uint` or `ulong`, the low-order bits of `x` are discarded, the remaining bits are shifted right, and the high-order empty bit positions are set to zero.

For the predefined operators, the number of bits to shift is computed as follows:

- When the type of `x` is `int` or `uint`, the shift count is given by the low-order five bits of `count`. In other words, the shift count is computed from `count & 0x1F`.
- When the type of `x` is `long` or `ulong`, the shift count is given by the low-order six bits of `count`. In other words, the shift count is computed from `count & 0x3F`.

If the resulting shift count is zero, the shift operators simply return the value of `x`.

Shift operations never cause overflows and produce the same results in `checked` and `unchecked` contexts.

When the left operand of the >> operator is of a signed integral type, the operator performs an arithmetic shift right wherein the value of the most significant bit (the sign bit) of the operand is propagated to the high-order empty bit positions. When the left operand of the >> operator is of an unsigned integral type, the operator performs a logical shift right wherein high-order empty bit positions are always set to zero. To perform the opposite operation of that inferred from the operand type, explicit casts can be used. For example, if `x` is a variable of type `int`, the operation `unchecked((int)((uint)x >> y))` performs a logical shift right of `x`.

7.9 Relational and type-testing operators

The `==`, `!=`, `<`, `>`, `<=`, `>=`, `is` and `as` operators are called the relational and type-testing operators.

relational-expression:
  shift-expression
  relational-expression `<` *shift-expression*
  relational-expression `>` *shift-expression*
  relational-expression `<=` *shift-expression*
  relational-expression `>=` *shift-expression*
  relational-expression `is` *type*
  relational-expression `as` *type*
equality-expression:
  relational-expression
  equality-expression `==` *relational-expression*
  equality-expression `!=` *relational-expression*

The `is` operator is described in Section 7.9.9 and the `as` operator is described in Section 7.9.10.

The `==`, `!=`, `<`, `>`, `<=` and `>=` operators are comparison operators. For an operation of the form `x` *op* `y`, where *op* is a comparison operator, overload resolution (Section 7.2.4) is applied to select a specific operator implementation. The operands are converted to the parameter types of the selected operator, and the type of the result is the return type of the operator.

The predefined comparison operators are described in the following sections. All predefined comparison operators return a result of type `bool`, as described in the following table.

Operation	Result
`x == y`	`true` if `x` is equal to `y`, `false` otherwise
`x != y`	`true` if `x` is not equal to `y`, `false` otherwise
`x < y`	`true` if `x` is less than `y`, `false` otherwise
`x > y`	`true` if `x` is greater than `y`, `false` otherwise
`x <= y`	`true` if `x` is less than or equal to `y`, `false` otherwise
`x >= y`	`true` if `x` is greater than or equal to `y`, `false` otherwise

7.9.1 Integer comparison operators

The predefined integer comparison operators are:

```
bool operator ==(int x, int y);
bool operator ==(uint x, uint y);
bool operator ==(long x, long y);
bool operator ==(ulong x, ulong y);
bool operator !=(int x, int y);
bool operator !=(uint x, uint y);
bool operator !=(long x, long y);
bool operator !=(ulong x, ulong y);
bool operator <(int x, int y);
bool operator <(uint x, uint y);
bool operator <(long x, long y);
bool operator <(ulong x, ulong y);
bool operator >(int x, int y);
bool operator >(uint x, uint y);
bool operator >(long x, long y);
bool operator >(ulong x, ulong y);
bool operator <=(int x, int y);
bool operator <=(uint x, uint y);
bool operator <=(long x, long y);
bool operator <=(ulong x, ulong y);
bool operator >=(int x, int y);
bool operator >=(uint x, uint y);
bool operator >=(long x, long y);
bool operator >=(ulong x, ulong y);
```

Each of these operators compares the numeric values of the two integer operands and returns a `bool` value that indicates whether the particular relation is `true` or `false`.

7.9.2 Floating-point comparison operators

The predefined floating-point comparison operators are:

```
bool operator ==(float x, float y);
bool operator ==(double x, double y);
bool operator !=(float x, float y);
bool operator !=(double x, double y);
bool operator <(float x, float y);
bool operator <(double x, double y);
bool operator >(float x, float y);
bool operator >(double x, double y);
bool operator <=(float x, float y);
bool operator <=(double x, double y);
bool operator >=(float x, float y);
bool operator >=(double x, double y);
```

The operators compare the operands according to the rules of the IEEE 754 standard:

- If either operand is NaN, the result is `false` for all operators except `!=`, for which the result is `true`. For any two operands, `x != y` always produces the same result as `!(x == y)`. However, when one or both operands are NaN, the `<`, `>`, `<=`, and `>=` operators do not produce the same results as the logical negation of the opposite operator. For example, if either of `x` and `y` is NaN, then `x < y` is `false`, but `!(x >= y)` is `true`.
- When neither operand is NaN, the operators compare the values of the two floating-point operands with respect to the ordering

  ```
  -8 < -max < ... < -min < -0.0 == +0.0 < +min < ... < +max < +8
  ```

where `min` and `max` are the smallest and largest positive finite values that can be represented in the given floating-point format. Notable effects of this ordering are:

- Negative and positive zeros are considered equal.
- A negative infinity is considered less than all other values, but equal to another negative infinity.
- A positive infinity is considered greater than all other values, but equal to another positive infinity.

7.9.3 Decimal comparison operators

The predefined decimal comparison operators are:

```
bool operator ==(decimal x, decimal y);
bool operator !=(decimal x, decimal y);
bool operator <(decimal x, decimal y);
bool operator >(decimal x, decimal y);
bool operator <=(decimal x, decimal y);
bool operator >=(decimal x, decimal y);
```

Each of these operators compares the numeric values of the two decimal operands and returns a `bool` value that indicates whether the particular relation is `true` or `false`. Each decimal comparison is equivalent to using the corresponding relational or equality operator of type `System.Decimal`.

7.9.4 Boolean equality operators

The predefined Boolean equality operators are:

```
bool operator ==(bool x, bool y);
bool operator !=(bool x, bool y);
```

The result of `==` is `true` if both `x` and `y` are `true` or if both `x` and `y` are `false`. Otherwise, the result is `false`.

The result of `!=` is `false` if both `x` and `y` are `true` or if both `x` and `y` are `false`. Otherwise, the result is `true`. When the operands are of type `bool`, the `!=` operator produces the same result as the `^` operator.

7.9.5 Enumeration comparison operators

Every enumeration type implicitly provides the following predefined comparison operators:

```
bool operator ==(E x, E y);
bool operator !=(E x, E y);
bool operator <(E x, E y);
bool operator >(E x, E y);
bool operator <=(E x, E y);
bool operator >=(E x, E y);
```

The result of evaluating `x` *op* `y`, where `x` and `y` are expressions of an enumeration type `E` with an underlying type `U`, and *op* is one of the comparison operators, is exactly the same as evaluating `((U)x)` *op* `((U)y)`. In other words, the enumeration type comparison operators simply compare the underlying integral values of the two operands.

7.9.6 Reference type equality operators

The predefined reference type equality operators are:

```
bool operator ==(object x, object y);
bool operator !=(object x, object y);
```

The operators return the result of comparing the two references for equality or non-equality.

Since the predefined reference type equality operators accept operands of type `object`, they apply to all types that do not declare applicable `operator ==` and `operator !=` members. Conversely, any applicable user-defined equality operators effectively hide the predefined reference type equality operators.

The predefined reference type equality operators require the operands to be *reference-type* values or the value `null`; furthermore, they require that a standard implicit conversion (Section 6.3.1) exists from the type of either operand to the type of the other operand. Unless both of these conditions are true, a compile-time error occurs. Notable implications of these rules are:

- It is a compile-time error to use the predefined reference type equality operators to compare two references that are known to be different at compile-time. For example, if the compile-time types of the operands are two class types `A` and `B`, and if neither `A` nor `B` derives from the other, then it would be impossible for the two operands to reference the same object. Thus, the operation is considered a compile-time error.
- The predefined reference type equality operators do not permit value type operands to be compared. Therefore, unless a struct type declares its own equality operators, it is not possible to compare values of that struct type.
- The predefined reference type equality operators never cause boxing operations to occur for their operands. It would be meaningless to perform such boxing operations, since references to the newly allocated boxed instances would necessarily differ from all other references.

For an operation of the form `x == y` or `x != y`, if any applicable `operator ==` or `operator !=` exists, the operator overload resolution (Section 7.2.4) rules will select that operator instead of the predefined reference type equality operator. However, it is always possible to select the predefined reference type equality operator by explicitly casting one or both of the operands to type `object`. The example

```
using System;
class Test
{
   static void Main() {
      string s = "Test";
      string t = string.Copy(s);
      Console.WriteLine(s == t);
      Console.WriteLine((object)s == t);
      Console.WriteLine(s == (object)t);
      Console.WriteLine((object)s == (object)t);
   }
}
```

produces the output

```
True
False
False
False
```

The `s` and `t` variables refer to two distinct `string` instances containing the same characters. The first comparison outputs `True` because the predefined string equality operator (Section 7.9.7) is selected when both operands are of type `string`. The remaining comparisons all output `False` because the predefined reference type equality operator is selected when one or both of the operands are of type `object`.

Note that the above technique is not meaningful for value types. The example

```
class Test
{
```

```
    static void Main() {
       int i = 123;
       int j = 123;
       System.Console.WriteLine((object)i == (object)j);
    }
 }
```

outputs `False` because the casts create references to two separate instances of boxed `int` values.

7.9.7 String equality operators

The predefined string equality operators are:

```
bool operator ==(string x, string y);
bool operator !=(string x, string y);
```

Two `string` values are considered equal when one of the following is true:

- Both values are `null`.
- Both values are non-null references to string instances that have identical lengths and identical characters in each character position.

The string equality operators compare string values rather than string references. When two separate string instances contain the exact same sequence of characters, the values of the strings are equal, but the references are different. As described in Section 7.9.6, the reference type equality operators can be used to compare string references instead of string values.

7.9.8 Delegate equality operators

Every delegate type implicitly provides the following predefined comparison operators:

```
bool operator ==(System.Delegate x, System.Delegate y);
bool operator !=(System.Delegate x, System.Delegate y);
```

Two delegate instances are considered equal as follows:

- If either of the delegate instances is `null`, they are equal if and only if both are `null`.
- If either of the delegate instances has an invocation list (Section 15.1) containing one entry, they are equal if and only if the other also has an invocation list containing one entry, and either:
  - both refer to the same static method, or
  - both refer to the same non-static method on the same target object.
- If either of the delegate instances has an invocation list containing two or more entries, those instances are equal if and only if their invocation lists are the same length, and each entry in one's invocation list is equal to the corresponding entry, in order, in the other's invocation list.

Note that delegates of different types can be considered equal by the above definition, as long as they have the same return type and parameter types.

7.9.9 The is operator

The `is` operator is used to dynamically check if the run-time type of an object is compatible with a given type. The result of the operation `e is T`, where `e` is an expression and `T` is a type, is a Boolean value indicating whether `e` can successfully be converted to type `T` by a reference conversion, a boxing conversion, or an unboxing conversion. The operation is evaluated as follows:

- If the compile-time type of `e` is the same as `T`, or if an implicit reference conversion (Section 6.1.4) or boxing conversion (Section 6.1.5) exists from the compile-time type of `e` to `T`:
  - If `e` is of a reference type, the result of the operation is equivalent to evaluating `e != null`.
  - If `e` is of a value type, the result of the operation is `true`.
- Otherwise, if an explicit reference conversion (Section 6.2.3) or unboxing conversion (Section 6.2.4) exists from the compile-time type of `e` to `T`, a dynamic type check is performed:
  - If the value of `e` is `null`, the result is `false`.
  - Otherwise, let `R` be the run-time type of the instance referenced by `e`. If `R` and `T` are the same type, if `R` is a reference type and an implicit reference conversion from `R` to `T` exists, or if `R` is a value type and `T` is an interface type that is implemented by `R`, the result is `true`.
  - Otherwise, the result is `false`.
- Otherwise, no reference or boxing conversion of `e` to type `T` is possible, and the result of the operation is false.

Note The `is` operator only considers reference conversions, boxing conversions, and unboxing conversions. Other conversions, such as user defined conversions, are not considered by the `is` operator.

7.9.10 The as operator

The `as` operator is used to explicitly convert a value to a given reference type using a reference conversion or a boxing conversion. Unlike a cast expression (Section 7.6.6), the `as` operator never throws an exception. Instead, if the indicated conversion is not possible, the resulting value is `null`.

In an operation of the form `e as T`, `e` must be an expression and `T` must be a reference type. The type of the result is `T`, and the result is always classified as a value. The operation is evaluated as follows:

- If the compile-time type of `e` is the same as `T`, the result is simply the value of `e`.
- Otherwise, if an implicit reference conversion (Section 6.1.4) or boxing conversion (Section 6.1.5) exists from the compile-time type of e to `T`, this conversion is performed and becomes the result of the operation.
- Otherwise, if an explicit reference conversion (Section 6.2.3) exists from the compile-time type of e to T, a dynamic type check is performed:
    - If the value of e is `null`, the result is the value `null` with the compile-time type `T`.
    - Otherwise, let `R` be the run-time type of the instance referenced by `e`. If `R` and `T` are the same type, if `R` is a reference type and an implicit reference conversion from `R` to `T` exists, or if `R` is a value type and `T` is an interface type that is implemented by `R`, the result is the reference given by `e` with the compile-time type `T`.
    - Otherwise, the result is the value `null` with the compile-time type `T`.
- Otherwise, the indicated conversion is never possible, and a compile-time error occurs.

Note that the `as` operator only performs reference conversions and boxing conversions. Other conversions, such as user defined conversions, are not possible with the `as` operator and should instead be performed using cast expressions.

7.10 Logical operators

The `&`, `^`, and `|` operators are called the logical operators.

and-expression:
  equality-expression
  and-expression & *equality-expression*
exclusive-or-expression:
  and-expression
  exclusive-or-expression ^ *and-expression*
inclusive-or-expression:
  exclusive-or-expression
  inclusive-or-expression | *exclusive-or-expression*

For an operation of the form `x` *op* `y`, where *op* is one of the logical operators, overload resolution (Section 7.2.4) is applied to select a specific operator implementation. The operands are converted to the parameter types of the selected operator, and the type of the result is the return type of the operator.

The predefined logical operators are described in the following sections.

7.10.1 Integer logical operators

The predefined integer logical operators are:

```
int operator &(int x, int y);
uint operator &(uint x, uint y);
long operator &(long x, long y);
ulong operator &(ulong x, ulong y);
int operator |(int x, int y);
uint operator |(uint x, uint y);
long operator |(long x, long y);
ulong operator |(ulong x, ulong y);
int operator ^(int x, int y);
uint operator ^(uint x, uint y);
long operator ^(long x, long y);
ulong operator ^(ulong x, ulong y);
```

The `&` operator computes the bitwise logical `AND` of the two operands, the `|` operator computes the bitwise logical `OR` of the two operands, and the `^` operator computes the bitwise logical exclusive `OR` of the two operands. No overflows are possible from these operations.

7.10.2 Enumeration logical operators

Every enumeration type `E` implicitly provides the following predefined logical operators:

```
E operator &(E x, E y);
E operator |(E x, E y);
E operator ^(E x, E y);
```

The result of evaluating `x` *op* `y`, where `x` and `y` are expressions of an enumeration type `E` with an underlying type `U`, and *op* is one of the logical operators, is exactly the same as evaluating (E) `((U)x` *op* `(U)y)`. In other words, the enumeration type logical operators simply perform the logical operation on the underlying type of the two operands.

7.10.3 Boolean logical operators

The predefined Boolean logical operators are:

```
bool operator &(bool x, bool y);
bool operator |(bool x, bool y);
bool operator ^(bool x, bool y);
```

The result of `x & y` is `true` if both `x` and `y` are `true`. Otherwise, the result is `false`.

The result of `x | y` is `true` if either `x` or `y` is `true`. Otherwise, the result is `false`.

The result of `x ^ y` is `true` if `x` is `true` and `y` is `false`, or `x` is `false` and `y` is `true`. Otherwise, the result is `false`. When the operands are of type `bool`, the `^` operator computes the same result as the `!=` operator.

7.11 Conditional logical operators

The `&&` and `||` operators are called the conditional logical operators. They are also called the "short-circuiting" logical operators.

conditional-and-expression:
  inclusive-or-expression
  conditional-and-expression && *inclusive-or-expression*
conditional-or-expression:
  conditional-and-expression
  conditional-or-expression || *conditional-and-expression*

The `&&` and `||` operators are conditional versions of the `&` and `|` operators:

- The operation `x && y` corresponds to the operation `x & y`, except that `y` is evaluated only if `x` is `true`.
- The operation `x || y` corresponds to the operation `x | y`, except that `y` is evaluated only if `x` is `false`.

An operation of the form `x && y` or `x || y` is processed by applying overload resolution (Section 7.2.4) as if the operation was written `x & y` or `x | y`. Then,

- If overload resolution fails to find a single best operator, or if overload resolution selects one of the predefined integer logical operators, a compile-time error occurs.
- Otherwise, if the selected operator is one of the predefined Boolean logical operators (Section 7.10.2), the operation is processed as described in Section 7.11.1.
- Otherwise, the selected operator is a user-defined operator, and the operation is processed as described in Section 7.11.2.

It is not possible to directly overload the conditional logical operators. However, because the conditional logical operators are evaluated in terms of the regular logical operators, overloads of the regular logical operators are, with certain restrictions, also considered overloads of the conditional logical operators. This is described further in Section 7.11.2.

7.11.1 Boolean conditional logical operators

When the operands of `&&` or `||` are of type `bool`, or when the operands are of types that do not define an applicable `operator &` or `operator |`, but do define implicit conversions to `bool`, the operation is processed as follows:

- The operation `x && y` is evaluated as `x ? y : false`. In other words, `x` is first evaluated and converted to type `bool`. Then, if `x` is `true`, `y` is evaluated and converted to type `bool`, and this becomes the result of the operation. Otherwise, the result of the operation is `false`.
- The operation `x || y` is evaluated as `x ? true : y`. In other words, `x` is first evaluated and converted to type `bool`. Then, if `x` is `true`, the result of the operation is `true`. Otherwise, `y` is evaluated and converted to type `bool`, and this becomes the result of the operation.

7.11.2 User-defined conditional logical operators

When the operands of `&&` or `||` are of types that declare an applicable user-defined `operator &` or `operator |`, both of the following must be true, where `T` is the type in which the selected operator is declared:

- The return type and the type of each parameter of the selected operator must be `T`. In other words, the operator must compute the logical `AND` or the logical `OR` of two operands of type `T`, and must return a result of type `T`.
- `T` must contain declarations of `operator true` and `operator false`.

A compile-time error occurs if either of these requirements is not satisfied. Otherwise, the `&&` or `||` operation is evaluated by combining the user-defined `operator true` or `operator false` with the selected user-defined operator:

- The operation `x && y` is evaluated as `T.false(x) ? x : T.&(x, y)`, where `T.false(x)` is an invocation of the `operator false` declared in `T`, and `T.&(x, y)` is an invocation of the selected `operator &`. In other words, `x` is first evaluated and `operator false` is invoked on the result to determine if `x` is definitely false. Then, if `x` is definitely false, the result of the operation is the value previously computed for `x`. Otherwise, `y` is evaluated, and the selected `operator &` is invoked on the value previously computed for `x` and the value computed for `y` to produce the result of the operation.
- The operation `x || y` is evaluated as `T.true(x) ? x : T.|(x, y)`, where `T.true(x)` is an invocation of the `operator true` declared in `T`, and `T.|(x, y)` is an invocation of the selected `operator |`. In other words, `x` is first evaluated and `operator true` is invoked on the result to determine if `x` is definitely true. Then, if `x` is definitely true, the result of the operation is the value previously computed for `x`. Otherwise, `y` is evaluated, and the selected `operator |` is invoked on the value previously computed for `x` and the value computed for `y` to produce the result of the operation.

In either of these operations, the expression given by `x` is only evaluated once, and the expression given by `y` is either not evaluated or evaluated exactly once.

For an example of a type that implements `operator true` and `operator false`, see Section 11.4.2.

7.12 Conditional operator

The `?:` operator is called the conditional operator. It is at times also called the ternary operator.

conditional-expression:
  conditional-or-expression
  conditional-or-expression ? *expression* : *expression*

A conditional expression of the form `b ? x : y` first evaluates the condition `b`. Then, if `b` is `true`, `x` is evaluated and becomes the result of the operation. Otherwise, `y` is evaluated and becomes the result of the operation. A conditional expression never evaluates both `x` and `y`.

The conditional operator is right-associative, meaning that operations are grouped from right to left. For example, an expression of the form `a ? b : c ? d : e` is evaluated as `a ? b : (c ? d : e)`.

The first operand of the `?:` operator must be an expression of a type that can be implicitly converted to `bool`, or an expression of a type that implements `operator true`. If neither requirement is satisfied, a compile-time error occurs.

The second and third operands of the `?:` operator control the type of the conditional expression. Let `X` and `Y` be the types of the second and third operands. Then,

- If `X` and `Y` are the same type, then this is the type of the conditional expression.
- Otherwise, if an implicit conversion (Section 6.1) exists from `X` to `Y`, but not from `Y` to `X`, then `Y` is the type of the conditional expression.
- Otherwise, if an implicit conversion (Section 6.1) exists from `Y` to `X`, but not from `X` to `Y`, then `X` is the type of the conditional expression.
- Otherwise, no expression type can be determined, and a compile-time error occurs.

The run-time processing of a conditional expression of the form `b ? x : y` consists of the following steps:

- First, `b` is evaluated, and the `bool` value of `b` is determined:
  - If an implicit conversion from the type of `b` to `bool` exists, then this implicit conversion is performed to produce a `bool` value.
  - Otherwise, the `operator true` defined by the type of `b` is invoked to produce a `bool` value.
- If the `bool` value produced by the step above is `true`, then `x` is evaluated and converted to the type of the conditional expression, and this becomes the result of the conditional expression.
- Otherwise, `y` is evaluated and converted to the type of the conditional expression, and this becomes the result of the conditional expression.

7.13 Assignment operators

The assignment operators assign a new value to a variable, a property, an event, or an indexer element.

assignment:
  unary-expression *assignment-operator* *expression*
assignment-operator: one of
  = += -= *= /= %= &= |= ^= <<= >>=

The left operand of an assignment must be an expression classified as a variable, a property access, an indexer access, or an event access.

The `=` operator is called the simple assignment operator. It assigns the value of the right operand to the variable, property, or indexer element given by the left operand. The left operand of the simple assignment operator may not be an event access (except as described in Section 10.7.1. The simple assignment operator is described in Section 7.13.1.

The assignment operators other than the `=` operator are called the compound assignment operators. These operators perform the indicated operation on the two operands, and then assign the resulting value to the variable, property, or indexer element given by the left operand. The compound assignment operators are described in Section 7.13.2.

The `+=` and `-=` operators with an event access expression as the left operand are called the event assignment operators. No other assignment operator is valid with an event access as the left operand. The event assignment operators are described in Section 7.13.3.

The assignment operators are right-associative, meaning that operations are grouped from right to left. For example, an expression of the form `a = b = c` is evaluated as `a = (b = c)`.

7.13.1 Simple assignment

The `=` operator is called the simple assignment operator. In a simple assignment, the right operand must be an expression of a type that is implicitly convertible to the type of the left operand. The operation assigns the value of the right operand to the variable, property, or indexer element given by the left operand.

The result of a simple assignment expression is the value assigned to the left operand. The result has the same type as the left operand and is always classified as a value.

If the left operand is a property or indexer access, the property or indexer must have a `set` accessor. If this is not the case, a compile-time error occurs.

The run-time processing of a simple assignment of the form `x = y` consists of the following steps:

- If x is classified as a variable:
  - `x` is evaluated to produce the variable.
  - `y` is evaluated and, if required, converted to the type of `x` through an implicit conversion (Section 6.1).
  - If the variable given by `x` is an array element of a *reference-type*, a run-time check is performed to ensure that the value computed for `y` is compatible with the array instance of which `x` is an element. The check succeeds if `y` is `null`, or if an implicit reference conversion (Section 6.1.4) exists from the actual type of the instance referenced by `y` to the actual element type of the array instance containing `x`. Otherwise, a `System.ArrayTypeMismatchException` is thrown.
  - The value resulting from the evaluation and conversion of `y` is stored into the location given by the evaluation of `x`.
- If x is classified as a property or indexer access:
  - The instance expression (if `x` is not `static`) and the argument list (if `x` is an indexer access) associated with `x` are evaluated, and the results are used in the subsequent `set` accessor invocation.
  - `y` is evaluated and, if required, converted to the type of `x` through an implicit conversion (Section 6.1).
  - The `set` accessor of `x` is invoked with the value computed for `y` as its `value` argument.

The array co-variance rules (Section 12.5) permit a value of an array type `A[]` to be a reference to an instance of an array type `B[]`, provided an implicit reference conversion exists from `B` to `A`. Because of these rules, assignment to an array element of a *reference-type* requires a run-time check to ensure that the value being assigned is compatible with the array instance. In the example

```
string[] sa = new string[10];
object[] oa = sa;
oa[0] = null;               // Ok
oa[1] = "Hello";            // Ok
oa[2] = new ArrayList();   // ArrayTypeMismatchException
```

the last assignment causes a `System.ArrayTypeMismatchException` to be thrown because an instance of `ArrayList` cannot be stored in an element of a `string[]`.

When a property or indexer declared in a struct-type is the target of an assignment, the instance expression associated with the property or indexer access must be classified as a variable. If the instance expression is classified as a value, a compile-time error occurs. Because of Section 7.5.4, the same rule also applies to fields.

Given the declarations:

```
struct Point
{
   int x, y;
   public Point(int x, int y) {
      this.x = x;
      this.y = y;
   }
   public int X {
      get { return x; }
      set { x = value; }
   }
   public int Y {
      get { return y; }
```

```
        set { y = value; }
    }
}
struct Rectangle
{
    Point a, b;
    public Rectangle(Point a, Point b) {
        this.a = a;
        this.b = b;
    }
    public Point A {
        get { return a; }
        set { a = value; }
    }
    public Point B {
        get { return b; }
        set { b = value; }
    }
}
```

in the example

```
Point p = new Point();
p.X = 100;
p.Y = 100;
Rectangle r = new Rectangle();
r.A = new Point(10, 10);
r.B = p;
```

the assignments to `p.X`, `p.Y`, `r.A`, and `r.B` are permitted because `p` and `r` are variables. However, in the example

```
Rectangle r = new Rectangle();
r.A.X = 10;
r.A.Y = 10;
r.B.X = 100;
r.B.Y = 100;
```

the assignments are all invalid, since `r.A` and `r.B` are not variables.

7.13.2 Compound assignment

An operation of the form `x` *op*`=` `y` is processed by applying binary operator overload resolution (Section 7.2.4) as if the operation was written `x` *op* `y`. Then,

- If the return type of the selected operator is implicitly convertible to the type of `x`, the operation is evaluated as `x = x` *op* `y`, except that `x` is evaluated only once.
- Otherwise, if the selected operator is a predefined operator, if the return type of the selected operator is explicitly convertible to the type of `x`, and if `y` is implicitly convertible to the type of `x`, then the operation is evaluated as `x = (T) (x` *op* `y)`, where `T` is the type of `x`, except that `x` is evaluated only once.
- Otherwise, the compound assignment is invalid, and a compile-time error occurs.

The term "evaluated only once" means that in the evaluation of `x` *op* `y`, the results of any constituent expressions of `x` are temporarily saved and then reused when performing the assignment to `x`. For example, in the assignment `A()[B()] += C()`, where `A` is a method returning `int[]`, and `B` and `C` are methods returning `int`, the methods are invoked only once, in the order `A`, `B`, `C`.

When the left operand of a compound assignment is a property access or indexer access, the property or indexer must have both a `get` accessor and a `set` accessor. If this is not the case, a compile-time error occurs.

The second rule above permits `x` *op*`=` `y` to be evaluated as `x = (T) (x` *op* `y)` in certain contexts. The rule exists such that the predefined operators can be used as compound operators when the left operand is of type `sbyte`, `byte`, `short`, `ushort`, or `char`. Even when both arguments are of one of those types, the predefined operators produce a result of type `int`, as described in Section 7.2.6.2. Thus, without a cast it would not be possible to assign the result to the left operand.

The intuitive effect of the rule for predefined operators is simply that `x` *op*`=` `y` is permitted if both of `x` *op* `y` and `x = y` are permitted. In the example

```
byte b = 0;
char ch = '\0';
int i = 0;
b += 1;            // Ok
b += 1000;         // Error, b = 1000 not permitted
b += i;            // Error, b = i not permitted
b += (byte)i;      // Ok
ch += 1;            // Error, ch = 1 not permitted
ch += (char)1;      // Ok
```

the intuitive reason for each error is that a corresponding simple assignment would also have been an error.

7.13.3 Event assignment

If the left operand of a `+=` or `-=` operator is classified as an event access, then the expression is evaluated as follows:

- The instance expression, if any, of the event access is evaluated.
- The right operand of the `+=` or `-=` operator is evaluated, and, if required, converted to the type of the left operand through an implicit conversion (Section 6.1).
- An event accessor of the event is invoked, with the argument list consisting of the right operand, after evaluation and, if necessary, conversion. If the operator was `+=`, the `add` accessor is invoked. If the operator was `-=`, the `remove` accessor is invoked.

An event assignment expression does not yield a value. Thus, an event assignment expression is valid only in the context of a *statement-expression* (Section 8.6).

7.14 Expression

An *expression* is either a conditional-expression or an assignment.

expression:
  conditional-expression
  assignment

7.15 Constant expressions

A constant-expression is an expression that can be fully evaluated at compile-time.

constant-expression:
  expression

The type of a constant expression can be one of the following: `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, `decimal`, `bool`, `string`, any enumeration type, or the null type. The following constructs are permitted in constant expressions:

- Literals (including the `null` literal).
- References to `const` members of class and struct types.
- References to members of enumeration types.
- Parenthesized sub-expressions, which are themselves constant expressions.
- Cast expressions, provided the target type is one of the types listed above.
- The predefined `+`, `-`, `!`, and `~` unary operators.
- The predefined `+`, `-`, `*`, `/`, `%`, `<<`, `>>`, `&`, `|`, `^`, `&&`, `||`, `==`, `!=`, `<`, `>`, `<=`, and `>=` binary operators, provided each operand is of a type listed above.
- The `?:` conditional operator.

Whenever an expression is of one of the types listed above and contains only the constructs listed above, the expression is evaluated at compile-time. This is true even if the expression is a sub-expression of a larger expression that contains non-constant constructs.

The compile-time evaluation of constant expressions uses the same rules as run-time evaluation of non-constant expressions, except that where run-time evaluation would have thrown an exception, compile-time evaluation causes a compile-time error to occur.

Unless a constant expression is explicitly placed in an `unchecked` context, overflows that occur in integral-type arithmetic operations and conversions during the compile-time evaluation of the expression always cause compile-time errors (Section 7.5.12).

Constant expressions occur in the contexts listed below. In these contexts, a compile-time error occurs if an expression cannot be fully evaluated at compile-time.

- Constant declarations (Section 10.3).
- Enumeration member declarations (Section 14.3).
- `case` labels of a `switch` statement (Section 8.7.2).
- `goto case` statements (Section 8.9.3).
- Dimension lengths in an array creation expression (Section 7.5.10.2) that includes an initializer.
- Attributes (Section 17).

An implicit constant expression conversion (Section 6.1.6) permits a constant expression of type `int` to be converted to `sbyte`, `byte`, `short`, `ushort`, `uint`, or `ulong`, provided the value of the constant expression is within the range of the destination type.

7.16 Boolean expressions

A boolean-expression is an expression that yields a result of type `bool`.

boolean-expression:
  expression

The controlling conditional expression of an *if-statement* (Section 8.7.1), while-statement (Section 8.8.1), do-statement (Section 8.8.2), or for-statement (Section 8.8.3) is a boolean-expression. The controlling conditional expression of the `?:` operator (Section 7.12) follows the same rules as a boolean-expression, but for reasons of operator precedence is classified as a conditional-or-expression.

A boolean-expression is required to be of a type that can be implicitly converted to `bool` or of a type that implements `operator true`. If neither requirement is satisfied, a compile-time error occurs.

When a Boolean expression is of a type that cannot be implicitly converted to `bool` but does implement `operator true`, then following evaluation of the expression, the `operator true` implementation provided by that type is invoked to produce a `bool` value.

The `DBBool` struct type in Section 11.4.2 provides an example of a type that implements `operator true` and `operator false`.

8. Statements

C# provides a variety of statements. Most of these statements will be familiar to developers who have programmed in C and C++.

statement:
  labeled-statement
  declaration-statement
  embedded-statement
embedded-statement:
  block
  empty-statement
  expression-statement
  selection-statement
  iteration-statement
  jump-statement
  try-statement
  checked-statement
  unchecked-statement
  lock-statement
  using-statement

The *embedded-statement* nonterminal is used for statements that appear within other statements. The use of *embedded-statement* rather than *statement* excludes the use of declaration statements and labeled statements in these contexts. The example

```
void F(bool b) {
   if (b)
      int i = 44;
}
```

results in a compile-time error because an `if` statement requires an *embedded-statement* rather than a *statement* for its if branch. If this code were permitted, then the variable `i` would be declared, but it could never be used. Note, however, that by placing `i`'s declaration in a block, the example is valid.

8.1 End points and reachability

Every statement has an end point. In intuitive terms, the end point of a statement is the location that immediately follows the statement. The execution rules for composite statements (statements that contain embedded statements) specify the action that is taken when control reaches the end point of an embedded statement. For example, when control reaches the end point of a statement in a block, control is transferred to the next statement in the block.

If a statement can possibly be reached by execution, the statement is said to be reachable. Conversely, if there is no possibility that a statement will be executed, the statement is said to be unreachable.

In the example

```
void F() {
   Console.WriteLine("reachable");
   goto Label;
   Console.WriteLine("unreachable");
   Label:
   Console.WriteLine("reachable");
}
```

the second invocation of `Console.WriteLine` is unreachable because there is no possibility that the statement will be executed.

A warning is reported if the compiler determines that a statement is unreachable. It is specifically not an error for a statement to be unreachable.

To determine whether a particular statement or end point is reachable, the compiler performs flow analysis according to the reachability rules defined for each statement. The flow analysis takes into account the values of constant expressions (Section 7.15) that control the behavior of statements, but the possible values of non-constant expressions are not considered. In other words, for purposes of control flow analysis, a non-constant expression of a given type is considered to have any possible value of that type.

In the example

```
void F() {
   const int i = 1;
   if (i == 2) Console.WriteLine("unreachable");
}
```

the Boolean expression of the `if` statement is a constant expression because both operands of the `==` operator are constants. As the constant expression is evaluated at compile-time, producing the value `false`, the `Console.WriteLine` invocation is considered unreachable. However, if `i` is changed to be a local variable

```
void F() {
   int i = 1;
   if (i == 2) Console.WriteLine("reachable");
}
```

the `Console.WriteLine` invocation is considered reachable, even though, in reality, it will never be executed.

The *block* of a function member is always considered reachable. By successively evaluating the reachability rules of each statement in a block, the reachability of any given statement can be determined.

In the example

```
void F(int x) {
   Console.WriteLine("start");
   if (x < 0) Console.WriteLine("negative");
}
```

the reachability of the second `Console.WriteLine` is determined as follows:

- The first `Console.WriteLine` expression statement is reachable because the block of the `F` method is reachable.
- The end point of the first `Console.WriteLine` expression statement is reachable because that statement is reachable.

- The `if` statement is reachable because the end point of the first `Console.WriteLine` expression statement is reachable.
- The second `Console.WriteLine` expression statement is reachable because the Boolean expression of the `if` statement does not have the constant value `false`.

There are two situations in which it is a compile-time error for the end point of a statement to be reachable:

- Because the `switch` statement does not permit a switch section to "fall through" to the next switch section, it is a compile-time error for the end point of the statement list of a switch section to be reachable. If this error occurs, it is typically an indication that a `break` statement is missing.
- It is a compile-time error for the end point of the block of a function member that computes a value to be reachable. If this error occurs, it is typically an indication that a `return` statement is missing.

8.2 Blocks

A *block* permits multiple statements to be written in contexts where a single statement is allowed.

block:
  { *statement-list*$_{opt}$ }

A *block* consists of an optional *statement-list* (Section 8.2.1), enclosed in braces. If the statement list is omitted, the block is said to be empty.

A block may contain declaration statements (Section 8.5). The scope of a local variable or constant declared in a block is the block.

Within a block, the meaning of a name used in an expression context must always be the same (Section 7.5.2.1).

A block is executed as follows:

- If the block is empty, control is transferred to the end point of the block.
- If the block is not empty, control is transferred to the statement list. When and if control reaches the end point of the statement list, control is transferred to the end point of the block.

The statement list of a block is reachable if the block itself is reachable.

The end point of a block is reachable if the block is empty or if the end point of the statement list is reachable.

8.2.1 Statement lists

A statement list consists of one or more statements written in sequence. Statement lists occur in *blocks* (Section 8.2) and in *switch-blocks* (Section 8.7.2).

statement-list:
  statement
  statement-list statement

A statement list is executed by transferring control to the first statement. When and if control reaches the end point of a statement, control is transferred to the next statement. When and if control reaches the end point of the last statement, control is transferred to the end point of the statement list.

A statement in a statement list is reachable if at least one of the following is true:

- The statement is the first statement and the statement list itself is reachable.
- The end point of the preceding statement is reachable.
- The statement is a labeled statement and the label is referenced by a reachable `goto` statement.

The end point of a statement list is reachable if the end point of the last statement in the list is reachable.

8.3 The empty statement

An *empty-statement* does nothing.

empty-statement:
  ;

An empty statement is used when there are no operations to perform in a context where a statement is required.

Execution of an empty statement simply transfers control to the end point of the statement. Thus, the end point of an empty statement is reachable if the empty statement is reachable.

An empty statement can be used when writing a `while` statement with a null body:

```
bool ProcessMessage() {...}
void ProcessMessages() {
   while (ProcessMessage())
      ;
}
```

Also, an empty statement can be used to declare a label just before the closing "`}`" of a block:

```
void F() {
   ...
   if (done) goto exit;
   ...
   exit: ;
}
```

8.4 Labeled statements

A *labeled-statement* permits a statement to be prefixed by a label. Labeled statements are permitted in blocks, but are not permitted as embedded statements.

labeled-statement:
  identifier : *statement*

A labeled statement declares a label with the name given by the *identifier*. The scope of a label is the whole block in which the label is declared, including any nested blocks. It is a compile-time error for two labels with the same name to have overlapping scopes.

A label can be referenced from `goto` statements (Section 8.9.3) within the scope of the label. This means that `goto` statements can transfer control within blocks and out of blocks, but never into blocks.

Labels have their own declaration space and do not interfere with other identifiers. The example

```
int F(int x) {
   if (x >= 0) goto x;
   x = -x;
   x: return x;
}
```

is valid and uses the name `x` as both a parameter and a label.

Execution of a labeled statement corresponds exactly to execution of the statement following the label.

In addition to the reachability provided by normal flow of control, a labeled statement is reachable if the label is referenced by a reachable `goto` statement. (Exception: If a `goto` statement is inside a `try` that includes a `finally` block, and the labeled statement is outside the `try`, and the end point of the `finally` block is unreachable, then the labeled statement is not reachable from that `goto` statement.)

8.5 Declaration statements

A *declaration-statement* declares a local variable or constant. Declaration statements are permitted in blocks, but are not permitted as embedded statements.

declaration-statement:
  local-variable-declaration ;
  local-constant-declaration ;

8.5.1 Local variable declarations

A *local-variable-declaration* declares one or more local variables.

local-variable-declaration:
  type *local-variable-declarators*
local-variable-declarators:
  local-variable-declarator
  local-variable-declarators , *local-variable-declarator*
local-variable-declarator:
  identifier
  identifier = *local-variable-initializer*
local-variable-initializer:
  expression
  array-initializer

The *type* of a *local-variable-declaration* specifies the type of the variables introduced by the declaration. The type is followed by a list of *local-variable-declarators*, each of which introduces a new variable. A *local-variable-declarator* consists of an *identifier* that names the variable, optionally followed by an "=" token and a *local-variable-initializer* that gives the initial value of the variable.

The value of a local variable is obtained in an expression using a *simple-name* (Section 7.5.2), and the value of a local variable is modified using an *assignment* (Section 7.13). A local variable must be definitely assigned (Section 5.3) at each location where its value is obtained.

The scope of a local variable declared in a *local-variable-declaration* is the block in which the declaration occurs. It is an error to refer to a local variable in a textual position that precedes the *local-variable-declarator* of the local variable. Within the scope of a local variable, it is a compile-time error to declare another local variable or constant with the same name.

A local variable declaration that declares multiple variables is equivalent to multiple declarations of single variables with the same type. Furthermore, a variable initializer in a local variable declaration corresponds exactly to an assignment statement that is inserted immediately after the declaration.

The example

```
void F() {
   int x = 1, y, z = x * 2;
}
```

corresponds exactly to

```
void F() {
   int x; x = 1;
   int y;
   int z; z = x * 2;
}
```

8.5.2 Local constant declarations

A *local-constant-declaration* declares one or more local constants.

local-constant-declaration:
  const *type* *constant-declarators*
constant-declarators:
  constant-declarator
  constant-declarators , *constant-declarator*
constant-declarator:
  identifier = *constant-expression*

The *type* of a *local-constant-declaration* specifies the type of the constants introduced by the declaration. The type is followed by a list of *constant-declarators*, each of which introduces a new constant. A *constant-declarator* consists of an *identifier* that names the constant, followed by an "=" token, followed by a *constant-expression* (Section 7.15) that gives the value of the constant.

The *type* and *constant-expression* of a local constant declaration must follow the same rules as those of a constant member declaration (Section 10.3).

The value of a local constant is obtained in an expression using a *simple-name* (Section 7.5.2).

The scope of a local constant is the block in which the declaration occurs. It is an error to refer to a local constant in a textual position that precedes its *constant-declarator*. Within the scope of a local constant, it is a compile-time error to declare another local variable or constant with the same name.

A local constant declaration that declares multiple constants is equivalent to multiple declarations of single constants with the same type.

8.6 Expression statements

An *expression-statement* evaluates a given expression. The value computed by the expression, if any, is discarded.

expression-statement:
  statement-expression ;
statement-expression:
  invocation-expression
  object-creation-expression
  assignment
  post-increment-expression
  post-decrement-expression
  pre-increment-expression
  pre-decrement-expression

Not all expressions are permitted as *statements*. In particular, expressions such as `x + y` and `x == 1` that merely compute a value (which will be discarded), are not permitted as *statements*.

Execution of an *expression-statement* evaluates the contained *expression* and then transfers control to the end point of the *expression-statement*. The end point of an *expression-statement* is reachable if that *expression-statement* is reachable.

8.7 Selection statements

Selection statements select one of a number of possible statements for execution based on the value of some expression.

selection-statement:
  if-statement
  switch-statement

8.7.1 The if statement

The `if` statement selects a statement for execution based on the value of a Boolean expression.

if-statement:
  if (*boolean-expression*) *embedded-statement*
  if (*boolean-expression*) *embedded-statement* else *embedded-statement*
boolean-expression:
  expression

An `else` part is associated with the lexically nearest preceding `if` that is allowed by the syntax. Thus, an `if` statement of the form

```
if (x) if (y) F(); else G();
```

is equivalent to

```
if (x) {
   if (y) {
      F();
   }
   else {
      G();
   }
}
```

An `if` statement is executed as follows:

- The *boolean-expression* (Section 7.16) is evaluated.
- If the Boolean expression yields `true`, control is transferred to the first embedded statement. When and if control reaches the end point of that statement, control is transferred to the end point of the `if` statement.
- If the Boolean expression yields `false` and if an `else` part is present, control is transferred to the second embedded statement. When and if control reaches the end point of that statement, control is transferred to the end point of the `if` statement.
- If the Boolean expression yields `false` and if an `else` part is not present, control is transferred to the end point of the `if` statement.

The first embedded statement of an `if` statement is reachable if the `if` statement is reachable and the Boolean expression does not have the constant value `false`.

The second embedded statement of an `if` statement, if present, is reachable if the `if` statement is reachable and the Boolean expression does not have the constant value `true`.

The end point of an `if` statement is reachable if the end point of at least one of its embedded statements is reachable. In addition, the end point of an `if` statement with no `else` part is reachable if the `if` statement is reachable and the Boolean expression does not have the constant value `true`.

8.7.2 The switch statement

The switch statement selects for execution a statement list having an associated switch label that corresponds to the value of the switch expression.

switch-statement:
  switch (*expression*) *switch-block*
switch-block:
  { *switch-sections*$_{opt}$ }
switch-sections:
  switch-section
  switch-sections *switch-section*
switch-section:
  switch-labels *statement-list*
switch-labels:
  switch-label
  switch-labels *switch-label*
switch-label:
  case *constant-expression* :
  default :

A *switch-statement* consists of the keyword `switch`, followed by a parenthesized expression (called the switch expression), followed by a *switch-block*. The *switch-block* consists of zero or more *switch-sections*, enclosed in braces. Each *switch-section* consists of one or more *switch-labels* followed by a *statement-list* (Section 8.2.1).

The governing type of a `switch` statement is established by the switch expression. If the type of the switch expression is `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `string`, or an *enum-type*, then that is the governing type of the `switch` statement. Otherwise, exactly one user-defined implicit conversion (Section 6.4) must exist from the type of the switch expression to one of the following possible governing types: `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `string`. If no such implicit conversion exists, or if more than one such implicit conversion exists, a compile-time error occurs.

The constant expression of each `case` label must denote a value of a type that is implicitly convertible (Section 6.1) to the governing type of the `switch` statement. A compile-time error occurs if two or more `case` labels in the same `switch` statement specify the same constant value.

There can be at most one `default` label in a switch statement.

A `switch` statement is executed as follows:

- The switch expression is evaluated and converted to the governing type.
- If one of the constants specified in a `case` label in the same `switch` statement is equal to the value of the switch expression, control is transferred to the statement list following the matched `case` label.
- If none of the constants specified in `case` labels in the same `switch` statement is equal to the value of the switch expression, and if a `default` label is present, control is transferred to the statement list following the `default` label.
- If none of the constants specified in `case` labels in the same `switch` statement is equal to the value of the switch expression, and if no `default` label is present, control is transferred to the end point of the `switch` statement.

If the end point of the statement list of a switch section is reachable, a compile-time error occurs. This is known as the "no fall through" rule. The example

```
switch (i) {
case 0:
   CaseZero();
   break;
case 1:
   CaseOne();
   break;
default:
   CaseOthers();
   break;
}
```

is valid because no switch section has a reachable end point. Unlike C and C++, execution of a switch section is not permitted to

"fall through" to the next switch section, and the example

```
switch (i) {
case 0:
   CaseZero();
case 1:
   CaseZeroOrOne();
default:
   CaseAny();
}
```

results in a compile-time error. When execution of a switch section is to be followed by execution of another switch section, an explicit `goto case` or `goto default` statement must be used:

```
switch (i) {
case 0:
   CaseZero();
   goto case 1;
case 1:
   CaseZeroOrOne();
   goto default;
default:
   CaseAny();
   break;
}
```

Multiple labels are permitted in a *switch-section*. The example

```
switch (i) {
case 0:
   CaseZero();
   break;
case 1:
   CaseOne();
   break;
case 2:
default:
   CaseTwo();
   break;
}
```

is valid. The example does not violate the "no fall through" rule because the labels `case 2:` and `default:` are part of the same *switch-section*.

The "no fall through" rule prevents a common class of bugs that occur in C and C++ when `break` statements are accidentally omitted. In addition, because of this rule, the switch sections of a `switch` statement can be arbitrarily rearranged without affecting the behavior of the statement. For example, the sections of the `switch` statement above can be reversed without affecting the behavior of the statement:

```
switch (i) {
default:
   CaseAny();
   break;
case 1:
   CaseZeroOrOne();
   goto default;
case 0:
   CaseZero();
   goto case 1;
}
```

The statement list of a switch section typically ends in a `break`, `goto case`, or `goto default` statement, but any construct that renders the end point of the statement list unreachable is permitted. For example, a `while` statement controlled by the Boolean expression `true` is known to never reach its end point. Likewise, a `throw` or `return` statement always transfers control elsewhere

and never reaches its end point. Thus, the following example is valid:

```
switch (i) {
case 0:
   while (true) F();
case 1:
   throw new ArgumentException();
case 2:
   return;
}
```

The governing type of a `switch` statement may be the type `string`. For example:

```
void DoCommand(string command) {
   switch (command.ToLower()) {
   case "run":
      DoRun();
      break;
   case "save":
      DoSave();
      break;
   case "quit":
      DoQuit();
      break;
   default:
      InvalidCommand(command);
      break;
   }
}
```

Like the string equality operators (Section 7.9.7), the `switch` statement is case sensitive and will execute a given switch section only if the switch expression string exactly matches a `case` label constant.

When the governing type of a `switch` statement is `string`, the value `null` is permitted as a case label constant.

The *statement-lists* of a *switch-block* may contain declaration statements (Section 8.5). The scope of a local variable or constant declared in a switch block is the switch block.

Within a switch block, the meaning of a name used in an expression context must always be the same (Section 7.5.2.1).

The statement list of a given switch section is reachable if the `switch` statement is reachable and at least one of the following is true:

- The switch expression is a non-constant value.
- The switch expression is a constant value that matches a `case` label in the switch section.
- The switch expression is a constant value that doesn't match any `case` label, and the switch section contains the `default` label.
- A switch label of the switch section is referenced by a reachable `goto case` or `goto default` statement.

The end point of a `switch` statement is reachable if at least one of the following is true:

- The `switch` statement contains a reachable `break` statement that exits the `switch` statement.
- The `switch` statement is reachable, the switch expression is a non-constant value, and no `default` label is present.
- The `switch` statement is reachable, the switch expression is a constant value that doesn't match any `case` label, and no `default` label is present.

8.8 Iteration statements

Iteration statements repeatedly execute an embedded statement.

iteration-statement:
  while-statement
  do-statement
  for-statement
  foreach-statement

8.8.1 The while statement

The `while` statement conditionally executes an embedded statement zero or more times.

while-statement:
  while (*boolean-expression*) *embedded-statement*

A `while` statement is executed as follows:

- The *boolean-expression* (Section 7.16) is evaluated.
- If the Boolean expression yields `true`, control is transferred to the embedded statement. When and if control reaches the end point of the embedded statement (possibly from execution of a `continue` statement), control is transferred to the beginning of the `while` statement.
- If the Boolean expression yields `false`, control is transferred to the end point of the `while` statement.

Within the embedded statement of a `while` statement, a `break` statement (Section 8.9.1) may be used to transfer control to the end point of the `while` statement (thus ending iteration of the embedded statement), and a `continue` statement (Section 8.9.2) may be used to transfer control to the end point of the embedded statement.

The embedded statement of a `while` statement is reachable if the `while` statement is reachable and the Boolean expression does not have the constant value `false`.

The end point of a `while` statement is reachable if at least one of the following is true:

- The `while` statement contains a reachable `break` statement that exits the `while` statement.
- The `while` statement is reachable and the Boolean expression does not have the constant value `true`.

8.8.2 The do statement

The `do` statement conditionally executes an embedded statement one or more times.

do-statement:
  do *embedded-statement* while (*boolean-expression*) ;

A `do` statement is executed as follows:

- Control is transferred to the embedded statement.
- When and if control reaches the end point of the embedded statement (possibly from execution of a `continue` statement), the *boolean-expression* (Section 7.16) is evaluated. If the Boolean expression yields `true`, control is transferred to the beginning of the `do` statement. Otherwise, control is transferred to the end point of the `do` statement.

Within the embedded statement of a `do` statement, a `break` statement (Section 8.9.1) may be used to transfer control to the end point of the `do` statement (thus ending iteration of the embedded statement), and a `continue` statement (Section 8.9.2) may be used to transfer control to the end point of the embedded statement (thus performing another iteration of the `do` statement).

The embedded statement of a `do` statement is reachable if the `do` statement is reachable.

The end point of a `do` statement is reachable if at least one of the following is true:

- The `do` statement contains a reachable `break` statement that exits the `do` statement.
- The end point of the embedded statement is reachable and the Boolean expression does not have the constant value `true`.

8.8.3 The for statement

The `for` statement evaluates a sequence of initialization expressions and then, while a condition is true, repeatedly executes an embedded statement and evaluates a sequence of iteration expressions.

for-statement:
  for (*for-initializer*$_{opt}$; *for-condition*$_{opt}$; *for-iterator*$_{opt}$) *embedded-statement*
for-initializer:
  local-variable-declaration
  statement-expression-list
for-condition:
  boolean-expression
for-iterator:
  statement-expression-list
statement-expression-list:
  statement-expression
  statement-expression-list , *statement-expression*

The *for-initializer*, if present, consists of either a *local-variable-declaration* (Section 8.5.1) or a list of *statement-expressions* (Section 8.6) separated by commas. The scope of a local variable declared by a *for-initializer* starts at the *local-variable-declarator* for the variable and extends to the end of the embedded statement. The scope includes the *for-condition* and the *for-iterator*.

The *for-condition*, if present, must be a *boolean-expression* (Section 7.16).

The *for-iterator*, if present, consists of a list of *statement-expressions* (Section 8.6) separated by commas.

A for statement is executed as follows:

- If a *for-initializer* is present, the variable initializers or statement expressions are executed in the order they are written. This step is only performed once.
- If a *for-condition* is present, it is evaluated.
- If the *for-condition* is not present or if the evaluation yields `true`, control is transferred to the embedded statement. When and if control reaches the end point of the embedded statement (possibly from execution of a `continue` statement), the expressions of the *for-iterator*, if any, are evaluated in sequence, and then another iteration is performed, starting with evaluation of the *for-condition* in the step above.
- If the *for-condition* is present and the evaluation yields `false`, control is transferred to the end point of the `for` statement.

Within the embedded statement of a `for` statement, a `break` statement (Section 8.9.1) may be used to transfer control to the end point of the `for` statement (thus ending iteration of the embedded statement), and a `continue` statement (Section 8.9.2) may be used to transfer control to the end point of the embedded statement (thus executing the *for-iterator* and performing another iteration of the `for` statement, starting with the *for-condition*).

The embedded statement of a `for` statement is reachable if one of the following is true:

- The `for` statement is reachable and no *for-condition* is present.
- The `for` statement is reachable and a *for-condition* is present and does not have the constant value `false`.

The end point of a `for` statement is reachable if at least one of the following is true:

- The `for` statement contains a reachable `break` statement that exits the `for` statement.
- The `for` statement is reachable and a *for-condition* is present and does not have the constant value `true`.

8.8.4 The foreach statement

The `foreach` statement enumerates the elements of a collection, executing an embedded statement for each element of the collection.

foreach-statement:
foreach (*type* *identifier* in *expression*) *embedded-statement*

The *type* and *identifier* of a `foreach` statement declare the iteration variable of the statement. The iteration variable corresponds to a read-only local variable with a scope that extends over the embedded statement. During execution of a `foreach` statement, the iteration variable represents the collection element for which an iteration is currently being performed. A compile-time error occurs if the embedded statement attempts to modify the iteration variable (by assignment or the `++` and `--` operators) or pass the iteration variable as a `ref` or `out` parameter.

The type of the *expression* of a `foreach` statement must be a collection type (as defined below), and an explicit conversion (Section 6.2) must exist from the element type of the collection to the type of the iteration variable. If *expression* has the value `null`, a `System.NullReferenceException` is thrown.

A type `C` is said to be a collection type if it implements the `System.Collections.IEnumerable` interface or implements the collection pattern by meeting all of the following criteria:

- `C` contains a `public` instance method with the signature `GetEnumerator()` that returns a *struct-type*, *class-type*, or *interface-type*, which is called `E` in the following text.
- `E` contains a `public` instance method with the signature `MoveNext()` and the return type `bool`.
- `E` contains a `public` instance property named `Current` that permits reading the current value. The type of this property is said to be the element type of the collection type.

A type that implements `IEnumerable` is also a collection type, even if it does not satisfy the conditions above. (This is possible if it implements some of the `IEnumerable` members by means of explicit interface member implementation, as described in Section 13.4.1.)

The `System.Array` type (Section 12.1.1) is a collection type, and since all array types derive from `System.Array`, any array type expression is permitted in a `foreach` statement. The order in which `foreach` traverses the elements of an array is as follows: For single-dimensional arrays, elements are traversed in increasing index order, starting with index `0` and ending with index `Length - 1`. For multi-dimensional arrays, elements are traversed such that the indices of the rightmost dimension are increased first, then the next left dimension, and so on to the left.

A `foreach` statement of the form:

```
foreach (ElementType element in collection) statement
```

corresponds to one of two possible expansions:

- If the `collection` expression is of a type that implements the collection pattern (as defined above), the expansion of the `foreach` statement is:

  ```
  E enumerator = (collection).GetEnumerator();
  try {
     while (enumerator.MoveNext()) {
        ElementType element = (ElementType)enumerator.Current;
        statement;
     }
  }
  finally {
     IDisposable disposable = enumerator as System.IDisposable;
     if (disposable != null) disposable.Dispose();
  }
  ```

  Significant optimizations of the above are often easily available. If the type `E` implements `System.IDisposable`, then the expression (`enumerator as System.IDisposable`) will always be non-null and the implementation can safely substitute a simple conversion for a possibly more expensive type test. Conversely, if the type `E` is `sealed` and does not implement

`System.IDisposable`, then the expression (`enumerator as System.IDisposable`) will always evaluate to null. In this case, the implementation can safely optimize away the entire finally clause.

- Otherwise, the `collection` expression is of a type that implements `System.IEnumerable`, and the expansion of the `foreach` statement is:

```
IEnumerator enumerator =
        ((System.Collections.IEnumerable)(collection)).GetEnumerator();
try {
   while (enumerator.MoveNext()) {
      ElementType element = (ElementType)enumerator.Current;
      statement;
   }
}
finally {
   IDisposable disposable = enumerator as System.IDisposable;
   if (disposable != null) disposable.Dispose();
}
```

In either expansion, the `enumerator` variable is a temporary variable that is inaccessible in, and invisible to, the embedded `statement`, and the element variable is read-only in the embedded `statement`.

The following example prints out each value in a two-dimensional array, in element order:

```
using System;
class Test
{
   static void Main() {
      double[,] values = {
         {1.2, 2.3, 3.4, 4.5},
         {5.6, 6.7, 7.8, 8.9}
      };
      foreach (double elementValue in values)
         Console.Write("{0} ", elementValue);
      Console.WriteLine();
   }
}
```

The output produced is as follows:

```
1.2 2.3 3.4 4.5 5.6 6.7 7.8 8.9
```

8.9 Jump statements

Jump statements unconditionally transfer control.

jump-statement:
  break-statement
  continue-statement
  goto-statement
  return-statement
  throw-statement

The location to which a jump statement transfers control is called the target of the jump statement.

When a jump statement occurs within a block, and the target of that jump statement is outside that block, the jump statement is said to exit the block. While a jump statement may transfer control out of a block, it can never transfer control into a block.

Execution of jump statements is complicated by the presence of intervening `try` statements. In the absence of such `try` statements, a jump statement unconditionally transfers control from the jump statement to its target. In the presence of such intervening `try` statements, execution is more complex. If the jump statement exits one or more `try` blocks with associated `finally` blocks, control is initially transferred to the `finally` block of the innermost `try` statement. When and if control reaches the end point of a `finally` block, control is transferred to the `finally` block of the next enclosing `try` statement. This process is repeated until the `finally` blocks of all intervening `try` statements have been executed.

In the example

```
using System;
class Test
{
   static void Main() {
      while (true) {
         try {
            try {
               Console.WriteLine("Before break");
               break;
            }
            finally {
               Console.WriteLine("Innermost finally block");
            }
         }
         finally {
            Console.WriteLine("Outermost finally block");
         }
      }
      Console.WriteLine("After break");
   }
}
```

the finally blocks associated with two try statements are executed before control is transferred to the target of the jump statement.

The output produced is as follows:

```
Before break
Innermost finally block
Outermost finally block
After break
```

8.9.1 The break statement

The `break` statement exits the nearest enclosing `switch`, `while`, `do`, `for`, or `foreach` statement.

break-statement:
  break ;

The target of a `break` statement is the end point of the nearest enclosing `switch`, `while`, `do`, `for`, or `foreach` statement. If a `break` statement is not enclosed by a `switch`, `while`, `do`, `for`, or `foreach` statement, a compile-time error occurs.

When multiple `switch`, `while`, `do`, `for`, or `foreach` statements are nested within each other, a `break` statement applies only to the innermost statement. To transfer control across multiple nesting levels, a `goto` statement (Section 8.9.3) must be used.

A `break` statement cannot exit a `finally` block (Section 8.10). When a `break` statement occurs within a `finally` block, the target of the `break` statement must be within the same `finally` block; otherwise, a compile-time error occurs.

A `break` statement is executed as follows:

- If the `break` statement exits one or more `try` blocks with associated `finally` blocks, control is initially transferred to the `finally` block of the innermost `try` statement. When and if control reaches the end point of a `finally` block, control is transferred to the `finally` block of the next enclosing `try` statement. This process is repeated until the `finally` blocks of all intervening `try` statements have been executed.
- Control is transferred to the target of the `break` statement.

Because a `break` statement unconditionally transfers control elsewhere, the end point of a `break` statement is never reachable.

8.9.2 The continue statement

The `continue` statement starts a new iteration of the nearest enclosing `while`, `do`, `for`, or `foreach` statement.

continue-statement:
  continue ;

The target of a `continue` statement is the end point of the embedded statement of the nearest enclosing `while`, `do`, `for`, or `foreach` statement. If a `continue` statement is not enclosed by a `while`, `do`, `for`, or `foreach` statement, a compile-time error occurs.

When multiple `while`, `do`, `for`, or `foreach` statements are nested within each other, a `continue` statement applies only to the innermost statement. To transfer control across multiple nesting levels, a `goto` statement (Section 8.9.3) must be used.

A `continue` statement cannot exit a `finally` block (Section 8.10). When a `continue` statement occurs within a `finally` block, the target of the `continue` statement must be within the same `finally` block; otherwise, a compile-time error occurs.

A `continue` statement is executed as follows:

- If the `continue` statement exits one or more `try` blocks with associated `finally` blocks, control is initially transferred to the `finally` block of the innermost `try` statement. When and if control reaches the end point of a `finally` block, control is transferred to the `finally` block of the next enclosing `try` statement. This process is repeated until the `finally` blocks of all intervening `try` statements have been executed.
- Control is transferred to the target of the `continue` statement.

Because a `continue` statement unconditionally transfers control elsewhere, the end point of a `continue` statement is never reachable.

8.9.3 The goto statement

The `goto` statement transfers control to a statement that is marked by a label.

goto-statement:
  goto *identifier* ;
  goto case *constant-expression* ;
  goto default ;

The target of a `goto` *identifier* statement is the labeled statement with the given label. If a label with the given name does not exist in the current function member, or if the `goto` statement is not within the scope of the label, a compile-time error occurs. This rule permits the use of a `goto` statement to transfer control *out of* a nested scope, but not *into* a nested scope. In the example

```
using System;
class Test
{
   static void Main(string[] args) {
      string[,] table = {
         {"Red", "Blue", "Green"},
         {"Monday", "Wednesday", "Friday"}
      };
      foreach (string str in args) {
         int row, colm;
         for (row = 0; row <= 1; ++row)
            for (colm = 0; colm <= 2; ++colm)
               if (str == table[row,colm])
                   goto done;
         Console.WriteLine("{0} not found", str);
         continue;
   done:
         Console.WriteLine("Found {0} at [{1}][{2}]", str, row, colm);
      }
   }
}
```

a `goto` statement is used to transfer control out of a nested scope.

The target of a `goto case` statement is the statement list in the immediately enclosing switch statement (Section 8.7.2), which contains a `case` label with the given constant value. If the `goto case` statement is not enclosed by a `switch` statement, if the *constant-expression* is not implicitly convertible (Section 6.1) to the governing type of the nearest enclosing `switch` statement, or if the nearest enclosing `switch` statement does not contain a `case` label with the given constant value, a compile-time error occurs.

The target of a `goto default` statement is the statement list in the immediately enclosing switch statement (Section 8.7.2), which contains a default label. If the `goto default` statement is not enclosed by a `switch` statement, or if the nearest enclosing `switch` statement does not contain a `default` label, a compile-time error occurs.

A `goto` statement cannot exit a `finally` block (Section 8.10). When a `goto` statement occurs within a `finally` block, the target of the `goto` statement must be within the same `finally` block, or otherwise a compile-time error occurs.

A `goto` statement is executed as follows:

- If the `goto` statement exits one or more `try` blocks with associated `finally` blocks, control is initially transferred to the `finally` block of the innermost `try` statement. When and if control reaches the end point of a `finally` block, control is transferred to the `finally` block of the next enclosing `try` statement. This process is repeated until the `finally` blocks of all intervening `try` statements have been executed.
- Control is transferred to the target of the `goto` statement.

Because a `goto` statement unconditionally transfers control elsewhere, the end point of a `goto` statement is never reachable.

8.9.4 The return statement

The `return` statement returns control to the caller of the function member in which the `return` statement appears.

return-statement:
  return *expression*$_{opt}$;

A `return` statement with no expression can be used only in a function member that does not compute a value, that is, a method with the return type `void`, the `set` accessor of a property or indexer, the `add` and `remove` accessors of an event, an instance constructor, a static constructor, or a destructor.

A `return` statement with an expression can only be used in a function member that computes a value, that is, a method with a non-void return type, the `get` accessor of a property or indexer, or a user-defined operator. An implicit conversion (Section 6.1) must exist from the type of the expression to the return type of the containing function member.

It is a compile-time error for a `return` statement to appear in a `finally` block (Section 8.10).

A `return` statement is executed as follows:

- If the `return` statement specifies an expression, the expression is evaluated and the resulting value is converted to the return type of the containing function member by an implicit conversion. The result of the conversion becomes the value returned to the caller.
- If the `return` statement is enclosed by one or more `try` blocks with associated `finally` blocks, control is initially transferred to the `finally` block of the innermost `try` statement. When and if control reaches the end point of a `finally` block, control is transferred to the `finally` block of the next enclosing `try` statement. This process is repeated until the `finally` blocks of all enclosing `try` statements have been executed.
- Control is returned to the caller of the containing function member.

Because a `return` statement unconditionally transfers control elsewhere, the end point of a `return` statement is never reachable.

8.9.5 The throw statement

The `throw` statement throws an exception.

throw-statement:
  throw *expression*$_{opt}$;

A `throw` statement with an expression throws the value produced by evaluating the expression. The expression must denote a value of the class type `System.Exception` or of a class type that derives from `System.Exception`. If evaluation of the expression produces `null`, a `System.NullReferenceException` is thrown instead.

A `throw` statement with no expression can be used only in a `catch` block, in which case that statement re-throws the exception that is currently being handled by that `catch` block.

Because a `throw` statement unconditionally transfers control elsewhere, the end point of a `throw` statement is never reachable.

When an exception is thrown, control is transferred to the first `catch` clause in an enclosing `try` statement that can handle the exception. The process that takes place from the point of the exception being thrown to the point of transferring control to a suitable exception handler is known as exception propagation. Propagation of an exception consists of repeatedly evaluating the following steps until a `catch` clause that matches the exception is found. In this description, the throw point is initially the location at which the exception is thrown.

- In the current function member, each `try` statement that encloses the throw point is examined. For each statement `S`, starting with the innermost `try` statement and ending with the outermost `try` statement, the following steps are evaluated:
- If the `try` block of `S` encloses the throw point and if S has one or more `catch` clauses, the `catch` clauses are examined in order of appearance to locate a suitable handler for the exception. The first `catch` clause that specifies the exception type or a base type of the exception type is considered a match. A general `catch` clause (Section 8.10) is considered a match for any exception type. If a matching `catch` clause is located, the exception propagation is completed by transferring control to the block of that `catch` clause.
- Otherwise, if the `try` block or a `catch` block of `S` encloses the throw point and if `S` has a `finally` block, control is transferred to the `finally` block. If the `finally` block throws another exception, processing of the current exception is terminated. Otherwise, when control reaches the end point of the `finally` block, processing of the current exception is continued.
- If an exception handler was not located in the current function member invocation, the function member invocation is terminated. The steps above are then repeated for the caller of the function member with a throw point corresponding to the statement from which the function member was invoked.
- If the exception processing terminates all function member invocations in the current thread, indicating that the thread has no handler for the exception, then the thread is itself terminated. The impact of such termination is implementation-defined.

8.10 The try statement

The `try` statement provides a mechanism for catching exceptions that occur during execution of a block. Furthermore, the `try` statement provides the ability to specify a block of code that is always executed when control leaves the `try` statement.

try-statement:
  try *block* *catch-clauses*
  try *block* *finally-clause*
  try *block* *catch-clauses* *finally-clause*
catch-clauses:
  specific-catch-clauses *general-catch-clause*$_{opt}$
  specific-catch-clauses$_{opt}$ *general-catch-clause*
specific-catch-clauses:
  specific-catch-clause
  specific-catch-clauses *specific-catch-clause*
specific-catch-clause:
  catch (*class-type* *identifier*$_{opt}$) *block*
general-catch-clause:
  catch *block*
finally-clause:
  finally *block*

There are three possible forms of `try` statements:

- A `try` block followed by one or more `catch` blocks.
- A `try` block followed by a `finally` block.
- A `try` block followed by one or more `catch` blocks followed by a `finally` block.

When a `catch` clause specifies a *class-type*, the type must be `System.Exception` or a type that derives from `System.Exception`.

When a `catch` clause specifies both a *class-type* and an *identifier*, an exception variable of the given name and type is declared. The exception variable corresponds to a local variable with a scope that extends over the `catch` block. During execution of the `catch` block, the exception variable represents the exception currently being handled. For purposes of definite assignment checking, the exception variable is considered definitely assigned in its entire scope.

Unless a `catch` clause includes an exception variable name, it is impossible to access the exception object in the `catch` block.

A `catch` clause that specifies neither an exception type nor an exception variable name is called a general `catch` clause. A `try` statement can only have one general `catch` clause, and if one is present it must be the last `catch` clause.

Some programming languages may support exceptions that are not representable as an object derived from `System.Exception`, although such exceptions could never be generated by C# code. A general catch clause may be used to catch such exceptions. Thus, a general catch clause is semantically different from one that specifies the type `System.Exception`, in that the former may also catch exceptions from other languages.

In order to locate a handler for an exception, `catch` clauses are examined in lexical order. A compile-time error occurs if a `catch` clause specifies a type that is the same as, or is derived from, a type that was specified in an earlier `catch` clause for the same `try`. Without this restriction, it would be possible to write unreachable `catch` clauses.

Within a `catch` block, a `throw` statement (Section 8.9.5) with no expression can be used to re-throw the exception that was caught by the `catch` block. Assignments to an exception variable do not alter the exception that is re-thrown.

In the example

```
using System;
class Test
{
   static void F() {
      try {
         G();
      }
      catch (Exception e) {
         Console.WriteLine("Exception in F: " + e.Message);
         e = new Exception("F");
         throw;            // re-throw
```

```
        }
    }
    static void G() {
        throw new Exception("G");
    }
    static void Main() {
        try {
            F();
        }
        catch (Exception e) {
            Console.WriteLine("Exception in Main: " + e.Message);
        }
    }
}
```

the method `F` catches an exception, writes some diagnostic information to the console, alters the exception variable, and re-throws the exception. The exception that is re-thrown is the original exception, so the output produced is:

```
Exception in F: G
Exception in Main: G
```

If the first catch block had thrown `e` instead of rethrowing the current exception, the output produced is would be as follows:

```
Exception in F: G
Exception in Main: F
```

It is a compile-time error for a `break`, `continue`, or `goto` statement to transfer control out of a `finally` block. When a `break`, `continue`, or `goto` statement occurs in a `finally` block, the target of the statement must be within the same `finally` block, or otherwise a compile-time error occurs.

It is a compile-time error for a `return` statement to occur in a `finally` block.

A `try` statement is executed as follows:

- Control is transferred to the `try` block.
- When and if control reaches the end point of the `try` block:
- If the `try` statement has a `finally` block, the `finally` block is executed.
- Control is transferred to the end point of the `try` statement.
- If an exception is propagated to the `try` statement during execution of the `try` block:
- The `catch` clauses, if any, are examined in order of appearance to locate a suitable handler for the exception. The first `catch` clause that specifies the exception type or a base type of the exception type is considered a match. A general `catch` clause is considered a match for any exception type. If a matching `catch` clause is located:
- If the matching `catch` clause declares an exception variable, the exception object is assigned to the exception variable.
- Control is transferred to the matching `catch` block.
- When and if control reaches the end point of the `catch` block:
- If the `try` statement has a `finally` block, the `finally` block is executed.
- Control is transferred to the end point of the `try` statement.
- If an exception is propagated to the `try` statement during execution of the `catch` block:
- If the `try` statement has a `finally` block, the `finally` block is executed.
- The exception is propagated to the next enclosing `try` statement.
- If the `try` statement has no `catch` clauses or if no `catch` clause matches the exception:
- If the `try` statement has a `finally` block, the `finally` block is executed.
- The exception is propagated to the next enclosing `try` statement.

The statements of a `finally` block are always executed when control leaves a `try` statement. This is true whether the control transfer occurs as a result of normal execution, as a result of executing a `break`, `continue`, `goto`, or `return` statement, or as a result of propagating an exception out of the `try` statement.

If an exception is thrown during execution of a `finally` block, the exception is propagated to the next enclosing `try` statement. If

another exception was in the process of being propagated, that exception is lost. The process of propagating an exception is discussed further in the description of the `throw` statement (Section 8.9.5).

The `try` block of a `try` statement is reachable if the `try` statement is reachable.

A `catch` block of a `try` statement is reachable if the `try` statement is reachable.

The `finally` block of a `try` statement is reachable if the `try` statement is reachable.

The end point of a `try` statement is reachable if both of the following are true:

- The end point of the `try` block is reachable or the end point of at least one `catch` block is reachable.
- If a `finally` block is present, the end point of the `finally` block is reachable.

8.11 The checked and unchecked statements

The `checked` and `unchecked` statements are used to control the overflow checking context for integral-type arithmetic operations and conversions.

checked-statement:
  checked *block*
unchecked-statement:
  unchecked *block*

The `checked` statement causes all expressions in the *block* to be evaluated in a checked context, and the `unchecked` statement causes all expressions in the *block* to be evaluated in an unchecked context.

The `checked` and `unchecked` statements are precisely equivalent to the `checked` and `unchecked` operators (Section 7.5.12), except that they operate on blocks instead of expressions.

8.12 The lock statement

The `lock` statement obtains the mutual-exclusion lock for a given object, executes a statement, and then releases the lock.

lock-statement:
lock (*expression*) *embedded-statement*

The expression of a `lock` statement must denote a value of a *reference-type*. No implicit boxing conversion (Section 6.1.5) is ever performed for the expression of a `lock` statement, and thus it is a compile-time error for the expression to denote a value of a *value-type*.

A `lock` statement of the form

```
lock (x) ...
```

where `x` is an expression of a *reference-type*, is precisely equivalent to

```
System.Threading.Monitor.Enter(x);
try {
   ...
}
finally {
   System.Threading.Monitor.Exit(x);
}
```

except that `x` is only evaluated once.

While a *mutual-exclusion* lock is held, code executing in the same execution thread can also obtain and release the lock. However, code executing in other threads is blocked from obtaining the lock until the lock is released.

The `System.Type` object of a class can conveniently be used as the mutual-exclusion lock for static methods of the class. For example:

```
class Cache
{
   public static void Add(object x) {
      lock (typeof(Cache)) {
         ...
      }
   }
   public static void Remove(object x) {
      lock (typeof(Cache)) {
         ...
      }
   }
}
```

8.13 The using statement

The `using` statement obtains one or more resources, executes a statement, and then disposes of the resource.

using-statement:
  using (*resource-acquisition*) *embedded-statement*
resource-acquisition:
  local-variable-declaration
  expression

A resource is a class or struct that implements `System.IDisposable`, which includes a single parameterless method named `Dispose`. Code that is using a resource can call `Dispose` to indicate that the resource is no longer needed. If `Dispose` is not called, then automatic disposal eventually occurs as a consequence of garbage collection.

If the form of *resource-acquisition* is *local-variable-declaration* then the type of the *local-variable-declaration* must be `System.IDisposable` or a type that can be implicitly converted to `System.IDisposable`. If the form of *resource-acquisition* is *expression* then this expression must be of type `System.IDisposable` or a type that can be implicitly converted to `System.IDisposable`.

Local variables declared in a *resource-acquisition* are read-only, and must include an initializer. A compile-time error occurs if the embedded statement attempts to modify these local variables (by assignment or the ++ and -- operators) or pass them as `ref` or `out` parameters.

A `using` statement is translated into three parts: acquisition, usage, and disposal. Usage of the resource is implicitly enclosed in a `try` statement that includes a `finally` clause. This `finally` clause disposes of the resource. If a `null` resource is acquired, then no call to `Dispose` is made, and no exception is thrown.

A `using` statement of the form

```
using (ResourceType resource = expression) statement
```

corresponds to one of two possible expansions. When `ResourceType` is a value type, the expansion is

```
{
   ResourceType resource = expression;
   try {
      statement;
   }
   finally {
      ((IDisposable)resource).Dispose();
   }
}
```

Otherwise, when `ResourceType` is a reference type, the expansion is

```
{
   ResourceType resource = expression;
   try {
      statement;
   }
   finally {
      if (resource != null) ((IDisposable)resource).Dispose();
   }
}
```

In either expansion, the `resource` variable is read-only in the embedded statement.

A `using` statement of the form

```
using (expression) statement
```

has the same two possible expansions, but in this case `ResourceType` is implicitly the compile-time type of the `expression`, and the `resource` variable is inaccessible in, and invisible to, the embedded statement.

When a *resource-acquisition* takes the form of a *local-variable-declaration*, it is possible to acquire multiple resources of a given type. A `using` statement of the form

```
using (ResourceType r1 = e1, r2 = e2, ..., rN = eN) statement
```

is precisely equivalent to a sequence of nested `using` statements:

```
using (ResourceType r1 = e1)
   using (ResourceType r2 = e2)
      ...
         using (ResourceType rN = eN)
            statement
```

The example below creates a file named `log.txt` and writes two lines of text to the file. The example then opens that same file for reading and copies the contained lines of text to the console.

```
using System;
using System.IO;
class Test
{
   static void Main() {
      using (TextWriter w = File.CreateText("log.txt")) {
         w.WriteLine("This is line one");
         w.WriteLine("This is line two");
      }
      using (TextReader r = File.OpenText("log.txt")) {
         string s;
         while ((s = r.ReadLine()) != null) {
            Console.WriteLine(s);
         }
      }
   }
}
```

Since the `TextWriter` and `TextReader` classes implement the `IDisposable` interface, the example can use `using` statements to ensure that the underlying file is properly closed following the write or read operations.

9. Namespaces

C# programs are organized using namespaces. Namespaces are used both as an "internal" organization system for a program, and as an "external" organization system — a way of presenting program elements that are exposed to other programs.

Using directives (Section 9.3) are provided to facilitate the use of namespaces.

9.1 Compilation units

A *compilation-unit* defines the overall structure of a source file. A compilation unit consists of zero or more *using-directives* followed by zero or more *global-attributes* followed by zero or more *namespace-member-declarations*.

compilation-unit:
  using-directives$_{opt}$ *global-attributes*$_{opt}$ *namespace-member-declarations*$_{opt}$

A C# program consists of one or more compilation units, each contained in a separate source file. When a C# program is compiled, all of the compilation units are processed together. Thus, compilation units can depend on each other, possibly in a circular fashion.

The *using-directives* of a compilation unit affect the *global-attributes* and *namespace-member-declarations* of that compilation unit, but have no effect on other compilation units.

The *global-attributes* (Section 17) of a compilation unit permit the specification of attributes for the target assembly and module. Assemblies and modules act as physical containers for types. An assembly may consist of several physically separate modules.

The *namespace-member-declarations* of each compilation unit of a program contribute members to a single declaration space called the global namespace. For example:

File `A.cs`:

```
class A {}
```

File `B.cs`:

```
class B {}
```

The two compilation units contribute to the single global namespace, in this case declaring two classes with the fully qualified names `A` and `B`. Because the two compilation units contribute to the same declaration space, it would have been an error if each contained a declaration of a member with the same name.

9.2 Namespace declarations

A *namespace-declaration* consists of the keyword `namespace`, followed by a namespace name and body, optionally followed by a semicolon.

namespace-declaration:
  namespace *qualified-identifier* *namespace-body* $;_{opt}$
qualified-identifier:
  identifier
  qualified-identifier . *identifier*
namespace-body:
  { $using\text{-}directives_{opt}$ $namespace\text{-}member\text{-}declarations_{opt}$ }

A *namespace-declaration* may occur as a top-level declaration in a *compilation-unit* or as a member declaration within another *namespace-declaration*. When a *namespace-declaration* occurs as a top-level declaration in a *compilation-unit*, the namespace becomes a member of the global namespace. When a *namespace-declaration* occurs within another *namespace-declaration*, the inner namespace becomes a member of the outer namespace. In either case, the name of a namespace must be unique within the containing namespace.

Namespaces are implicitly `public` and the declaration of a namespace cannot include any access modifiers.

Within a *namespace-body*, the optional *using-directives* import the names of other namespaces and types, allowing them to be referenced directly instead of through qualified names. The optional *namespace-member-declarations* contribute members to the declaration space of the namespace. Note that all *using-directives* must appear before any member declarations.

The *qualified-identifier* of a *namespace-declaration* may be a single identifier or a sequence of identifiers separated by "`.`" tokens. The latter form permits a program to define a nested namespace without lexically nesting several namespace declarations. For example,

```
namespace N1.N2
{
   class A {}
   class B {}
}
```

is semantically equivalent to

```
namespace N1
{
   namespace N2
   {
      class A {}
      class B {}
   }
}
```

Namespaces are open-ended, and two namespace declarations with the same fully qualified name contribute to the same declaration space (Section 3.3). In the example

```
namespace N1.N2
{
   class A {}
}
namespace N1.N2
{
   class B {}
}
```

the two namespace declarations above contribute to the same declaration space, in this case declaring two classes with the fully qualified names `N1.N2.A` and `N1.N2.B`. Because the two declarations contribute to the same declaration space, it would have been an error if each contained a declaration of a member with the same name.

9.3 Using directives

Using directives facilitate the use of namespaces and types defined in other namespaces. Using directives impact the name resolution process of *namespace-or-type-name*s (Section 3.8) and *simple-name*s (Section 7.5.2), but unlike declarations, using directives do not contribute new members to the underlying declaration spaces of the compilation units or namespaces within which they are used.

using-directives:
  using-directive
  using-directives *using-directive*
using-directive:
  using-alias-directive
  using-namespace-directive

A *using-alias-directive* (Section 9.3.1) introduces an alias for a namespace or type.

A *using-namespace-directive* (Section 9.3.2) imports the type members of a namespace.

The scope of a *using-directive* extends over the *namespace-member-declarations* of its immediately containing compilation unit or namespace body. The scope of a *using-directive* specifically does not include its peer *using-directive*s. Thus, peer *using-directive*s do not affect each other, and the order in which they are written is insignificant.

9.3.1 Using alias directives

A *using-alias-directive* introduces an identifier that serves as an alias for a namespace or type within the immediately enclosing compilation unit or namespace body.

using-alias-directive:
  using *identifier* = *namespace-or-type-name* ;

Within member declarations in a compilation unit or namespace body that contains a *using-alias-directive*, the identifier introduced by the *using-alias-directive* can be used to reference the given namespace or type. For example:

```
namespace N1.N2
{
   class A {}
}
namespace N3
{
   using A = N1.N2.A;
   class B: A {}
}
```

Above, within member declarations in the `N3` namespace, `A` is an alias for `N1.N2.A`, and thus class `N3.B` derives from class `N1.N2.A`. The same effect can be obtained by creating an alias `R` for `N1.N2` and then referencing `R.A`:

```
namespace N3
{
   using R = N1.N2;
   class B: R.A {}
}
```

The *identifier* of a *using-alias-directive* must be unique within the declaration space of the compilation unit or namespace that immediately contains the *using-alias-directive*. For example:

```
namespace N3
{
   class A {}
}
namespace N3
{
   using A = N1.N2.A;      // Error, A already exists
}
```

Above, `N3` already contains a member `A`, so it is a compile-time error for a *using-alias-directive* to use that identifier. Likewise, it is a compile-time error for two or more *using-alias-directives* in the same compilation unit or namespace body to declare aliases by the same name.

A *using-alias-directive* makes an alias available within a particular compilation unit or namespace body, but it does not contribute any new members to the underlying declaration space. In other words, a *using-alias-directive* is not transitive but rather affects only the compilation unit or namespace body in which it occurs. In the example

```
namespace N3
{
   using R = N1.N2;
}
namespace N3
{
   class B: R.A {}         // Error, R unknown
}
```

the scope of the *using-alias-directive* that introduces `R` only extends to member declarations in the namespace body in which it is contained, so `R` is unknown in the second namespace declaration. However, placing the *using-alias-directive* in the containing compilation unit causes the alias to become available within both namespace declarations:

```
using R = N1.N2;
namespace N3
{
   class B: R.A {}
}
namespace N3
{
   class C: R.A {}
}
```

Just like regular members, names introduced by *using-alias-directive*s are hidden by similarly named members in nested scopes. In the example

```
using R = N1.N2;
namespace N3
{
   class R {}
   class B: R.A {}      // Error, R has no member A
}
```

the reference to `R.A` in the declaration of `B` causes a compile-time error because `R` refers to `N3.R`, not `N1.N2`.

The order in which *using-alias-directive*s are written has no significance, and resolution of the *namespace-or-type-name* referenced by a *using-alias-directive* is not affected by the *using-alias-directive* itself or by other *using-directive*s in the immediately containing compilation unit or namespace body. In other words, the *namespace-or-type-name* of a *using-alias-directive* is resolved as if the immediately containing compilation unit or namespace body had no *using-directive*s. In the example

```
namespace N1.N2 {}
namespace N3
{
   using R1 = N1;         // OK
   using R2 = N1.N2;      // OK
   using R3 = R1.N2;      // Error, R1 unknown
}
```

the last *using-alias-directive* results in a compile-time error because it is not affected by the first *using-alias-directive*.

A *using-alias-directive* can create an alias for any namespace or type, including the namespace within which it appears and any namespace or type nested within that namespace.

Accessing a namespace or type through an alias yields exactly the same result as accessing that namespace or type through its declared name. For example, given

```
namespace N1.N2
{
   class A {}
}
namespace N3
{
   using R1 = N1;
   using R2 = N1.N2;
   class B
   {
      N1.N2.A a;         // refers to N1.N2.A
      R1.N2.A b;         // refers to N1.N2.A
      R2.A c;            // refers to N1.N2.A
   }
}
```

the names `N1.N2.A`, `R1.N2.A`, and `R2.A` are equivalent and all refer to the class whose fully qualified name is `N1.N2.A`.

9.3.2 Using namespace directives

A *using-namespace-directive* imports the types contained in a namespace into the immediately enclosing compilation unit or namespace body, enabling the identifier of each type to be used without qualification.

using-namespace-directive:
  using *namespace-name* ;

Within member declarations in a compilation unit or namespace body that contains a *using-namespace-directive*, the types contained in the given namespace can be referenced directly. For example:

```
namespace N1.N2
{
   class A {}
}
namespace N3
{
   using N1.N2;
   class B: A {}
}
```

Above, within member declarations in the `N3` namespace, the type members of `N1.N2` are directly available, and thus class `N3.B` derives from class `N1.N2.A`.

A *using-namespace-directive* imports the types contained in the given namespace, but specifically does not import nested namespaces. In the example

```
namespace N1.N2
{
   class A {}
}
namespace N3
{
   using N1;
   class B: N2.A {}      // Error, N2 unknown
}
```

the *using-namespace-directive* imports the types contained in `N1`, but not the namespaces nested in `N1`. Thus, the reference to `N2.A` in the declaration of `B` results in a compile-time error because no members named `N2` are in scope.

Unlike a *using-alias-directive*, a *using-namespace-directive* may import types whose identifiers are already defined within the enclosing compilation unit or namespace body. In effect, names imported by a *using-namespace-directive* are hidden by similarly named members in the enclosing compilation unit or namespace body. For example:

```
namespace N1.N2
{
   class A {}
   class B {}
}
namespace N3
{
   using N1.N2;
   class A {}
}
```

Here, within member declarations in the `N3` namespace, `A` refers to `N3.A` rather than `N1.N2.A`.

When more than one namespace imported by *using-namespace-directive*s in the same compilation unit or namespace body contain types by the same name, references to that name are considered ambiguous. In the example

```
namespace N1
{
   class A {}
}
```

```
namespace N2
{
   class A {}
}
namespace N3
{
   using N1;
   using N2;
   class B: A {}            // Error, A is ambiguous
}
```

both `N1` and `N2` contain a member `A`, and because `N3` imports both, referencing `A` in `N3` is a compile-time error. In this situation, the conflict can be resolved either through qualification of references to `A`, or by introducing a *using-alias-directive* that picks a particular `A`. For example:

```
namespace N3
{
   using N1;
   using N2;
   using A = N1.A;
   class B: A {}            // A means N1.A
}
```

Like a *using-alias-directive*, a *using-namespace-directive* does not contribute any new members to the underlying declaration space of the compilation unit or namespace, but rather affects only the compilation unit or namespace body in which it appears.

The *namespace-name* referenced by a *using-namespace-directive* is resolved in the same way as the *namespace-or-type-name* referenced by a *using-alias-directive*. Thus, *using-namespace-directive*s in the same compilation unit or namespace body do not affect each other and can be written in any order.

9.4 Namespace members

A *namespace-member-declaration* is either a *namespace-declaration* (Section 9.2) or a *type-declaration* (Section 9.5).

namespace-member-declarations:
  namespace-member-declaration
  namespace-member-declarations namespace-member-declaration
namespace-member-declaration:
  namespace-declaration
  type-declaration

A compilation unit or a namespace body can contain *namespace-member-declarations*, and such declarations contribute new members to the underlying declaration space of the containing compilation unit or namespace body.

9.5 Type declarations

A *type-declaration* is a *class-declaration* (Section 10.1), a *struct-declaration* (Section 11.1), an *interface-declaration* (Section 13.1), an *enum-declaration* (Section 14.1), or a *delegate-declaration* (Section 15.1).

type-declaration:
  class-declaration
  struct-declaration
  interface-declaration
  enum-declaration
  delegate-declaration

A *type-declaration* can occur as a top-level declaration in a compilation unit or as a member declaration within a namespace, class, or struct.

When a type declaration for a type `T` occurs as a top-level declaration in a compilation unit, the fully qualified name of the newly declared type is simply `T`. When a type declaration for a type `T` occurs within a namespace, class, or struct, the fully qualified name of the newly declared type is `N.T`, where `N` is the fully qualified name of the containing namespace, class, or struct.

A type declared within a class or struct is called a nested type (Section 10.2.6).

The permitted access modifiers and the default access for a type declaration depend on the context in which the declaration takes place (Section 3.5.1):

- Types declared in compilation units or namespaces can have `public` or `internal` access. The default is `internal` access.
- Types declared in classes can have `public`, `protected internal`, `protected`, `internal`, or `private` access. The default is `private` access.
- Types declared in structs can have `public`, `internal`, or `private` access. The default is `private` access.

10. Classes

A class is a data structure that may contain data members (constants and fields), function members (methods, properties, events, indexers, operators, instance constructors, destructors and static constructors), and nested types. Class types support inheritance, a mechanism whereby a derived class can extend and specialize a base class.

10.1 Class declarations

A *class-declaration* is a *type-declaration* (Section 9.5) that declares a new class.

class-declaration:
  attributes$_{opt}$ *class-modifiers*$_{opt}$ class *identifier* *class-base*$_{opt}$ *class-body* ;$_{opt}$

A *class-declaration* consists of an optional set of *attributes* (Section 17), followed by an optional set of *class-modifiers* (Section 10.1.1), followed by the keyword `class` and an *identifier* that names the class, followed by an optional *class-base* specification (Section 10.1.2), followed by a *class-body* (Section 10.1.3), optionally followed by a semicolon.

10.1.1 Class modifiers

A *class-declaration* may optionally include a sequence of class modifiers:

class-modifiers:
  class-modifier
  class-modifiers *class-modifier*
class-modifier:
  new
  public
  protected
  internal
  private
  abstract
  sealed

It is a compile-time error for the same modifier to appear multiple times in a class declaration.

The `new` modifier is permitted on nested classes. It specifies that the class hides an inherited member by the same name, as described in Section 10.2.2. It is a compile-time error for the `new` modifier to appear on a class declaration that is not a nested class declaration.

The `public`, `protected`, `internal`, and `private` modifiers control the accessibility of the class. Depending on the context in which the class declaration occurs, some of these modifiers may not be permitted (Section 3.5.1).

The `abstract` and `sealed` modifiers are discussed in the following sections.

10.1.1.1 Abstract classes

The `abstract` modifier is used to indicate that a class is incomplete and that it is intended to be used only as a base class. An abstract class differs from a non-abstract class in the following ways:

- An abstract class cannot be instantiated directly, and it is a compile-time error to use the `new` operator on an abstract class. While it is possible to have variables and values whose compile-time types are abstract, such variables and values will necessarily either be `null` or contain references to instances of non-abstract classes derived from the abstract types.
- An abstract class is permitted (but not required) to contain abstract members.
- An abstract class cannot be sealed.

When a non-abstract class is derived from an abstract class, the non-abstract class must include actual implementations of all inherited abstract members, thereby overriding those abstract members. In the example

```
abstract class A
{
   public abstract void F();
}
abstract class B: A
{
   public void G() {}
}
class C: B
{
   public override void F() {
      // actual implementation of F
   }
}
```

the abstract class `A` introduces an abstract method `F`. Class `B` introduces an additional method `G`, but since it doesn't provide an implementation of `F`, `B` must also be declared abstract. Class `C` overrides `F` and provides an actual implementation. Since there are no abstract members in `C`, `C` is permitted (but not required) to be non-abstract.

10.1.1.2 Sealed classes

The `sealed` modifier is used to prevent derivation from a class. A compile-time error occurs if a sealed class is specified as the base class of another class.

A sealed class cannot also be an abstract class.

The `sealed` modifier is primarily used to prevent unintended derivation, but it also enables certain run-time optimizations. In particular, because a sealed class is known to never have any derived classes, it is possible to transform virtual function member invocations on sealed class instances into non-virtual invocations.

10.1.2 Class base specification

A class declaration may include a *class-base* specification, which defines the direct base class of the class and the interfaces (**Error! Reference source not found.**) implemented by the class.

class-base:
  : *class-type*
  : *interface-type-list*
  : *class-type* , *interface-type-list*
interface-type-list:
  interface-type
  interface-type-list , *interface-type*

10.1.2.1 Base classes

When a *class-type* is included in the *class-base*, it specifies the direct base class of the class being declared. If a class declaration has no *class-base*, or if the *class-base* lists only interface types, the direct base class is assumed to be `object`. A class inherits members from its direct base class, as described in Section 10.2.1.

In the example

```
class A {}
class B: A {}
```

class `A` is said to be the direct base class of `B`, and `B` is said to be derived from `A`. Since `A` does not explicitly specify a direct base class, its direct base class is implicitly `object`.

The direct base class of a class type must be at least as accessible as the class type itself (Section 3.5.4). For example, it is a compile-time error for a `public` class to derive from a `private` or `internal` class.

The direct base class of a class type must not be any of the following types: `System.Array`, `System.Delegate`, `System.Enum`, or `System.ValueType`.

The base classes of a class are the direct base class and its base classes. In other words, the set of base classes is the transitive closure of the direct base class relationship. Referring to the example above, the base classes of `B` are `A` and `object`.

Except for class `object`, every class has exactly one direct base class. The `object` class has no direct base class and is the ultimate base class of all other classes.

When a class `B` derives from a class `A`, it is a compile-time error for `A` to depend on `B`. A class *directly depends on* its direct base class (if any) and directly depends on the class within which it is immediately nested (if any). Given this definition, the complete set of classes upon which a class depends is the transitive closure of the *directly depends on* relationship.

The example

```
class A: B {}
class B: C {}
class C: A {}
```

is in error because the classes circularly depend on themselves. Likewise, the example

```
class A: B.C {}
class B: A
{
   public class C {}
}
```

results in a compile-time error because `A` depends on `B.C` (its direct base class), which depends on `B` (its immediately enclosing class), which circularly depends on `A`.

Note that a class does not depend on the classes that are nested within it. In the example

```
class A
{
   class B: A {}
}
```

`B` depends on `A` (because `A` is both its direct base class and its immediately enclosing class), but `A` does not depend on `B` (since `B` is neither a base class nor an enclosing class of `A`). Thus, the example is valid.

It is not possible to derive from a `sealed` class. In the example

```
sealed class A {}
class B: A {}         // Error, cannot derive from a sealed class
```

class `B` is in error because it attempts to derive from the `sealed` class `A`.

10.1.2.2 Interface implementations

A *class-base* specification may include a list of interface types, in which case the class is said to implement the given interface types. Interface implementations are discussed further in Section 13.4.

10.1.3 Class body

The *class-body* of a class defines the members of that class.

class-body:
  { *class-member-declarations*$_{opt}$ }

10.2 Class members

The members of a class consist of the members introduced by its *class-member-declaration*s and the members inherited from the direct base class.

class-member-declarations:
  class-member-declaration
  class-member-declarations class-member-declaration
class-member-declaration:
  constant-declaration
  field-declaration
  method-declaration
  property-declaration
  event-declaration
  indexer-declaration
  operator-declaration
  constructor-declaration
  destructor-declaration
  static-constructor-declaration
  type-declaration

The members of a class are divided into the following categories:

- Constants, which represent constant values associated with the class (Section 10.3).
- Fields, which are the variables of the class (Section 10.4).
- Methods, which implement the computations and actions that can be performed by the class (Section 10.5).
- Properties, which define named characteristics associated with reading and writing those characteristics (Section 10.6).
- Events, which define notifications that can be generated by the class (Section 10.7).
- Indexers, which permit instances of the class to be indexed in the same way (syntactically) as arrays (Section 10.8).
- Operators, which define the expression operators that can be applied to instances of the class (Section 10.9).
- Instance constructors, which implement the actions required to initialize instances of the class (Section 10.10)
- Destructors, which implement the actions to be performed before instances of the class are permanently discarded (Section 10.12).
- Static constructors, which implement the actions required to initialize the class itself (Section 10.11).
- Types, which represent the types that are local to the class (Section 9.5).

Members that can contain executable code are collectively known as the *function members* of the class 0. The function members of a class are the methods, properties, events, indexers, operators, instance constructors, destructors, and static constructors of that class.

A *class-declaration* creates a new declaration space (Section 3.3), and the *class-member-declarations* immediately contained by the *class-declaration* introduce new members into this declaration space. The following rules apply to *class-member-declarations*:

- Instance constructors, destructors and static constructors must have the same name as the immediately enclosing class. All other members must have names that differ from the name of the immediately enclosing class.
- The name of a constant, field, property, event, or type must differ from the names of all other members declared in the same class.
- The name of a method must differ from the names of all other non-methods declared in the same class. In addition, the signature (Section 3.6) of a method must differ from the signatures of all other methods declared in the same class.
- The signature of an instance constructor must differ from the signatures of all other instance constructors declared in the same class.
- The signature of an indexer must differ from the signatures of all other indexers declared in the same class.
- The signature of an operator must differ from the signatures of all other operators declared in the same class.

The inherited members of a class (Section 10.2.1) are not part of the declaration space of a class. Thus, a derived class is allowed to declare a member with the same name or signature as an inherited member (which in effect hides the inherited member).

10.2.1 Inheritance

A class inherits the members of its direct base class. Inheritance means that a class implicitly contains all members of its direct base class, except for the instance constructors, destructors and static constructors of the base class. Some important aspects of inheritance are:

- Inheritance is transitive. If `C` is derived from `B`, and `B` is derived from `A`, then `C` inherits the members declared in `B` as well as the members declared in `A`.
- A derived class *extends* its direct base class. A derived class can add new members to those it inherits, but it cannot remove the definition of an inherited member.
- Instance constructors, destructors, and static constructors are not inherited, but all other members are, regardless of their declared accessibility (Section 3.5). However, depending on their declared accessibility, inherited members might not be accessible in a derived class.
- A derived class can hide (Section 3.7.1.2) inherited members by declaring new members with the same name or signature. Note however that hiding an inherited member does not remove that member — it merely makes that member inaccessible directly through the derived class.
- An instance of a class contains a set of all instance fields declared in the class and its base classes, and an implicit conversion (Section 6.1.4) exists from a derived class type to any of its base class types. Thus, a reference to an instance of some derived class can be treated as a reference to an instance of any of its base classes.
- A class can declare virtual methods, properties, and indexers, and derived classes can override the implementation of these function members. This enables classes to exhibit polymorphic behavior wherein the actions performed by a function member invocation varies depending on the run-time type of the instance through which that function member is invoked.

10.2.2 The new modifier

A *class-member-declaration* is permitted to declare a member with the same name or signature as an inherited member. When this occurs, the derived class member is said to hide the base class member. Hiding an inherited member is not considered an error, but it does cause the compiler to issue a warning. To suppress the warning, the declaration of the derived class member can include a `new` modifier to indicate that the derived member is intended to hide the base member. This topic is discussed further in Section 3.7.1.2.

If a `new` modifier is included in a declaration that does not hide an inherited member, a warning to that effect is issued. This warning is suppressed by removing the `new` modifier.

10.2.3 Access modifiers

A *class-member-declaration* can have any one of the five possible kinds of declared accessibility (Section 3.5.1): `public`, `protected internal`, `protected`, `internal`, or `private`. Except for the `protected internal` combination, it is a compile-time error to specify more than one access modifier. When a *class-member-declaration* does not include any access modifiers, `private` is assumed.

10.2.4 Constituent types

Types that are used in the declaration of a member are called the constituent types of that member. Possible constituent types are the type of a constant, field, property, event, or indexer, the return type of a method or operator, and the parameter types of a method, indexer, operator, or instance constructor. The constituent types of a member must be at least as accessible as that member itself (Section 3.5.4).

10.2.5 Static and instance members

Members of a class are either static members or instance members. Generally speaking, it is useful to think of static members as belonging to classes and instance members as belonging to objects (instances of classes).

When a field, method, property, event, operator, or constructor declaration includes a `static` modifier, it declares a static member. In addition, a constant or type declaration implicitly declares a static member. Static members have the following characteristics:

- When a static member `M` is referenced in a *member-access* (Section 7.5.4) of the form `E.M`, `E` must denote a type containing `M`. It is a compile-time error for `E` to denote an instance.
- A static field identifies exactly one storage location. No matter how many instances of a class are created, there is only ever one copy of a static field.
- A static function member (method, property, event, operator, or constructor) does not operate on a specific instance, and it is a compile-time error to refer to `this` in such a function member.

When a field, method, property, event, indexer, constructor, or destructor declaration does not include a `static` modifier, it declares an instance member. (An instance member is sometimes called a non-static member.) Instance members have the following characteristics:

- When an instance member `M` is referenced in a *member-access* (Section 7.5.4) of the form `E.M`, `E` must denote an instance of a type containing `M`. It is a compile-time error for `E` to denote a type.
- Every instance of a class contains a separate set of all instance fields of the class.
- An instance function member (method, property, indexer, instance constructor, or destructor) operates on a given instance of the class, and this instance can be accessed as `this` (Section 7.5.7).

The following example illustrates the rules for accessing static and instance members:

```
class Test
{
   int x;
   static int y;
   void F() {
      x = 1;         // Ok, same as this.x = 1
      y = 1;         // Ok, same as Test.y = 1
   }
   static void G() {
      x = 1;         // Error, cannot access this.x
      y = 1;         // Ok, same as Test.y = 1
   }
   static void Main() {
      Test t = new Test();
      t.x = 1;         // Ok
      t.y = 1;         // Error, cannot access static member through instance
      Test.x = 1;      // Error, cannot access instance member through type
      Test.y = 1;      // Ok
   }
}
```

The `F` method shows that in an instance function member, a *simple-name* (Section 7.5.2) can be used to access both instance members and static members. The `G` method shows that in a static function member, it is a compile-time error to access an instance member through a *simple-name*. The `Main` method shows that in a *member-access* (Section 7.5.4), instance members must be accessed through instances, and static members must be accessed through types.

10.2.6 Nested types

A type declared within a class or struct is called a nested type. A type that is declared within a compilation unit or namespace is called a non-nested type.

In the example

```
using System;
class A
{
   class B
   {
      static void F() {
         Console.WriteLine("A.B.F");
      }
   }
}
```

class `B` is a nested type because it is declared within class `A`, and class `A` is a non-nested type because it is declared within a compilation unit.

10.2.6.1 Fully qualified name

The fully qualified name (Section 3.8.1) for a nested type is `S.N` where `S` is the fully qualified name of the type in which type `N` is declared.

10.2.6.2 Declared accessibility

Non-nested types can have `public` or `internal` declared accessibility and have `internal` declared accessibility by default. Nested types can have these forms of declared accessibility too, plus one or more additional forms of declared accessibility, depending on whether the containing type is a class or struct:

- A nested type that is declared in a class can have any of the five forms of declared accessibility (`public`, `protected internal`, `protected`, `internal`, or `private`) and, like other class members, defaults to `private` declared accessibility.
- A nested type that is declared in a struct can have any of three forms of declared accessibility (`public`, `internal`, or `private`) and, like other struct members, defaults to `private` declared accessibility.

The example

```
public class List
{
   // Private data structure
   private class Node
   {
      public object Data;
      public Node Next;
      public Node(object data, Node next) {
         this.Data = data;
         this.Next = next;
      }
   }
   private Node first = null;
   private Node last = null;
   // Public interface
   public void AddToFront(object o) {...}
   public void AddToBack(object o) {...}
   public object RemoveFromFront() {...}
   public object RemoveFromBack() {...}
   public int Count { get {...} }
}
```

declares a private nested class `Node`.

10.2.6.3 Hiding

A nested type may hide (Section 3.7.1) a base member. The `new` modifier is permitted on nested type declarations so that hiding can be expressed explicitly. The example

```
using System;
class Base
{
   public static void M() {
      Console.WriteLine("Base.M");
   }
}
class Derived: Base
{
   new public class M
   {
      public static void F() {
         Console.WriteLine("Derived.M.F");
      }
   }
}
class Test
{
   static void Main() {
      Derived.M.F();
   }
}
```

shows a nested class `M` that hides the method `M` defined in `Base`.

10.2.6.4 this access

A nested type and its containing type do not have a special relationship with regard to *this-access* (Section 7.5.7). Specifically, `this` within a nested type cannot be used to refer to instance members of the containing type. In cases where a nested type needs access to the instance members of its containing type, access can be provided by providing the `this` for the instance of the containing type as a constructor argument for the nested type. The following example

```
using System;
class C
{
   int i = 123;
   public void F() {
      Nested n = new Nested(this);
      n.G();
   }
   public class Nested {
      C this_c;
      public Nested(C c) {
         this_c = c;
      }
      public void G() {
         Console.WriteLine(this_c.i);
      }
   }
}
class Test {
   static void Main() {
      C c = new C();
      c.F();
   }
}
```

shows this technique. An instance of `C` creates an instance of `Nested` and passes its own this to `Nested`'s constructor in order to provide subsequent access to `C`'s instance members.

10.2.6.5 Access to private and protected members of the containing type

A nested type has access to all of the members that are accessible to its containing type, including members of the containing type that have `private` and `protected` declared accessibility. The example

```
using System;
class C
{
   private static void F() {
      Console.WriteLine("C.F");
   }
   public class Nested
   {
      public static void G() {
         F();
      }
   }
}
class Test
{
   static void Main() {
      C.Nested.G();
   }
}
```

shows a class `C` that contains a nested class `Nested`. Within `Nested`, the method `G` calls the static method `F` defined in `C`, and `F` has private declared accessibility.

A nested type also may access protected members defined in a base type of its containing type. In the example

```
using System;
class Base
{
   protected void F() {
      Console.WriteLine("Base.F");
   }
}
class Derived: Base
{
   public class Nested
   {
      public void G() {
         Derived d = new Derived();
         d.F();      // ok
      }
   }
}
class Test
{
   static void Main() {
      Derived.Nested n = new Derived.Nested();
      n.G();
   }
}
```

the nested class `Derived.Nested` accesses the protected method `F` defined in `Derived`'s base class, `Base`, by calling through an instance of `Derived`.

10.2.7 Reserved member names

To facilitate the underlying C# runtime implementation, for each source member declaration that is a property, event, or indexer, the implementation must reserve two method signatures based on the kind of the member declaration, its name, and its type. It is a compile-time error for a program to declare a member whose signature matches one of these reserved signatures, even if the underlying runtime implementation does not make use of these reservations.

The reserved names do not introduce declarations, thus they do not participate in member lookup. However, a declaration's associated reserved method signatures do participate in inheritance (Section 10.2.1), and can be hidden with the new modifier (Section 10.2.2).

The reservation of these names serves three purposes:

- To allow the underlying implementation to use an ordinary identifier as a method name for get or set access to the C# language feature.
- To allow other languages to interoperate using an ordinary identifier as a method name for get or set access to the C# language feature.
- To help ensure that the source accepted by one conforming compiler is accepted by another, by making the specifics of reserved member names consistent across all C# implementations.

The declaration of a destructor (Section 10.12) also causes a signature to be reserved (Section 10.2.7.4).

10.2.7.1 Member names reserved for properties

For a property `P` (Section 10.6) of type `T`, the following signatures are reserved:

```
T get_P();
void set_P(T value);
```

Both signatures are reserved, even if the property is read-only or write-only.

In the example

```
using System;
class A
{
   public int P {
      get { return 123; }
   }
}
class B: A
{
   new public int get_P() {
      return 456;
   }
   new public void set_P(int value) {
   }
}
class Test
{
   static void Main() {
      B b = new B();
      A a = b;
      Console.WriteLine(a.P);
      Console.WriteLine(b.P);
      Console.WriteLine(b.get_P());
   }
}
```

a class `A` defines a read-only property `P`, thus reserving signatures for `get_P` and `set_P` methods. A class `B` derives from `A` and hides both of these reserved signatures. The example produces the output:

```
123
123
456
```

10.2.7.2 Member names reserved for events

For an event `E` (Section 10.7) of delegate type `T`, the following signatures are reserved:

```
void add_E(T handler);
void remove_E(T handler);
```

10.2.7.3 Member names reserved for indexers

For an indexer (Section 10.8) of type `T` with parameter-list `L`, the following signatures are reserved:

```
T get_Item(L);
void set_Item(L, T value);
```

Both signatures are reserved, even if the indexer is read-only or write-only.

10.2.7.4 Member names reserved for destructors

For a class containing a destructor (Section 10.12), the following signature is reserved:

```
void Finalize();
```

10.3 Constants

A constant is a class member that represents a constant value: a value that can be computed at compile-time. A *constant-declaration* introduces one or more constants of a given type.

constant-declaration:
  attributes$_{opt}$ *constant-modifiers*$_{opt}$ const *type* *constant-declarators* ;
constant-modifiers:
  constant-modifier
  constant-modifiers *constant-modifier*
constant-modifier:
  new
  public
  protected
  internal
  private
constant-declarators:
  constant-declarator
  constant-declarators , *constant-declarator*
constant-declarator:
  identifier = *constant-expression*

A *constant-declaration* may include a set of *attributes* (Section 17), a `new` modifier (Section 10.2.2), and a valid combination of the four access modifiers (Section 10.2.3). The attributes and modifiers apply to all of the members declared by the *constant-declaration*. Even though constants are considered static members, a *constant-declaration* neither requires nor allows a `static` modifier. It is an error for the same modifier to appear multiple times in a constant declaration.

The *type* of a *constant-declaration* specifies the type of the members introduced by the declaration. The type is followed by a list of *constant-declarator*s, each of which introduces a new member. A *constant-declarator* consists of an *identifier* that names the member, followed by an "=" token, followed by a *constant-expression* (Section 7.15) that gives the value of the member.

The *type* specified in a constant declaration must be `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, `decimal`, `bool`, `string`, an *enum-type*, or a *reference-type*. Each *constant-expression* must yield a value of the target type or of a type that can be converted to the target type by an implicit conversion (Section 6.1).

The *type* of a constant must be at least as accessible as the constant itself (Section 3.5.4).

The value of a constant is obtained in an expression using a *simple-name* (Section 7.5.2) or a *member-access* (Section 7.5.4).

A constant can itself participate in a *constant-expression*. Thus, a constant may be used in any construct that requires a *constant-expression*. Examples of such constructs include `case` labels, `goto case` statements, `enum` member declarations, attributes, and other constant declarations.

As described in Section 7.15, a *constant-expression* is an expression that can be fully evaluated at compile-time. Since the only way to create a non-null value of a *reference-type* other than `string` is to apply the `new` operator, and since the `new` operator is not permitted in a *constant-expression*, the only possible value for constants of *reference-types* other than `string` is `null`.

When a symbolic name for a constant value is desired, but when the type of that value is not permitted in a constant declaration, or when the value cannot be computed at compile-time by a *constant-expression*, a `readonly` field (Section 10.4.2) may be used instead.

A constant declaration that declares multiple constants is equivalent to multiple declarations of single constants with the same attributes, modifiers, and type. For example

```
class A
{
   public const double X = 1.0, Y = 2.0, Z = 3.0;
}
```

is equivalent to

```
class A
{
   public const double X = 1.0;
   public const double Y = 2.0;
```

```
   public const double Z = 3.0;
}
```

Constants are permitted to depend on other constants within the same program as long as the dependencies are not of a circular nature. The compiler automatically arranges to evaluate the constant declarations in the appropriate order. In the example

```
class A
{
   public const int X = B.Z + 1;
   public const int Y = 10;
}
class B
{
   public const int Z = A.Y + 1;
}
```

the compiler first evaluates `A.Y`, then evaluates `B.Z`, and finally evaluates `A.X`, producing the values `10`, `11`, and `12`. Constant declarations may depend on constants from other programs, but such dependencies are only possible in one direction. Referring to the example above, if `A` and `B` were declared in separate programs, it would be possible for `A.X` to depend on `B.Z`, but `B.Z` could then not simultaneously depend on `A.Y`.

10.4 Fields

A field is a member that represents a variable associated with an object or class. A *field-declaration* introduces one or more fields of a given type.

field-declaration:
  attributes$_{opt}$ *field-modifiers*$_{opt}$ *type* *variable-declarators* ;
field-modifiers:
  field-modifier
  field-modifiers *field-modifier*
field-modifier:
  new
  public
  protected
  internal
  private
  static
  readonly
  volatile
variable-declarators:
  variable-declarator
  variable-declarators , *variable-declarator*
variable-declarator:
  identifier
  identifier = *variable-initializer*
variable-initializer:
  expression
  array-initializer

A *field-declaration* may include a set of *attributes* (Section 17), a `new` modifier (Section 10.2.2), a valid combination of the four access modifiers (Section 10.2.3), and a `static` modifier (Section 10.4.1). In addition, a *field-declaration* may include a `readonly` modifier (Section 10.4.2) or a `volatile` modifier (Section 10.4.3) but not both. The attributes and modifiers apply to all of the members declared by the *field-declaration*. It is an error for the same modifier to appear multiple times in a field declaration.

The *type* of a *field-declaration* specifies the type of the members introduced by the declaration. The type is followed by a list of *variable-declarators*, each of which introduces a new member. A *variable-declarator* consists of an *identifier* that names that member, optionally followed by an "=" token and a *variable-initializer* (Section 10.4.5) that gives the initial value of that member.

The *type* of a field must be at least as accessible as the field itself (Section 3.5.4).

The value of a field is obtained in an expression using a *simple-name* (Section 7.5.2) or a *member-access* (Section 7.5.4). The value of a non-readonly field is modified using an *assignment* (Section 7.13). The value of a non-readonly field can be both obtained and modified using postfix increment and decrement operators (Section 7.5.9) and prefix increment and decrement operators (Section 7.6.5).

A field declaration that declares multiple fields is equivalent to multiple declarations of single fields with the same attributes, modifiers, and type. For example

```
class A
{
   public static int X = 1, Y, Z = 100;
}
```

is equivalent to

```
class A
{
   public static int X = 1;
   public static int Y;
   public static int Z = 100;
}
```

10.4.1 Static and instance fields

When a *field declaration* includes a `static` modifier, the fields introduced by the declaration are static fields. When no `static` modifier is present, the fields introduced by the declaration are *instance fields*. Static fields and instance fields are two of the several kinds of variables (Section 5) supported by C#, and at times they are referred to as static variables and instance variables, respectively.

A static field is not part of a specific instance; instead, it identifies exactly one storage location. No matter how many instances of a class are created, there is only ever one copy of a static field for the associated application domain.

An instance field belongs to an instance. Specifically, every instance of a class contains a separate set of all the instance fields of that class.

When a field is referenced in a *member-access* (Section 7.5.4) of the form `E.M`, if `M` is a static field, `E` must denote a type containing `M`, and if `M` is an instance field, E must denote an instance of a type containing `M`.

The differences between static and instance members are discussed further in Section 10.2.5.

10.4.2 Readonly fields

When a *field-declaration* includes a `readonly` modifier, the fields introduced by the declaration are readonly fields. Direct assignments to readonly fields can only occur as part of that declaration or in an instance constructor or static constructor in the same class. (A readonly field can be assigned to multiple times in these contexts.) Specifically, direct assignments to a `readonly` field are permitted only in the following contexts:

- In the *variable-declarator* that introduces the field (by including a *variable-initializer* in the declaration).
- For an instance field, in the instance constructors of the class that contains the field declaration; for a static field, in the static constructor of the class that contains the field declaration. These are also the only contexts in which it is valid to pass a `readonly` field as an `out` or `ref` parameter.

Attempting to assign to a `readonly` field or pass it as an `out` or `ref` parameter in any other context is a compile-time error.

10.4.2.1 Using static readonly fields for constants

A `static readonly` field is useful when a symbolic name for a constant value is desired, but when the type of the value is not permitted in a `const` declaration, or when the value cannot be computed at compile-time. In the example

```
public class Color
{
   public static readonly Color Black = new Color(0, 0, 0);
   public static readonly Color White = new Color(255, 255, 255);
   public static readonly Color Red = new Color(255, 0, 0);
   public static readonly Color Green = new Color(0, 255, 0);
   public static readonly Color Blue = new Color(0, 0, 255);
   private byte red, green, blue;
   public Color(byte r, byte g, byte b) {
      red = r;
      green = g;
      blue = b;
   }
}
```

the `Black`, `White`, `Red`, `Green`, and `Blue` members cannot be declared as `const` members because their values cannot be computed at compile-time. However, declaring them `static readonly` instead has much the same effect.

10.4.2.2 Versioning of constants and static readonly fields

Constants and readonly fields have different binary versioning semantics. When an expression references a constant, the value of the constant is obtained at compile-time, but when an expression references a readonly field, the value of the field is not obtained until run-time. Consider an application that consists of two separate programs:

```
using System;
namespace Program1
{
   public class Utils
   {
      public static readonly int X = 1;
   }
}
namespace Program2
{
   class Test
   {
      static void Main() {
         Console.WriteLine(Program1.Utils.X);
      }
   }
}
```

The `Program1` and `Program2` namespaces denote two programs that are compiled separately. Because `Program1.Utils.X` is declared as a static readonly field, the value output by the `Console.WriteLine` statement is not known at compile-time, but rather is obtained at run-time. Thus, if the value of `X` is changed and `Program1` is recompiled, the `Console.WriteLine` statement will output the new value even if `Program2` is not recompiled. However, had `X` been a constant, the value of `X` would have been obtained at the time `Program2` was compiled, and would remain unaffected by changes in `Program1` until `Program2` is recompiled.

10.4.3 Volatile fields

When a *field-declaration* includes a `volatile` modifier, the fields introduced by that declaration are volatile fields.

For non-volatile fields, optimization techniques that reorder instructions can lead to unexpected and unpredictable results in multi-threaded programs that access fields without synchronization such as that provided by the *lock-statement* (Section 8.12). These optimizations can be performed by the compiler, by the runtime system, or by hardware. For volatile fields, such reordering optimizations are restricted:

- A read of a volatile field is called a volatile read. A volatile read has "acquire semantics"; that is, it is guaranteed to occur prior to any references to memory that occur after it in the instruction sequence.
- A write of a volatile field is called a volatile write. A volatile write has "release semantics"; that is, it is guaranteed to happen after any memory references prior to the write instruction in the instruction sequence.

These restrictions ensure that all threads will observe volatile writes performed by any other thread in the order in which they were performed. A conforming implementation is not required to provide a single total ordering of volatile writes as seen from all threads of execution. The type of a volatile field must be one of the following:

- A *reference-type*.
- The type `byte`, `sbyte`, `short`, `ushort`, `int`, `uint`, `char`, `float`, or `bool`.
- An *enum-type* having an enum base type of `byte`, `sbyte`, `short`, `ushort`, `int`, or `uint`.

The example

```
using System;
using System.Threading;
class Test
{
   public static int result;
   public static volatile bool finished;
   static void Thread2() {
      result = 143;
      finished = true;
   }
   static void Main() {
      finished = false;
      // Run Thread2() in a new thread
      new Thread(new ThreadStart(Thread2)).Start();
      // Wait for Thread2 to signal that it has a result by setting
      // finished to true.
      for (;;) {
         if (finished) {
            Console.WriteLine("result = {0}", result);
            return;
         }
      }
   }
}
```

produces the output:

```
result = 143
```

In this example, the method `Main` starts a new thread running the method `Thread2`. This method stores a value into a non-volatile field called `result`, then stores `true` in the volatile field `finished`. The main thread waits for the field `finished` to be set to `true`, then reads the field `result`. Since `finished` has been declared `volatile`, the main thread must read the value `143` from the field `result`. If the field `finished` had not been declared `volatile`, then it would be permissible for the store to `result` to be visible to the main thread *after* the store to `finished`, and hence for the main thread to read the value `0` from the field `result`. Declaring `finished` as a `volatile` field prevents any such inconsistency.

10.4.4 Field initialization

The initial value of a field, whether it be a static field or an instance field, is the default value (Section 5.2) of the field's type. It is not possible to observe the value of a field before this default initialization has occurred, and a field is thus never "uninitialized". The example

```
using System;
class Test
{
   static bool b;
   int i;
   static void Main() {
      Test t = new Test();
      Console.WriteLine("b = {0}, i = {1}", b, t.i);
   }
}
```

produces the output

```
b = False, i = 0
```

because `b` and `i` are both automatically initialized to default values.

10.4.5 Variable initializers

Field declarations may include *variable-initializer*s. For static fields, variable initializers correspond to assignment statements that are executed during class initialization. For instance fields, variable initializers correspond to assignment statements that are executed when an instance of the class is created.

The example

```
using System;
class Test
{
   static double x = Math.Sqrt(2.0);
   int i = 100;
   string s = "Hello";
   static void Main() {
      Test a = new Test();
      Console.WriteLine("x = {0}, i = {1}, s = {2}", x, a.i, a.s);
   }
}
```

produces the output

```
x = 1.4142135623731, i = 100, s = Hello
```

because an assignment to `x` occurs when static field initializers execute and assignments to `i` and `s` occur when the instance field initializers execute.

The default value initialization described in Section 10.4.4 occurs for all fields, including fields that have variable initializers. Thus, when a class is initialized, all static fields in that class are first initialized to their default values, and then the static field initializers are executed in textual order. Likewise, when an instance of a class is created, all instance fields in that instance are first initialized to their default values, and then the instance field initializers are executed in textual order.

It is possible for static fields with variable initializers to be observed in their default value state. However, this is strongly discouraged as a matter of style. The example

```
using System;
class Test
{
   static int a = b + 1;
   static int b = a + 1;
   static void Main() {
      Console.WriteLine("a = {0}, b = {1}", a, b);
   }
}
```

exhibits this behavior. Despite the circular definitions of a and b, the program is valid. It results in the output

```
a = 1, b = 2
```

because the static fields `a` and `b` are initialized to `0` (the default value for `int`) before their initializers are executed. When the initializer for `a` runs, the value of `b` is zero, and so `a` is initialized to `1`. When the initializer for `b` runs, the value of `a` is already `1`, and so `b` is initialized to `2`.

10.4.5.1 Static field initialization

The static field variable initializers of a class correspond to a sequence of assignments that are executed in the textual order in which they appear in the class declaration. If a static constructor (Section 10.11) exists in the class, execution of the static field initializers occurs immediately prior to executing that static constructor. Otherwise, the static field initializers are executed at an implementation-dependent time prior to the first use of a static field of that class. The example

```
using System;
class Test
{
   static void Main() {
      Console.WriteLine("{0} {1}", B.Y, A.X);
   }
   public static int F(string s) {
      Console.WriteLine(s);
      return 1;
   }
}
class A
{
   public static int X = Test.F("Init A");
}
class B
{
   public static int Y = Test.F("Init B");
}
```

might produce either the output:

```
Init A
Init B
1 1
```

or the output:

```
Init B
Init A
1 1
```

because the execution of `X`'s initializer and `Y`'s initializer could occur in either order; they are only constrained to occur before the references to those fields. However, in the example:

```
using System;
class Test
{
   static void Main() {
      Console.WriteLine("{0} {1}", B.Y, A.X);
   }
   public static int F(string s) {
      Console.WriteLine(s);
      return 1;
   }
}
class A
{
   static A() {}
   public static int X = Test.F("Init A");
}
class B
{
   static B() {}
   public static int Y = Test.F("Init B");
}
```

the output must be:

```
Init B
Init A
1 1
```

because the rules for when static constructors execute (as defined in Section 10.11) provide that `B`'s static constructor (and hence `B`'s static field initializers) must run before `A`'s static constructor and field initializers.

10.4.5.2 Instance field initialization

The instance field variable initializers of a class correspond to a sequence of assignments that are executed immediately upon entry to any one of the instance constructors (Section 10.10.1) of that class. The variable initializers are executed in the textual order in which they appear in the class declaration. The class instance creation and initialization process is described further in Section 10.10.

A variable initializer for an instance field cannot reference the instance being created. Thus, it is a compile-time error to reference `this` in a variable initializer, as it is a compile-time error for a variable initializer to reference any instance member through a *simple-name*. In the example

```
class A
{
   int x = 1;
   int y = x + 1;      // Error, reference to instance member of this
}
```

the variable initializer for `y` results in a compile-time error because it references a member of the instance being created.

10.5 Methods

A method is a member that implements a computation or action that can be performed by an object or class. Methods are declared using *method-declarations*:

method-declaration:
  method-header *method-body*
method-header:
  attributes$_{opt}$ *method-modifiers*$_{opt}$ *return-type* *member-name* (*formal-parameter-list*$_{opt}$)
method-modifiers:
  method-modifier
  method-modifiers *method-modifier*
method-modifier:
  new
  public
  protected
  internal
  private
  static
  virtual
  sealed
  override
  abstract
  extern
return-type:
  type
  void
member-name:
  identifier
  interface-type . *identifier*
method-body:
  block
  ;

A *method-declaration* may include a set of *attributes* (Section 17) and a valid combination of the four access modifiers (Section 10.2.3), the `new` (Section 10.2.2), `static` (Section 10.5.2), `virtual` (Section 10.5.3), `override` (Section 10.5.4), `sealed` (Section 10.5.5), `abstract` (Section 10.5.6), and `extern` (Section 10.5.7) modifiers.

A declaration has a valid combination of modifiers if all of the following are true:

- The declaration includes a valid combination of access modifiers (Section 10.2.3).
- The declaration does not include the same modifier multiple times.
- The declaration includes at most one of the following modifiers: `static`, `virtual`, and `override`.
- The declaration includes at most one of the following modifiers: `new` and `override`.
- If the declaration includes the `abstract` modifier, then the declaration does not include any of the following modifiers: `static`, `virtual`, `sealed`, or `extern`.
- If the declaration includes the `private` modifier, then the declaration does not include any of the following modifiers: `virtual`, `override`, or `abstract`.
- If the declaration includes the `sealed` modifier, then the declaration also includes the `override` modifier.

The *return-type* of a method declaration specifies the type of the value computed and returned by the method. The *return-type* is `void` if the method does not return a value.

The *member-name* specifies the name of the method. Unless the method is an explicit interface member implementation (Section 13.4.1), the *member-name* is simply an *identifier*. For an explicit interface member implementation, the *member-name* consists of an *interface-type* followed by a "`.`" and an *identifier*.

The optional *formal-parameter-list* specifies the parameters of the method (Section 10.5.1).

The *return-type* and each of the types referenced in the *formal-parameter-list* of a method must be at least as accessible as the method itself (Section 3.5.4).

For `abstract` and `extern` methods, the *method-body* consists simply of a semicolon. For all other methods, the *method-body* consists of a *block*, which specifies the statements to execute when the method is invoked.

The name and the formal parameter list of a method define the signature (Section 3.6) of the method. Specifically, the signature of a method consists of its name and the number, modifiers, and types of its formal parameters. The return type is not part of a method's signature, nor are the names of the formal parameters.

The name of a method must differ from the names of all other non-methods declared in the same class. In addition, the signature of a method must differ from the signatures of all other methods declared in the same class.

10.5.1 Method parameters

The parameters of a method, if any, are declared by the method's *formal-parameter-list*.

formal-parameter-list:
  fixed-parameters
  fixed-parameters , *parameter-array*
  parameter-array
fixed-parameters:
  fixed-parameter
  fixed-parameters , *fixed-parameter*
fixed-parameter:
  attributes$_{opt}$ *parameter-modifier*$_{opt}$ *type* *identifier*
parameter-modifier:
  ref
  out
parameter-array:
  attributes$_{opt}$ params *array-type* *identifier*

The formal parameter list consists of one or more comma-separated parameters of which only the last may be a *parameter-array*.

A *fixed-parameter* consists of an optional set of *attributes* (Section 17), an optional `ref` or `out` modifier, a *type*, and an *identifier*. Each *fixed-parameter* declares a parameter of the given type with the given name.

A *parameter-array* consists of an optional set of *attributes* (Section 17), a `params` modifier, an *array-type*, and an *identifier*. A parameter array declares a single parameter of the given array type with the given name. The *array-type* of a parameter array must be a single-dimensional array type (Section 12.1). In a method invocation, a parameter array permits either a single argument of the given array type to be specified, or it permits zero or more arguments of the array element type to be specified. Parameter arrays are further described in Section 10.5.1.4.

A method declaration creates a separate declaration space for parameters and local variables. Names are introduced into this declaration space by the formal parameter list of the method and by local variable declarations in the *block* of the method. All names in the declaration space of a method must be unique. Thus, it is a compile-time error for a parameter or local variable to have the same name as another parameter or local variable.

A method invocation (Section 7.5.5.1) creates a copy, specific to that invocation, of the formal parameters and local variables of the method, and the argument list of the invocation assigns values or variable references to the newly created formal parameters. Within the *block* of a method, formal parameters can be referenced by their identifiers in *simple-name* expressions (Section 7.5.2).

There are four kinds of formal parameters:

- Value parameters, which are declared without any modifiers.
- Reference parameters, which are declared with the `ref` modifier.
- Output parameters, which are declared with the `out` modifier.
- Parameter arrays, which are declared with the `params` modifier.

As described in Section 3.6, the `ref` and `out` modifiers are part of a method's signature, but the `params` modifier is not.

10.5.1.1 Value parameters

A parameter declared with no modifiers is a value parameter. A value parameter corresponds to a local variable that gets its initial value from the corresponding argument supplied in the method invocation.

When a formal parameter is a value parameter, the corresponding argument in a method invocation must be an expression of a type that is implicitly convertible (Section 6.1) to the formal parameter type.

A method is permitted to assign new values to a value parameter. Such assignments only affect the local storage location represented by the value parameter — they have no effect on the actual argument given in the method invocation.

10.5.1.2 Reference parameters

A parameter declared with a `ref` modifier is a reference parameter. Unlike a value parameter, a reference parameter does not create a new storage location. Instead, a reference parameter represents the same storage location as the variable given as the argument in the method invocation.

When a formal parameter is a reference parameter, the corresponding argument in a method invocation must consist of the keyword `ref` followed by a *variable-reference* (Section 5.4) of the same type as the formal parameter. A variable must be definitely assigned before it can be passed as a reference parameter.

Within a method, a reference parameter is always considered definitely assigned.

The example

```
using System;
class Test
{
   static void Swap(ref int x, ref int y) {
      int temp = x;
      x = y;
      y = temp;
   }
   static void Main() {
      int i = 1, j = 2;
      Swap(ref i, ref j);
      Console.WriteLine("i = {0}, j = {1}", i, j);
   }
}
```

produces the output

```
i = 2, j = 1
```

For the invocation of `Swap` in `Main`, `x` represents `i` and `y` represents `j`. Thus, the invocation has the effect of swapping the values of `i` and `j`.

In a method that takes reference parameters it is possible for multiple names to represent the same storage location. In the example

```
class A
{
   string s;
   void F(ref string a, ref string b) {
      s = "One";
      a = "Two";
      b = "Three";
   }
   void G() {
      F(ref s, ref s);
   }
}
```

the invocation of `F` in `G` passes a reference to `s` for both `a` and `b`. Thus, for that invocation, the names `s`, `a`, and `b` all refer to the same storage location, and the three assignments all modify the instance field `s`.

10.5.1.3 Output parameters

A parameter declared with an `out` modifier is an output parameter. Similar to a reference parameter, an output parameter does not create a new storage location. Instead, an output parameter represents the same storage location as the variable given as the argument in the method invocation.

When a formal parameter is an output parameter, the corresponding argument in a method invocation must consist of the keyword `out` followed by a *variable-reference* (Section 5.4) of the same type as the formal parameter. A variable need not be definitely assigned before it can be passed as an output parameter, but following an invocation where a variable was passed as an output parameter, the variable is considered definitely assigned.

Within a method, just like a local variable, an output parameter is initially considered unassigned and must be definitely assigned before its value is used.

Every output parameter of a method must be definitely assigned before the method returns.

Output parameters are typically used in methods that produce multiple return values. For example:

```
using System;
class Test
{
   static void SplitPath(string path, out string dir, out string name) {
      int i = path.Length;
      while (i > 0) {
         char ch = path[i - 1];
         if (ch == '\\' || ch == '/' || ch == ':') break;
         i--;
      }
      dir = path.Substring(0, i);
      name = path.Substring(i);
   }
   static void Main() {
      string dir, name;
      SplitPath("c:\\Windows\\System\\hello.txt", out dir, out name);
      Console.WriteLine(dir);
      Console.WriteLine(name);
   }
}
```

The example produces the output:

```
c:\Windows\System\
hello.txt
```

Note that the `dir` and `name` variables can be unassigned before they are passed to `SplitPath`, and that they are considered definitely assigned following the call.

10.5.1.4 Parameter arrays

A parameter declared with a `params` modifier is a parameter array. If a formal parameter list includes a parameter array, it must be the last parameter in the list and it must be of a single-dimensional array type. For example, the types `string[]` and `string[][]` can be used as the type of a parameter array, but the type `string[,]` cannot. It is not possible to combine the `params` modifier with the modifiers `ref` and `out`.

A parameter array permits arguments to be specified in one of two ways in a method invocation:

- The argument given for a parameter array can be a single expression of a type that is implicitly convertible (Section 6.1) to the parameter array type. In this case, the parameter array acts precisely like a value parameter.
- Alternatively, the invocation can specify zero or more arguments for the parameter array, where each argument is an expression of a type that is implicitly convertible (Section 6.1) to the element type of the parameter array. In this case, the invocation creates an instance of the parameter array type with a length corresponding to the number of arguments, initializes the elements of the array instance with the given argument values, and uses the newly created array instance as the actual argument.

Except for allowing a variable number of arguments in an invocation, a parameter array is precisely equivalent to a value parameter (Section 10.5.1.1) of the same type.

The example

```
using System;
class Test
{
   static void F(params int[] args) {
      Console.Write("Array contains {0} elements:", args.Length);
      foreach (int i in args)
         Console.Write(" {0}", i);
      Console.WriteLine();
   }
   static void Main() {
      int[] arr = {1, 2, 3};
      F(arr);
      F(10, 20, 30, 40);
      F();
   }
}
```

produces the output

```
Array contains 3 elements: 1 2 3
Array contains 4 elements: 10 20 30 40
Array contains 0 elements:
```

The first invocation of `F` simply passes the array `a` as a value parameter. The second invocation of `F` automatically creates a four-element `int[]` with the given element values and passes that array instance as a value parameter. Likewise, the third invocation of `F` creates a zero-element `int[]` and passes that instance as a value parameter. The second and third invocations are precisely equivalent to writing:

```
F(new int[] {10, 20, 30, 40});
F(new int[] {});
```

When performing overload resolution, a method with a parameter array may be applicable either in its normal form or in its expanded form (Section 7.4.2.1). The expanded form of a method is available only if the normal form of the method is not applicable and only if a method with the same signature as the expanded form is not already declared in the same type.

The example

```
using System;
class Test
{
```

```
   static void F(params object[] a) {
      Console.WriteLine("F(object[])");
   }
   static void F() {
      Console.WriteLine("F()");
   }
   static void F(object a0, object a1) {
      Console.WriteLine("F(object,object)");
   }
   static void Main() {
      F();
      F(1);
      F(1, 2);
      F(1, 2, 3);
      F(1, 2, 3, 4);
   }
}
```

produces the output

```
F();
F(object[]);
F(object,object);
F(object[]);
F(object[]);
```

In the example, two of the possible expanded forms of the method with a parameter array are already included in the class as regular methods. These expanded forms are therefore not considered when performing overload resolution, and the first and third method invocations thus select the regular methods. When a class declares a method with a parameter array, it is not uncommon to also include some of the expanded forms as regular methods. By doing so it is possible to avoid the allocation of an array instance that occurs when an expanded form of a method with a parameter array is invoked.

When the type of a parameter array is `object[]`, a potential ambiguity arises between the normal form of the method and the expended form for a single `object` parameter. The reason for the ambiguity is that an `object[]` is itself implicitly convertible to type `object`. The ambiguity presents no problem, however, since it can be resolved by inserting a cast if needed.

The example

```
using System;
class Test
{
   static void F(params object[] args) {
      foreach (object o in args) {
         Console.Write(o.GetType().FullName);
         Console.Write(" ");
      }
      Console.WriteLine();
   }
   static void Main() {
      object[] a = {1, "Hello", 123.456};
      object o = a;
      F(a);
      F((object)a);
      F(o);
      F((object[])o);
   }
}
```

produces the output

```
System.Int32 System.String System.Double
System.Object[]
System.Object[]
System.Int32 System.String System.Double
```

In the first and last invocations of `F`, the normal form of `F` is applicable because an implicit conversion exists from the argument type to the parameter type (both are of type `object[]`). Thus, overload resolution selects the normal form of `F`, and the argument is passed as a regular value parameter. In the second and third invocations, the normal form of `F` is not applicable because no implicit conversion exists from the argument type to the parameter type (type `object` cannot be implicitly converted to type `object[]`). However, the expanded form of `F` is applicable, so it is selected by overload resolution. As a result, a one-element `object[]` is created by the invocation, and the single element of the array is initialized with the given argument value (which itself is a reference to an `object[]`).

10.5.2 Static and instance methods

When a method declaration includes a `static` modifier, that method is said to be a static method. When no `static` modifier is present, the method is said to be an instance method.

A static method does not operate on a specific instance, and it is a compile-time error to refer to `this` in a static method.

An instance method operates on a given instance of a class, and that instance can be accessed as `this` (Section 7.5.7).

When a method is referenced in a *member-access* (Section 7.5.4) of the form `E.M`, if `M` is a static method, `E` must denote a type containing `M`, and if `M` is an instance method, `E` must denote an instance of a type containing `M`.

The differences between static and instance members are discussed further in Section 10.2.5.

10.5.3 Virtual methods

When an instance method declaration includes a `virtual` modifier, that method is said to be a virtual method. When no `virtual` modifier is present, the method is said to be a non-virtual method.

The implementation of a non-virtual method is invariant: The implementation is the same whether the method is invoked on an instance of the class in which it is declared or an instance of a derived class. In contrast, the implementation of a virtual method can be superseded by derived classes. The process of superseding the implementation of an inherited virtual method is known as overriding that method (Section 10.5.4).

In a virtual method invocation, the run-time type of the instance for which that invocation takes place determines the actual method implementation to invoke. In a non-virtual method invocation, the compile-time type of the instance is the determining factor. In precise terms, when a method named `N` is invoked with an argument list `A` on an instance with a compile-time type `C` and a run-time type `R` (where `R` is either `C` or a class derived from `C`), the invocation is processed as follows:

- First, overload resolution is applied to `C`, `N`, and `A`, to select a specific method `M` from the set of methods declared in and inherited by `C`. This is described in Section 7.5.5.1.
- Then, if `M` is a non-virtual method, `M` is invoked.
- Otherwise, `M` is a virtual method, and the most derived implementation of `M` with respect to `R` is invoked.

For every virtual method declared in or inherited by a class, there exists a most derived implementation of the method with respect to that class. The most derived implementation of a virtual method `M` with respect to a class `R` is determined as follows:

- If `R` contains the introducing `virtual` declaration of `M`, then this is the most derived implementation of `M`.
- Otherwise, if `R` contains an `override` of `M`, then this is the most derived implementation of `M`.
- Otherwise, the most derived implementation of `M` with respect to `R` is the same as the most derived implementation of `M` with respect to the direct base class of `R`.

The following example illustrates the differences between virtual and non-virtual methods:

```
using System;
class A
{
   public void F() { Console.WriteLine("A.F"); }
   public virtual void G() { Console.WriteLine("A.G"); }
}
class B: A
{
   new public void F() { Console.WriteLine("B.F"); }
   public override void G() { Console.WriteLine("B.G"); }
}
class Test
{
   static void Main() {
      B b = new B();
      A a = b;
      a.F();
      b.F();
      a.G();
      b.G();
   }
}
```

In the example, `A` introduces a non-virtual method `F` and a virtual method `G`. The class `B` introduces a *new* non-virtual method `F`, thus *hiding* the inherited `F`, and also *overrides* the inherited method `G`. The example produces the output:

```
A.F
B.F
B.G
B.G
```

Notice that the statement `a.G()` invokes `B.G`, not `A.G`. This is because the run-time type of the instance (which is `B`), not the

compile-time type of the instance (which is `A`), determines the actual method implementation to invoke.

Because methods are allowed to hide inherited methods, it is possible for a class to contain several virtual methods with the same signature. This does not present an ambiguity problem, since all but the most derived method are hidden. In the example

```
using System;
class A
{
   public virtual void F() { Console.WriteLine("A.F"); }
}
class B: A
{
   public override void F() { Console.WriteLine("B.F"); }
}
class C: B
{
   new public virtual void F() { Console.WriteLine("C.F"); }
}
class D: C
{
   public override void F() { Console.WriteLine("D.F"); }
}
class Test
{
   static void Main() {
      D d = new D();
      A a = d;
      B b = d;
      C c = d;
      a.F();
      b.F();
      c.F();
      d.F();
   }
}
```

the `C` and `D` classes contain two virtual methods with the same signature: The one introduced by `A` and the one introduced by `C`. The method introduced by `C` hides the method inherited from `A`. Thus, the override declaration in `D` overrides the method introduced by `C`, and it is not possible for `D` to override the method introduced by `A`. The example produces the output:

```
B.F
B.F
D.F
D.F
```

Note that it is possible to invoke the hidden virtual method by accessing an instance of `D` through a less derived type in which the method is not hidden.

10.5.4 Override methods

When an instance method declaration includes an `override` modifier, the method is said to be an override method. An override method overrides an inherited virtual method with the same signature. Whereas a virtual method declaration *introduces* a new method, an override method declaration *specializes* an existing inherited virtual method by providing a new implementation of that method.

The method overridden by an `override` declaration is known as the overridden base method. For an override method `M` declared in a class `C`, the overridden base method is determined by examining each base class of `C`, starting with the direct base class of `C` and continuing with each successive direct base class, until an accessible method with the same signature as `M` is located. For the purposes of locating the overridden base method, a method is considered accessible if it is `public`, if it is `protected`, if it is `protected internal`, or if it is `internal` and declared in the same program as `C`.

A compile-time error occurs unless all of the following are true for an override declaration:

- An overridden base method can be located as described above.
- The overridden base method is a virtual, abstract, or override method. In other words, the overridden base method cannot be static or non-virtual.
- The overridden base method is not a sealed method.
- The override declaration and the overridden base method have the same return type.
- The override declaration and the overridden base method have the same declared accessibility. In other words, an override declaration cannot change the accessibility of the virtual method.

An override declaration can access the overridden base method using a *base-access* (Section 7.5.8). In the example

```
class A
{
   int x;
   public virtual void PrintFields() {
      Console.WriteLine("x = {0}", x);
   }
}
class B: A
{
   int y;
   public override void PrintFields() {
      base.PrintFields();
      Console.WriteLine("y = {0}", y);
   }
}
```

the `base.PrintFields()` invocation in `B` invokes the `PrintFields` method declared in `A`. A *base-access* disables the virtual invocation mechanism and simply treats the base method as a non-virtual method. Had the invocation in `B` been written `((A)this).PrintFields()`, it would recursively invoke the `PrintFields` method declared in `B`, not the one declared in `A`, since `PrintFields` is virtual and the run-time type of `((A)this)` is `B`.

Only by including an `override` modifier can a method override another method. In all other cases, a method with the same signature as an inherited method simply hides the inherited method. In the example

```
class A
{
   public virtual void F() {}
}
class B: A
{
   public virtual void F() {}      // Warning, hiding inherited F()
}
```

the `F` method in `B` does not include an `override` modifier and therefore does not override the `F` method in `A`. Rather, the `F` method in `B` hides the method in `A`, and a warning is reported because the declaration does not include a `new` modifier.

In the example

```
class A
{
   public virtual void F() {}
}
class B: A
{
   new private void F() {}         // Hides A.F within B
}
class C: B
{
   public override void F() {}   // Ok, overrides A.F
}
```

the `F` method in `B` hides the virtual `F` method inherited from `A`. Since the new `F` in `B` has private access, its scope only includes the class body of `B` and does not extend to `C`. Therefore, the declaration of `F` in `C` is permitted to override the `F` inherited from `A`.

10.5.5 Sealed methods

When an instance method declaration includes a `sealed` modifier, that method is said to be a sealed method. If an instance method declaration includes the `sealed` modifier, it must also include the `override` modifier. Use of the `sealed` modifier prevents a derived class from further overriding the method.

The example

```
using System;
class A
{
   public virtual void F() {
      Console.WriteLine("A.F");
   }
   public virtual void G() {
      Console.WriteLine("A.G");
   }
}
class B: A
{
   sealed override public void F() {
      Console.WriteLine("B.F");
   }
   override public void G() {
      Console.WriteLine("B.G");
   }
}
class C: B
{
   override public void G() {
      Console.WriteLine("C.G");
   }
}
```

the class `B` provides two override methods: an `F` method that has the `sealed` modifier and a `G` method that does not. `B`'s use of the sealed `modifier` prevents `C` from further overriding `F`.

10.5.6 Abstract methods

When an instance method declaration includes an `abstract` modifier, that method is said to be an abstract method. Although an abstract method is implicitly also a virtual method, it cannot have the modifier `virtual`.

An abstract method declaration introduces a new virtual method but does not provide an implementation of that method. Instead, non-abstract derived classes are required to provide their own implementation by overriding that method. Because an abstract method provides no actual implementation, the *method-body* of an abstract method simply consists of a semicolon.

Abstract method declarations are only permitted in abstract classes (Section 10.1.1.1).

In the example

```
public abstract class Shape
{
   public abstract void Paint(Graphics g, Rectangle r);
}
public class Ellipse: Shape
{
   public override void Paint(Graphics g, Rectangle r) {
      g.DrawEllipse(r);
   }
}
public class Box: Shape
{
   public override void Paint(Graphics g, Rectangle r) {
      g.DrawRect(r);
   }
}
```

the `Shape` class defines the abstract notion of a geometrical shape object that can paint itself. The `Paint` method is abstract because there is no meaningful default implementation. The `Ellipse` and `Box` classes are concrete `Shape` implementations. Because these classes are non-abstract, they are required to override the `Paint` method and provide an actual implementation.

It is a compile-time error for a *base-access* (Section 7.5.8) to reference an abstract method. In the example

```
abstract class A
{
   public abstract void F();
}
class B: A
{
   public override void F() {
      base.F();                  // Error, base.F is abstract
   }
}
```

a compile-time error is reported for the `base.F()` invocation because it references an abstract method.

An abstract method declaration is permitted to override a virtual method. This allows an abstract class to force re-implementation of the method in derived classes, and makes the original implementation of the method unavailable. In the example

```
using System;
class A
{
   public virtual void F() {
      Console.WriteLine("A.F");
   }
}
abstract class B: A
{
   public abstract override void F();
}
class C: B
{
```

```
    public override void F() {
       Console.WriteLine("C.F");
    }
 }
```

class `A` declares a virtual method, class `B` overrides this method with an abstract method, and class `C` overrides the abstract method to provide its own implementation.

10.5.7 External methods

When a method declaration includes an `extern` modifier, that method is said to be an external method. External methods are implemented externally, typically using a language other than C#. Because an external method declaration provides no actual implementation, the *method-body* of an external method simply consists of a semicolon.

The `extern` modifier is typically used in conjunction with a `DllImport` attribute (Section 10.5.1), allowing external methods to be implemented by DLLs (Dynamic Link Libraries). The execution environment may support other mechanisms whereby implementations of external methods can be provided.

When an external method includes a `DllImport` attribute, the method declaration must also include a `static` modifier. This example demonstrates the use of the `extern` modifier and the `DllImport` attribute:

```
using System.Text;
using System.Security.Permissions;
using System.Runtime.InteropServices;
class Path
{
   [DllImport("kernel32", SetLastError=true)]
   static extern bool CreateDirectory(string name, SecurityAttribute sa);
   [DllImport("kernel32", SetLastError=true)]
   static extern bool RemoveDirectory(string name);
   [DllImport("kernel32", SetLastError=true)]
   static extern int GetCurrentDirectory(int bufSize, StringBuilder buf);
   [DllImport("kernel32", SetLastError=true)]
   static extern bool SetCurrentDirectory(string name);
}
```

10.5.8 Method body

The *method-body* of a method declaration consists of either a *block* or a semicolon.

Abstract and external method declarations do not provide a method implementation, so their method bodies simply consist of a semicolon. For any other method, the method body is a block (Section 8.2) that contains the statements to execute when that method is invoked.

When the return type of a method is `void`, `return` statements (Section 8.9.4) in that method's body are not permitted to specify an expression. If execution of the method body of a void method completes normally (that is, control flows off the end of the method body), that method simply returns to its caller.

When the return type of a method is not `void`, each `return` statement in that method's body must specify an expression of a type that is implicitly convertible to the return type. The endpoint of the method body of a value-returning method must not be reachable. In other words, in a value-returning method, control is not permitted to flow off the end of the method body.

In the example

```
class A
{
   public int F() {}         // Error, return value required
   public int G() {
      return 1;
   }
   public int H(bool b) {
      if (b) {
         return 1;
      }
      else {
         return 0;
      }
   }
}
```

the value-returning `F` method results in a compile-time error because control can flow off the end of the method body. The `G` and `H` methods are correct because all possible execution paths end in a return statement that specifies a return value.

10.5.9 Method overloading

The method overload resolution rules are described in Section 7.4.2.

10.6 Properties

A property is a member that provides access to a characteristic of an object or a class. Examples of properties include the length of a string, the size of a font, the caption of a window, the name of a customer, and so on. Properties are a natural extension of fields — both are named members with associated types, and the syntax for accessing fields and properties is the same. However, unlike fields, properties do not denote storage locations. Instead, properties have accessors that specify the statements to be executed when their values are read or written. Properties thus provide a mechanism for associating actions with the reading and writing of an object's attributes; furthermore, they permit such attributes to be computed.

Properties are declared using *property-declarations*:

property-declaration:
  attributes$_{opt}$ *property-modifiers*$_{opt}$ *type* *member-name* { *accessor-declarations* }
property-modifiers:
  property-modifier
  property-modifiers *property-modifier*
property-modifier:
  new
  public
  protected
  internal
  private
  static
  virtual
  sealed
  override
  abstract
  extern
member-name:
  identifier
  interface-type . *identifier*

A *property-declaration* may include a set of *attributes* (Section 17) and a valid combination of the four access modifiers (Section 10.2.3), the `new` (Section 10.2.2), `static` (Section 10.5.2), `virtual` (Section 10.5.3), `override` (Section 10.5.4), `sealed` (Section 10.5.5), `abstract` (Section 10.5.6), and `extern` (Section 10.5.7) modifiers.

Property declarations are subject to the same rules as method declarations (Section 10.5) with regard to valid combinations of modifiers.

The *type* of a property declaration specifies the type of the property introduced by the declaration, and the *member-name* specifies the name of the property. Unless the property is an explicit interface member implementation, the *member-name* is simply an *identifier*. For an explicit interface member implementation (Section 13.4.1), the *member-name* consists of an *interface-type* followed by a "`.`" and an *identifier*.

The *type* of a property must be at least as accessible as the property itself (Section 3.5.4).

The *accessor-declarations*, which must be enclosed in "`{`" and "`}`" tokens, declare the accessors (Section 10.6.2) of the property. The accessors specify the executable statements associated with reading and writing the property.

Even though the syntax for accessing a property is the same as that for a field, a property is not classified as a variable. Thus, it is not possible to pass a property as a `ref` or `out` argument.

When a property declaration includes an `extern` modifier, the property is said to be an external property. Because an external property declaration provides no actual implementation, each of its *accessor-declarations* consists of a semicolon.

10.6.1 Static and instance properties

When a property declaration includes a `static` modifier, the property is said to be a static property. When no `static` modifier is present, the property is said to be an instance property.

A static property is not associated with a specific instance, and it is a compile-time error to refer to `this` in the accessors of a static property.

An instance property is associated with a given instance of a class, and that instance can be accessed as `this` (Section 7.5.7) in the accessors of that property.

When a property is referenced in a *member-access* (Section 7.5.4) of the form `E.M`, if `M` is a static property, `E` must denote a type containing `M`, and if `M` is an instance property, `E` must denote an instance of a type containing `M`.

The differences between static and instance members are discussed further in Section 10.2.5.

10.6.2 Accessors

The *accessor-declarations* of a property specify the executable statements associated with reading and writing that property.

accessor-declarations:
  get-accessor-declaration *set-accessor-declaration*$_{opt}$
  set-accessor-declaration *get-accessor-declaration*$_{opt}$
get-accessor-declaration:
  attributes$_{opt}$ get *accessor-body*
set-accessor-declaration:
  attributes$_{opt}$ set *accessor-body*
accessor-body:
  block
  ;

The accessor declarations consist of a *get-accessor-declaration*, a *set-accessor-declaration*, or both. Each accessor declaration consists of the token `get` or `set` followed by an *accessor-body*. For `abstract` and `extern` properties, the *accessor-body* for each accessor specified is simply a semicolon. For the accessors of any non-abstract, non-extern property, the *accessor-body* is a *block* that specifies the statements to be executed when the corresponding accessor is invoked.

A `get` accessor corresponds to a parameterless method with a return value of the property type. Except as the target of an assignment, when a property is referenced in an expression, the `get` accessor of the property is invoked to compute the value of the property (Section 7.1.1). The body of a `get` accessor must conform to the rules for value-returning methods described in Section 10.5.8. In particular, all `return` statements in the body of a `get` accessor must specify an expression that is implicitly convertible to the property type. Furthermore, the endpoint of a `get` accessor must not be reachable.

A `set` accessor corresponds to a method with a single value parameter of the property type and a `void` return type. The implicit parameter of a `set` accessor is always named `value`. When a property is referenced as the target of an assignment (Section 7.13), or as the operand of `++` or `--` (Section 7.5.9, Section 7.6.5), the `set` accessor is invoked with an argument (whose value is that of the right-hand side of the assignment or the operand of the `++` or `--` operator) that provides the new value (Section 7.13.1). The body of a `set` accessor must conform to the rules for `void` methods described in Section 10.5.8. In particular, `return` statements in the `set` accessor body are not permitted to specify an expression. Since a `set` accessor implicitly has a parameter named `value`, it is a compile-time error for a local variable or constant declaration in a `set` accessor to have that name.

Based on the presence or absence of the `get` and `set` accessors, a property is classified as follows:

- A property that includes both a `get` accessor and a `set` accessor is said to be a read-write property.
- A property that has only a `get` accessor is said to be a read-only property. It is a compile-time error for a read-only property to be the target of an assignment.
- A property that has only a `set` accessor is said to be a write-only property. Except as the target of an assignment, it is a compile-time error to reference a write-only property in an expression.

In the example

```
public class Button: Control
{
   private string caption;
   public string Caption {
      get {
         return caption;
      }
      set {
         if (caption != value) {
            caption = value;
            Repaint();
         }
      }
   }
   public override void Paint(Graphics g, Rectangle r) {
      // Painting code goes here
   }
}
```

the `Button` control declares a public `Caption` property. The `get` accessor of the `Caption` property returns the string stored in the private `caption` field. The `set` accessor checks if the new value is different from the current value, and if so, it stores the new value and repaints the control. Properties often follow the pattern shown above: The `get` accessor simply returns a value stored in a private field, and the `set` accessor modifies that private field and then performs any additional actions required to fully update the state of the object.

Given the `Button` class above, the following is an example of use of the `Caption` property:

```
Button okButton = new Button();
okButton.Caption = "OK";         // Invokes set accessor
string s = okButton.Caption;      // Invokes get accessor
```

Here, the `set` accessor is invoked by assigning a value to the property, and the `get` accessor is invoked by referencing the property in an expression.

The `get` and `set` accessors of a property are not distinct members, and it is not possible to declare the accessors of a property separately. As such, it is not possible for the two accessors of a read-write property to have different accessibility. The example

```
class A
{
   private string name;
   public string Name {            // Error, duplicate member name
      get { return name; }
   }
   public string Name {            // Error, duplicate member name
      set { name = value; }
   }
}
```

does not declare a single read-write property. Rather, it declares two properties with the same name, one read-only and one write-only. Since two members declared in the same class cannot have the same name, the example causes a compile-time error to occur.

When a derived class declares a property by the same name as an inherited property, the derived property hides the inherited property with respect to both reading and writing. In the example

```
class A
{
   public int P {
      set {...}
   }
}
class B: A
{
   new public int P {
      get {...}
   }
}
```

the `P` property in `B` hides the `P` property in `A` with respect to both reading and writing. Thus, in the statements

```
B b = new B();
b.P = 1;         // Error, B.P is read-only
((A)b).P = 1;   // Ok, reference to A.P
```

the assignment to `b.P` causes a compile-time error to be reported, since the read-only `P` property in `B` hides the write-only `P` property in `A`. Note, however, that a cast can be used to access the hidden `P` property.

Unlike public fields, properties provide a separation between an object's internal state and its public interface. Consider the example:

```
class Label
{
   private int x, y;
```

```
   private string caption;
   public Label(int x, int y, string caption) {
      this.x = x;
      this.y = y;
      this.caption = caption;
   }
   public int X {
      get { return x; }
   }
   public int Y {
      get { return y; }
   }
   public Point Location {
      get { return new Point(x, y); }
   }
   public string Caption {
      get { return caption; }
   }
}
```

Here, the `Label` class uses two `int` fields, `x` and `y`, to store its location. The location is publicly exposed both as an `X` and a `Y` property and as a `Location` property of type `Point`. If, in a future version of `Label`, it becomes more convenient to store the location as a `Point` internally, the change can be made without affecting the public interface of the class:

```
class Label
{
   private Point location;
   private string caption;
   public Label(int x, int y, string caption) {
      this.location = new Point(x, y);
      this.caption = caption;
   }
   public int X {
      get { return location.x; }
   }
   public int Y {
      get { return location.y; }
   }
   public Point Location {
      get { return location; }
   }
   public string Caption {
      get { return caption; }
   }
}
```

Had `x` and `y` instead been `public readonly` fields, it would have been impossible to make such a change to the `Label` class.

Exposing state through properties is not necessarily any less efficient than exposing fields directly. In particular, when a property is non-virtual and contains only a small amount of code, the execution environment may replace calls to accessors with the actual code of the accessors. This process is known as inlining, and it makes property access as efficient as field access, yet preserves the increased flexibility of properties.

Since invoking a `get` accessor is conceptually equivalent to reading the value of a field, it is considered bad programming style for `get` accessors to have observable side effects. In the example

```
class Counter
{
   private int next;
   public int Next {
      get { return next++; }
   }
}
```

the value of the `Next` property depends on the number of times the property has previously been accessed. Thus, accessing the

property produces an observable side effect, and the property should be implemented as a method instead.

The "no side-effects" convention for `get` accessors doesn't mean that `get` accessors should always be written to simply return values stored in fields. Indeed, `get` accessors often compute the value of a property by accessing multiple fields or invoking methods. However, a properly designed `get` accessor performs no actions that cause observable changes in the state of the object.

Properties can be used to delay initialization of a resource until the moment it is first referenced. For example:

```
using System.IO;
public class Console
{
   private static TextReader reader;
   private static TextWriter writer;
   private static TextWriter error;
   public static TextReader In {
      get {
         if (reader == null) {
            reader = new StreamReader(Console.OpenStandardInput());
         }
         return reader;
      }
   }
   public static TextWriter Out {
      get {
         if (writer == null) {
            writer = new StreamWriter(Console.OpenStandardOutput());
         }
         return writer;
      }
   }
   public static TextWriter Error {
      get {
         if (error == null) {
            error = new StreamWriter(Console.OpenStandardError());
         }
         return error;
      }
   }
}
```

The `Console` class contains three properties, `In`, `Out`, and `Error`, that represent the standard input, output, and error devices, respectively. By exposing these members as properties, the `Console` class can delay their initialization until they are actually used. For example, upon first referencing the `Out` property, as in

```
Console.Out.WriteLine("hello, world");
```

the underlying `TextWriter` for the output device is created. But if the application makes no reference to the `In` and `Error` properties, then no objects are created for those devices.

10.6.3 Virtual, sealed, override, and abstract accessors

A `virtual` property declaration specifies that the accessors of the property are virtual. The `virtual` modifier applies to both accessors of a read-write property — it is not possible for only one accessor of a read-write property to be virtual.

An `abstract` property declaration specifies that the accessors of the property are virtual, but does not provide an actual implementation of the accessors. Instead, non-abstract derived classes are required to provide their own implementation for the accessors by overriding the property. Because an accessor for an abstract property declaration provides no actual implementation, its *accessor-body* simply consists of a semicolon.

A property declaration that includes both the `abstract` and `override` modifiers specifies that the property is abstract and overrides a base property. The accessors of such a property are also abstract.

Abstract property declarations are only permitted in abstract classes (Section 10.1.1.1).The accessors of an inherited virtual property can be overridden in a derived class by including a property declaration that specifies an `override` directive. This is known as an overriding property declaration. An overriding property declaration does not declare a new property. Instead, it simply specializes the implementations of the accessors of an existing virtual property.

An overriding property declaration must specify the exact same accessibility modifiers, type, and name as the inherited property. If the inherited property has only a single accessor (i.e., if the inherited property is read-only or write-only), the overriding property must include only that accessor. If the inherited property includes both accessors (i.e., if the inherited property is read-write), the overriding property can include either a single accessor or both accessors.

An overriding property declaration may include the `sealed` modifier. Use of this modifier prevents a derived class from further overriding the property. The accessors of a sealed property are also sealed.

Except for differences in declaration and invocation syntax, virtual, sealed, override, and abstract accessors behave exactly like virtual, sealed, override and abstract methods. Specifically, the rules described in Section 10.5.3, Section 10.5.4, Section 10.5.5, and Section 10.5.6 apply as if accessors were methods of a corresponding form:

- A `get` accessor corresponds to a parameterless method with a return value of the property type and the same modifiers as the containing property.
- A `set` accessor corresponds to a method with a single value parameter of the property type, a `void` return type, and the same modifiers as the containing property.

In the example

```
abstract class A
{
   int y;
   public virtual int X {
      get { return 0; }
   }
   public virtual int Y {
      get { return y; }
      set { y = value; }
   }
   public abstract int Z { get; set; }
}
```

`X` is a virtual read-only property, `Y` is a virtual read-write property, and `Z` is an abstract read-write property. Because `Z` is abstract, the containing class `A` must also be declared abstract.

A class that derives from `A` is show below:

```
class B: A
{
   int z;
   public override int X {
      get { return base.X + 1; }
   }
   public override int Y {
      set { base.Y = value < 0? 0: value; }
   }
   public override int Z {
```

```
        get { return z; }
        set { z = value; }
    }
}
```

Here, the declarations of `X`, `Y`, and `Z` are overriding property declarations. Each property declaration exactly matches the accessibility modifiers, type, and name of the corresponding inherited property. The `get` accessor of `X` and the `set` accessor of `Y` use the `base` keyword to access the inherited accessors. The declaration of `Z` overrides both abstract accessors — thus, there are no outstanding abstract function members in `B`, and `B` is permitted to be a non-abstract class.

10.7 Events

An event is a member that enables an object or class to provide notifications. Clients can attach executable code for events by supplying event handlers.

Events are declared using *event-declarations*:

event-declaration:
  attributes$_{opt}$ *event-modifiers*$_{opt}$ event *type* *variable-declarators* ;
  attributes$_{opt}$ *event-modifiers*$_{opt}$ event *type* *member-name* { *event-accessor-declarations* }
event-modifiers:
  event-modifier
  event-modifiers *event-modifier*
event-modifier:
  new
  public
  protected
  internal
  private
  static
  virtual
  sealed
  override
  abstract
  extern
event-accessor-declarations:
  add-accessor-declaration *remove-accessor-declaration*
  remove-accessor-declaration *add-accessor-declaration*
add-accessor-declaration:
  attributes$_{opt}$ add *block*
remove-accessor-declaration:
  attributes$_{opt}$ remove *block*

An *event-declaration* may include a set of *attributes* (Section 17) and a valid combination of the four access modifiers (Section 10.2.3), the `new` (Section 10.2.2), `static` (Section 10.5.2), `virtual` (Section 10.5.3), `override` (Section 10.5.4), `sealed` (Section 10.5.5), `abstract` (Section 10.5.6), and `extern` (Section 10.5.7) modifiers.

Event declarations are subject to the same rules as method declarations (Section 10.5) with regard to valid combinations of modifiers.

The *type* of an event declaration must be a *delegate-type* (Section 4.2), and that *delegate-type* must be at least as accessible as the event itself (Section 3.5.4).

An event declaration may include *event-accessor-declaration*s. However, if it does not, for non-extern, non-abstract events, the compiler supplies them automatically (Section 3.7.1); for extern events, the accessors are provided externally.

An event declaration that omits *event-accessor-declaration*s defines one or more events — one for each of the *variable-declarator*s. The attributes and modifiers apply to all of the members declared by such an *event-declaration*.

It is a compile-time error for an *event-declaration* to include both the `abstract` modifier and brace-delimited *event-accessor-declaration*s.

When an event declaration includes an `extern` modifier, the event is said to be an external event. Because an external event declaration provides no actual implementation, it is an error for it to include both the `extern` modifier and *event-accessor-declaration*s.

An event can be used as the left-hand operand of the `+=` and `-=` operators (Section 7.13.3). These operators are used, respectively, to attach event handlers to or to remove event handlers from an event, and the access modifiers of the event control the contexts in which such operations are permitted.

Since `+=` and `-=` are the only operations that are permitted on an event outside the type that declares the event, external code can add and remove handlers for an event, but cannot in any other way obtain or modify the underlying list of event handlers.

In an operation of the form `x += y` or `x -= y`, when `x` is an event and the reference takes place outside the type that contains the declaration of `x`, the result of the operation has type `void` (as opposed to having the type of `x`, with the value of `x` after the

assignment). This rule prohibits external code from indirectly examining the underlying delegate of an event.

The following example shows how event handlers are attached to instances of the `Button` class:

```
public delegate void EventHandler(object sender, EventArgs e);
public class Button: Control
{
   public event EventHandler Click;
}
public class LoginDialog: Form
{
   Button OkButton;
   Button CancelButton;
   public LoginDialog() {
      OkButton = new Button(...);
      OkButton.Click += new EventHandler(OkButtonClick);
      CancelButton = new Button(...);
      CancelButton.Click += new EventHandler(CancelButtonClick);
   }
   void OkButtonClick(object sender, EventArgs e) {
      // Handle OkButton.Click event
   }
   void CancelButtonClick(object sender, EventArgs e) {
      // Handle CancelButton.Click event
   }
}
```

Here, the `LoginDialog` instance constructor creates two `Button` instances and attaches event handlers to the `Click` events.

10.7.1 Field-like events

Within the program text of the class or struct that contains the declaration of an event, certain events can be used like fields. To be used in this way, an event must not be `abstract` or `extern`, and must not explicitly include *event-accessor-declarations*. Such an event can be used in any context that permits a field. The field contains a delegate (Section 15) that refers to the list of event handlers that have been added to the event. If no event handlers have been added, the field contains `null`.

In the example

```
public delegate void EventHandler(object sender, EventArgs e);
public class Button: Control
{
   public event EventHandler Click;
   protected void OnClick(EventArgs e) {
      if (Click != null) Click(this, e);
   }
   public void Reset() {
      Click = null;
   }
}
```

`Click` is used as a field within the `Button` class. As the example demonstrates, the field can be examined, modified, and used in delegate invocation expressions. The `OnClick` method in the `Button` class "raises" the `Click` event. The notion of raising an event is precisely equivalent to invoking the delegate represented by the event — thus, there are no special language constructs for raising events. Note that the delegate invocation is preceded by a check that ensures the delegate is non-null.

Outside the declaration of the `Button` class, the `Click` member can only be used on the left side of the += and -= operators, as in

```
b.Click += new EventHandler(...);
```

which appends a delegate to the invocation list of the `Click` event, and

```
b.Click -= new EventHandler(...);
```

which removes a delegate from the invocation list of the `Click` event.

When compiling a field-like event, the compiler automatically creates storage to hold the delegate, and creates accessors for the event that add or remove event handlers to the delegate field. In order to be thread-safe, the addition or removal operations are done while holding the lock (Section 8.12) on the containing object for an instance event, or the type object (Section 7.5.11) for a static event.

Thus, an instance event declaration of the form:

```
class X {
   public event D Ev;
}
```

could be compiled to something equivalent to:

```
class X {
   private D __Ev;  // field to hold the delegate
   public event D Ev {
      add {
         lock(this) { __Ev = __Ev + value; }
      }
      remove {
         lock(this) { __Ev = __Ev - value; }
      }
   }
}
```

Within the class `X`, references to `Ev` are compiled to reference the hidden field `__Ev` instead. The name "`__Ev`" is arbitrary; the

hidden field could have any name or no name at all.

Similarly, a static event declaration of the form:

```
class X {
   public static event D Ev;
```

could be compiled to something equivalent to:

```
class X {
   private static D __Ev;  // field to hold the delegate
   public static event D Ev {
      add {
         lock(typeof(X)) { __Ev = __Ev + value; }
      }
      remove {
         lock(typeof(X)) { __Ev = __Ev - value; }
      }
   }
}
```

10.7.2 Event accessors

Event declarations typically omit *event-accessor-declarations*, as in the `Button` example above. One situation for doing so involves the case in which the storage cost of one field per event is not acceptable. In such cases, a class can include *event-accessor-declarations* and use a private mechanism for storing the list of event handlers.

The *event-accessor-declarations* of an event specify the executable statements associated with adding and removing event handlers.

The accessor declarations consist of an *add-accessor-declaration* and a *remove-accessor-declaration*. Each accessor declaration consists of the token `add` or `remove` followed by a *block*. The *block* associated with an *add-accessor-declaration* specifies the statements to execute when an event handler is added, and the *block* associated with a *remove-accessor-declaration* specifies the statements to execute when an event handler is removed.

Each *add-accessor-declaration* and *remove-accessor-declaration* corresponds to a method with a single value parameter of the event type and a `void` return type. The implicit parameter of an event accessor is named `value`. When an event is used in an event assignment, the appropriate event accessor is used. Specifically, if the assignment operator is `+=` then the add accessor is used, and if the assignment operator is `-=` then the remove accessor is used. In either case, the right-hand operand of the assignment operator is used as the argument to the event accessor. The block of an *add-accessor-declaration* or a *remove-accessor-declaration* must conform to the rules for `void` methods described in Section 10.5.8. In particular, `return` statements in such a block are not permitted to specify an expression.

Since an event accessor implicitly has a parameter named `value`, it is a compile-time error for a local variable or constant declared in an event accessor to have that name.

In the example

```
class Control: Component
{
   // Unique keys for events
   static readonly object mouseDownEventKey = new object();
   static readonly object mouseUpEventKey = new object();
   // Return event handler associated with key
   protected Delegate GetEventHandler(object key) {...}
   // Add event handler associated with key
   protected void AddEventHandler(object key, Delegate handler) {...}
   // Remove event handler associated with key
   protected void RemoveEventHandler(object key, Delegate handler) {...}
   // MouseDown event
   public event MouseEventHandler MouseDown {
      add { AddEventHandler(mouseDownEventKey, value); }
      remove { RemoveEventHandler(mouseDownEventKey, value); }
   }
   // MouseUp event
   public event MouseEventHandler MouseUp {
      add { AddEventHandler(mouseUpEventKey, value); }
      remove { RemoveEventHandler(mouseUpEventKey, value); }
   }
   // Invoke the MouseUp event
   protected void OnMouseUp(MouseEventArgs args) {
      MouseEventHandler handler;
      handler = (MouseEventHandler)GetEventHandler(mouseUpEventKey);
      if (handler != null)
         handler(this, args);
   }
}
```

the `Control` class implements an internal storage mechanism for events. The `AddEventHandler` method associates a delegate value with a key, the `GetEventHandler` method returns the delegate currently associated with a key, and the `RemoveEventHandler` method removes a delegate as an event handler for the specified event. Presumably, the underlying storage mechanism is designed such that there is no cost for associating a `null` delegate value with a key, and thus unhandled events consume no storage.

10.7.3 Static and instance events

When an event declaration includes a `static` modifier, the event is said to be a static event. When no `static` modifier is present, the event is said to be an instance event.

A static event is not associated with a specific instance, and it is a compile-time error to refer to `this` in the accessors of a static event.

An instance event is associated with a given instance of a class, and this instance can be accessed as `this` (Section 7.5.7) in the accessors of that event.

When an event is referenced in a *member-access* (Section 7.5.4) of the form `E.M`, if `M` is a static event, `E` must denote a type containing `M`, and if `M` is an instance event, `E` must denote an instance of a type containing `M`.

The differences between static and instance members are discussed further in Section 10.2.5.

10.7.4 Virtual, sealed, override, and abstract accessors

A `virtual` event declaration specifies that the accessors of that event are virtual. The `virtual` modifier applies to both accessors of an event.

An `abstract` event declaration specifies that the accessors of the event are virtual, but does not provide an actual implementation of the accessors. Instead, non-abstract derived classes are required to provide their own implementation for the accessors by overriding the event. Because an accessor for an abstract event declaration provides no actual implementation, its *accessor-body* simply consists of a semicolon.

An event declaration that includes both the `abstract` and `override` modifiers specifies that the event is abstract and overrides a base event. The accessors of such an event are also abstract.

Abstract event declarations are only permitted in abstract classes (Section 10.1.1.1).

The accessors of an inherited virtual event can be overridden in a derived class by including an event declaration that specifies an `override` modifier. This is known as an overriding event declaration. An overriding event declaration does not declare a new event. Instead, it simply specializes the implementations of the accessors of an existing virtual event.

An overriding event declaration must specify the exact same accessibility modifiers, type, and name as the overridden event.

An overriding event declaration may include the `sealed` modifier. Use of this modifier prevents a derived class from further overriding the event. The accessors of a sealed event are also sealed.

It is a compile-time error for an overriding event declaration to include a `new` modifier.

Except for differences in declaration and invocation syntax, virtual, sealed, override, and abstract accessors behave exactly like virtual, sealed, override and abstract methods. Specifically, the rules described in Section 10.5.3, Section 10.5.4, Section 10.5.5, and Section 10.5.6 apply as if accessors were methods of a corresponding form. Each accessor corresponds to a method with a single value parameter of the event type, a `void` return type, and the same modifiers as the containing event.

10.8 Indexers

An indexer is a member that enables an object to be indexed in the same way as an array. Indexers are declared using *indexer-declarations*:

indexer-declaration:
  attributes$_{opt}$ *indexer-modifiers*$_{opt}$ *indexer-declarator* { *accessor-declarations* }
indexer-modifiers:
  indexer-modifier
  indexer-modifiers *indexer-modifier*
indexer-modifier:
  new
  public
  protected
  internal
  private
  virtual
  sealed
  override
  abstract
  extern
indexer-declarator:
  type this [*formal-parameter-list*]
  type *interface-type* . this [*formal-parameter-list*]

An *indexer-declaration* may include a set of *attributes* (Section 17) and a valid combination of the four access modifiers (Section 10.2.3), the `new` (Section 10.2.2), `virtual` (Section 10.5.3), `override` (Section 10.5.4), `sealed` (Section 10.5.5), `abstract` (Section 10.5.6), and `extern` (Section 10.5.7) modifiers.

Indexer declarations are subject to the same rules as method declarations (Section 10.5) with regard to valid combinations of modifiers, with the one exception being that the static modifier is not permitted on an indexer declaration.

The modifiers `virtual`, `override`, and `abstract` are mutually exclusive except in one case. The `abstract` and `override` modifiers may be used together so that an abstract indexer can override a virtual one.

The *type* of an indexer declaration specifies the element type of the indexer introduced by the declaration. Unless the indexer is an explicit interface member implementation, the *type* is followed by the keyword `this`. For an explicit interface member implementation, the *type* is followed by an *interface-type*, a "`.`", and the keyword `this`. Unlike other members, indexers do not have user-defined names.

The *formal-parameter-list* specifies the parameters of the indexer. The formal parameter list of an indexer corresponds to that of a method (Section 10.5.1), except that at least one parameter must be specified, and that the `ref` and `out` parameter modifiers are not permitted.

The *type* of an indexer and each of the types referenced in the *formal-parameter-list* must be at least as accessible as the indexer itself (Section 3.5.4).

The *accessor-declarations* (Section 10.6.2), which must be enclosed in "`{`" and "`}`" tokens, declare the accessors of the indexer. The accessors specify the executable statements associated with reading and writing indexer elements.

Even though the syntax for accessing an indexer element is the same as that for an array element, an indexer element is not classified as a variable. Thus, it is not possible to pass an indexer element as a `ref` or `out` argument.

The formal parameter list of an indexer defines the signature (Section 3.6) of the indexer. Specifically, the signature of an indexer consists of the number and types of its formal parameters. The element type and names of the formal parameters are not part of an indexer's signature.

The signature of an indexer must differ from the signatures of all other indexers declared in the same class.

Indexers and properties are very similar in concept, but differ in the following ways:

- A property is identified by its name, whereas an indexer is identified by its signature.
- A property is accessed through a *simple-name* (Section 7.5.2) or a *member-access* (Section 7.5.4), whereas an indexer element is accessed through an *element-access* (Section 7.5.6.2).
- A property can be a `static` member, whereas an indexer is always an instance member.

- A `get` accessor of a property corresponds to a method with no parameters, whereas a `get` accessor of an indexer corresponds to a method with the same formal parameter list as the indexer.
- A `set` accessor of a property corresponds to a method with a single parameter named `value`, whereas a `set` accessor of an indexer corresponds to a method with the same formal parameter list as the indexer, plus an additional parameter named `value`.
- It is a compile-time error for an indexer accessor to declare a local variable with the same name as an indexer parameter.
- In an overriding property declaration, the inherited property is accessed using the syntax `base.P`, where `P` is the property name. In an overriding indexer declaration, the inherited indexer is accessed using the syntax `base[E]`, where `E` is a comma separated list of expressions.

Aside from these differences, all rules defined in Section 10.6.2 and Section 10.6.3 apply to indexer accessors as well as to property accessors.

When an indexer declaration includes an `extern` modifier, the indexer is said to be an external indexer. Because an external indexer declaration provides no actual implementation, each of its *accessor-declarations* consists of a semicolon.

The example below declares a `BitArray` class that implements an indexer for accessing the individual bits in the bit array.

```
using System;
class BitArray
{
   int[] bits;
   int length;
   public BitArray(int length) {
      if (length < 0) throw new ArgumentException();
      bits = new int[((length - 1) >> 5) + 1];
      this.length = length;
   }
   public int Length {
      get { return length; }
   }
   public bool this[int index] {
      get {
         if (index < 0 || index >= length) {
            throw new IndexOutOfRangeException();
         }
         return (bits[index >> 5] & 1 << index) != 0;
      }
      set {
         if (index < 0 || index >= length) {
            throw new IndexOutOfRangeException();
         }
         if (value) {
            bits[index >> 5] |= 1 << index;
         }
         else {
            bits[index >> 5] &= ~(1 << index);
         }
      }
   }
}
```

An instance of the `BitArray` class consumes substantially less memory than a corresponding `bool[]` (since each value of the former occupies only one bit instead of the latter's one byte), but it permits the same operations as a `bool[]`.

The following `CountPrimes` class uses a `BitArray` and the classical "sieve" algorithm to compute the number of primes between 1 and a given maximum:

```
class CountPrimes
{
   static int Count(int max) {
      BitArray flags = new BitArray(max + 1);
      int count = 1;
      for (int i = 2; i <= max; i++) {
         if (!flags[i]) {
```

```
            for (int j = i * 2; j <= max; j += i) flags[j] = true;
            count++;
         }
      }
      return count;
   }
   static void Main(string[] args) {
      int max = int.Parse(args[0]);
      int count = Count(max);
      Console.WriteLine("Found {0} primes between 1 and {1}", count, max);
   }
}
```

Note that the syntax for accessing elements of the `BitArray` is precisely the same as for a `bool[]`.

The following example shows a 26 by 10 grid class that has an indexer with two parameters. The first parameter is required to be an upper- or lowercase letter in the range A–Z, and the second is required to be an integer in the range 0–9.

```
using System;
class Grid
{
   const int NumRows = 26;
   const int NumCols = 10;
   int[,] cells = new int[NumRows, NumCols];
   public int this[char c, int col] {
      get {
         c = Char.ToUpper(c);
         if (c < 'A' || c > 'Z') {
            throw new ArgumentException();
         }
         if (col < 0 || col >= NumCols) {
            throw new IndexOutOfRangeException();
         }
         return cells[c - 'A', col];
      }
      set {
         c = Char.ToUpper(c);
         if (c < 'A' || c > 'Z') {
            throw new ArgumentException();
         }
         if (col < 0 || col >= NumCols) {
            throw new IndexOutOfRangeException();
         }
         cells[c - 'A', col] = value;
      }
   }
}
```

10.8.1 Indexer overloading

The indexer overload resolution rules are described in Section 7.4.2.

10.9 Operators

An operator is a member that defines the meaning of an expression operator that can be applied to instances of the class. Operators are declared using *operator-declarations*:

operator-declaration:
  attributes$_{opt}$ *operator-modifiers* *operator-declarator* *operator-body*
operator-modifiers:
  operator-modifier
  operator-modifiers *operator-modifier*
operator-modifier:
  public
  static
  extern
operator-declarator:
  unary-operator-declarator
  binary-operator-declarator
  conversion-operator-declarator
unary-operator-declarator:
  type operator *overloadable-unary-operator* (*type* *identifier*)
overloadable-unary-operator: one of
  + - ! ~ ++ -- true false
binary-operator-declarator:
  type operator *overloadable-binary-operator* (*type* *identifier* , *type* *identifier*)
overloadable-binary-operator: one of
  + - * / % & | ^ << >> == != > < >= <=
conversion-operator-declarator:
  implicit operator *type* (*type* *identifier*)
  explicit operator *type* (*type* *identifier*)
operator-body:
  block
  ;

There are three categories of overloadable operators: Unary operators (Section 10.9.1), binary operators (Section 10.9.2), and conversion operators (Section 10.9.3).

When an operator declaration includes an `extern` modifier, the operator is said to be an external operator. Because an external operator provides no actual implementation, its *operator-body* consists of a semi-colon. For all other operators, the *operator-body* consists of a *block*, which specifies the statements to execute when the operator is invoked. The *block* of an operator must conform to the rules for value-returning methods described in Section 10.5.8.

The following rules apply to all operator declarations:

- An operator declaration must include both a `public` and a `static` modifier.
- The parameter(s) of an operator must be value parameters. It is a compile-time error for an operator declaration to specify `ref` or `out` parameters.
- The signature of an operator (Section 10.9.1, Section 10.9.2, Section 10.9.3) must differ from the signatures of all other operators declared in the same class.
- All types referenced in an operator declaration must be at least as accessible as the operator itself (Section 3.5.4).
- It is an error for the same modifier to appear multiple times in an operator declaration.

Each operator category imposes additional restrictions, as described in the following sections.

Like other members, operators declared in a base class are inherited by derived classes. Because operator declarations always require the class or struct in which the operator is declared to participate in the signature of the operator, it is not possible for an operator declared in a derived class to hide an operator declared in a base class. Thus, the `new` modifier is never required, and therefore never permitted, in an operator declaration.

Additional information on unary and binary operators can be found in Section 7.2.

Additional information on conversion operators can be found in Section 6.4.

10.9.1 Unary operators

The following rules apply to unary operator declarations, where `T` denotes the class or struct type that contains the operator declaration:

- A unary `+`, `-`, `!`, or `~` operator must take a single parameter of type `T` and can return any type.
- A unary `++` or `--` operator must take a single parameter of type `T` and must return type `T`.
- A unary `true` or `false` operator must take a single parameter of type `T` and must return type `bool`.

The signature of a unary operator consists of the operator token (`+`, `-`, `!`, `~`, `++`, `--`, `true`, or `false`) and the type of the single formal parameter. The return type is not part of a unary operator's signature, nor is the name of the formal parameter.

The `true` and `false` unary operators require pair-wise declaration. A compile-time error occurs if a class declares one of these operators without also declaring the other. The `true` and `false` operators are described further in Section 7.16.

The following example shows an implementation and subsequent usage of operator `++` for an integer vector class:

```
public class IntVector
{
   public IntVector(int length) {...}
   public int Length {...}               // read-only property
   public int this[int index] {...}      // read-write indexer
   public static IntVector operator ++(IntVector iv) {
      IntVector temp = new IntVector(iv.Length);
      for (int i = 0; i < iv.Length; i++)
         temp[i] = iv[i] + 1;
      return temp;
   }
}
class Test
{
   static void Main() {
      IntVector iv1 = new IntVector(4);   // vector of 4 x 0
      IntVector iv2;
      iv2 = iv1++;   // iv2 contains 4 x 0, iv1 contains 4 x 1
      iv2 = ++iv1;   // iv2 contains 4 x 2, iv1 contains 4 x 2
   }
}
```

Note how the operator method returns the value produced by adding 1 to the operand, just like the postfix increment and decrement operators (Section 7.5.9), and the prefix increment and decrement operators (Section 7.6.5). Unlike in C++, this method need not modify the value of its operand directly. In fact, modifying the operand value would violate the standard semantics of the postfix increment operator.

10.9.2 Binary operators

A binary operator must take two parameters, at least one of which must have the class or struct type in which the operator is declared. Parameters of the shift operators (Section 7.8) are further constrained. A binary operator can return any type.

The signature of a binary operator consists of the operator token (`+`, `-`, `*`, `/`, `%`, `&`, `|`, `^`, `<<`, `>>`, `==`, `!=`, `>`, `<`, `>=`, or `<=`) and the types of the two formal parameters. The return type and the names of the formal parameters are not part of a binary operator's signature.

Certain binary operators require pair-wise declaration. For every declaration of either operator of a pair, there must be a matching declaration of the other operator of the pair. Two operator declarations match when they have the same return type and the same type for each parameter. The following operators require pair-wise declaration:

- `operator ==` and `operator !=`
- `operator >` and `operator <`
- `operator >=` and `operator <=`

10.9.3 Conversion operators

A conversion operator declaration introduces a user-defined conversion (Section 6.4), which augments the pre-defined implicit and explicit conversions.

A conversion operator declaration that includes the `implicit` keyword introduces a user-defined implicit conversion. Implicit conversions can occur in a variety of situations, including function member invocations, cast expressions, and assignments. This is described further in Section 6.1.

A conversion operator declaration that includes the `explicit` keyword introduces a user-defined explicit conversion. Explicit conversions can occur in cast expressions, and are described further in Section 6.2.

A conversion operator converts from a source type, indicated by the parameter type of the conversion operator, to a target type, indicated by the return type of the conversion operator. A class or struct is permitted to declare a conversion from a source type `S` to a target type `T` provided all of the following are true:

- `S` and `T` are different types.
- Either `S` or `T` is the class or struct type in which the operator declaration takes place.
- Neither `S` nor `T` is `object` or an *interface-type*.
- `T` is not a base class of `S`, and `S` is not a base class of `T`.

From the second rule it follows that a conversion operator must convert either to or from the class or struct type in which the operator is declared. For example, it is possible for a class or struct type `C` to define a conversion from `C` to `int` and from `int` to `C`, but not from `int` to `bool`.

It is not possible to redefine a pre-defined conversion. Thus, conversion operators are not allowed to convert from or to `object` because implicit and explicit conversions already exist between `object` and all other types. Likewise, neither the source nor the target types of a conversion can be a base type of the other, since a conversion would then already exist.

User-defined conversions are not allowed to convert from or to *interface-types*. In particular, this restriction ensures that no user-defined transformations occur when converting to an *interface-type*, and that a conversion to an *interface-type* succeeds only if the object being converted actually implements the specified *interface-type*.

The signature of a conversion operator consists of the source type and the target type. (Note that this is the only form of member for which the return type participates in the signature.) The `implicit` or `explicit` classification of a conversion operator is not part of the operator's signature. Thus, a class or struct cannot declare both an `implicit` and an `explicit` conversion operator with the same source and target types.

In general, user-defined implicit conversions should be designed to never throw exceptions and never lose information. If a user-defined conversion can give rise to exceptions (for example, because the source argument is out of range) or loss of information (such as discarding high-order bits), then that conversion should be defined as an explicit conversion.

In the example

```
using System;
public struct Digit
{
   byte value;
   public Digit(byte value) {
      if (value < 0 || value > 9) throw new ArgumentException();
      this.value = value;
   }
   public static implicit operator byte(Digit d) {
      return d.value;
   }
   public static explicit operator Digit(byte b) {
      return new Digit(b);
   }
}
```

the conversion from `Digit` to `byte` is implicit because it never throws exceptions or loses information, but the conversion from `byte` to `Digit` is explicit since `Digit` can only represent a subset of the possible values of a `byte`.

10.10 Instance constructors

An instance constructor is a member that implements the actions required to initialize an instance of a class. Instance constructors are declared using *constructor-declarations*:

constructor-declaration:
  attributes$_{opt}$ *constructor-modifiers*$_{opt}$ *constructor-declarator* *constructor-body*
constructor-modifiers:
  constructor-modifier
  constructor-modifiers *constructor-modifier*
constructor-modifier:
  public
  protected
  internal
  private
  extern
constructor-declarator:
  identifier (*formal-parameter-list*$_{opt}$) *constructor-initializer*$_{opt}$
constructor-initializer:
  : base (*argument-list*$_{opt}$)
  : this (*argument-list*$_{opt}$)
constructor-body:
  block
  ;

A *constructor-declaration* may include a set of *attributes* (Section 17), a valid combination of the four access modifiers (Section 10.2.3), and an `extern` (Section 10.5.7) modifier. A constructor declaration is not permitted to include the same modifier multiple times.

The *identifier* of a *constructor-declarator* must name the class in which the instance constructor is declared. If any other name is specified, a compile-time error occurs.

The optional *formal-parameter-list* of an instance constructor is subject to the same rules as the *formal-parameter-list* of a method (Section 10.5). The formal parameter list defines the signature (Section 3.6) of an instance constructor and governs the process whereby overload resolution (Section 7.4.2) selects a particular instance constructor in an invocation.

Each of the types referenced in the *formal-parameter-list* of an instance constructor must be at least as accessible as the constructor itself (Section 3.5.4).

The optional *constructor-initializer* specifies another instance constructor to invoke before executing the statements given in the *constructor-body* of this instance constructor. This is described further in Section 10.10.1.

When a constructor declaration includes an `extern` modifier, the constructor is said to be an external constructor. Because an external constructor declaration provides no actual implementation, its *constructor-body* consists of a semicolon. For all other constructors, the *constructor-body* consists of a *block* that specifies the statements to initialize a new instance of the class. This corresponds exactly to the *block* of an instance method with a `void` return type (Section 10.5.8).

Instance constructors are not inherited. Thus, a class has no instance constructors other than those actually declared in the class. If a class contains no instance constructor declarations, a default instance constructor is automatically provided (Section 10.10.4).

Instance constructors are invoked by *object-creation-expressions* (Section 7.5.10.1) and through *constructor-initializers*.

10.10.1 Constructor initializers

All instance constructors (except those for class `object`) implicitly include an invocation of another instance constructor immediately before the *constructor-body*. The constructor to implicitly invoke is determined by the *constructor-initializer*:

- An instance constructor initializer of the form `base(`*argument-listopt*`)` causes an instance constructor from the direct base class to be invoked. That constructor is selected using *argument-list* and the overload resolution rules of Section 7.4.2. The set of candidate instance constructors consists of all accessible instance constructors contained in the direct base class (including any default constructor, as defined in Section 10.10.4). If this set is empty, or if a single best instance constructor cannot be identified, a compile-time error occurs.
- An instance constructor initializer of the form `this(`*argument-listopt*`)` causes an instance constructor from the class itself to be invoked. The constructor is selected using *argument-list* and the overload resolution rules of Section 7.4.2. The set of candidate instance constructors consists of all accessible instance constructors declared in the class itself. If this set is empty, or if a single best instance constructor cannot be identified, a compile-time error occurs. If an instance constructor declaration includes a constructor initializer that invokes the constructor itself, a compile-time error occurs.

If an instance constructor has no constructor initializer, a constructor initializer of the form `base()` is implicitly provided. Thus, an instance constructor declaration of the form

```
C(...) {...}
```

is exactly equivalent to

```
C(...): base() {...}
```

The scope of the parameters given by the *formal-parameter-list* of an instance constructor declaration includes the constructor initializer of that declaration. Thus, a constructor initializer is permitted to access the parameters of the constructor. For example:

```
class A
{
   public A(int x, int y) {}
}
class B: A
{
   public B(int x, int y): base(x + y, x - y) {}
}
```

An instance constructor initializer cannot access the instance being created. Therefore it is a compile-time error to reference `this` in an argument expression of the constructor initializer, as is it a compile-time error for an argument expression to reference any instance member through a *simple-name*.

10.10.2 Instance variable initializers

When an instance constructor has no constructor initializer, or it has a constructor initializer of the form `base(...)`, that constructor implicitly performs the initializations specified by the *variable-initializer*s of the instance fields declared in its class. This corresponds to a sequence of assignments that are executed immediately upon entry to the constructor and before the implicit invocation of the direct base class constructor. The variable initializers are executed in the textual order in which they appear in the class declaration.

10.10.3 Constructor execution

Variable initializers are transformed into assignment statements, and these assignment statements are executed before the invocation of the base class instance constructor. This ordering ensures that all instance fields are initialized by their variable initializers before any statements that have access to that instance are executed.

Given the example

```
using System;
class A
{
   public A() {
      PrintFields();
   }
   public virtual void PrintFields() {}
}
class B: A
{
   int x = 1;
   int y;
   public B() {
      y = -1;
   }
   public override void PrintFields() {
      Console.WriteLine("x = {0}, y = {1}", x, y);
   }
}
```

when `new B()` is used to create an instance of `B`, the following output is produced:

```
x = 1, y = 0
```

The value of `x` is 1 because the variable initializer is executed before the base class instance constructor is invoked. However, the value of `y` is 0 (the default value of an `int`) because the assignment to `y` is not executed until after the base class constructor returns.

It is useful to think of instance variable initializers and constructor initializers as statements that are automatically inserted before the *constructor-body*. The example

```
using System;
using System.Collections;
class A
{
   int x = 1, y = -1, count;
   public A() {
      count = 0;
   }
   public A(int n) {
      count = n;
   }
}
class B: A
{
   double sqrt2 = Math.Sqrt(2.0);
   ArrayList items = new ArrayList(100);
   int max;
   public B(): this(100) {
      items.Add("default");
   }
   public B(int n): base(n - 1) {
      max = n;
   }
}
```

contains several variable initializers; it also contains constructor initializers of both forms (`base` and `this`). The example corresponds to the code shown below, where each comment indicates an automatically inserted statement (the syntax used for the automatically inserted constructor invocations is not valid, but merely serves to illustrate the mechanism).

```
using System.Collections;
class A
{
   int x, y, count;
   public A() {
      x = 1;                        // Variable initializer
      y = -1;                        // Variable initializer
      object();                     // Invoke object() constructor
      count = 0;
   }
   public A(int n) {
      x = 1;                        // Variable initializer
      y = -1;                        // Variable initializer
      object();                     // Invoke object() constructor
      count = n;
   }
}
class B: A
{
   double sqrt2;
   ArrayList items;
   int max;
   public B(): this(100) {
      B(100);                        // Invoke B(int) constructor
      items.Add("default");
   }
   public B(int n): base(n - 1) {
      sqrt2 = Math.Sqrt(2.0);         // Variable initializer
      items = new ArrayList(100);   // Variable initializer
      A(n - 1);                     // Invoke A(int) constructor
      max = n;
   }
}
```

10.10.4 Default constructors

If a class contains no instance constructor declarations, a default instance constructor is automatically provided. That default constructor simply invokes the parameterless constructor of the direct base class. If the direct base class does not have an accessible parameterless instance constructor, a compile-time error occurs. If the class is abstract then the declared accessibility for the default constructor is protected. Otherwise, the declared accessibility for the default constructor is public. Thus, the default constructor is always of the form

```
protected C(): base() {}
```

or

```
public C(): base() {}
```

where `C` is the name of the class.

In the example

```
class Message
{
   object sender;
   string text;
}
```

a default constructor is provided because the class contains no instance constructor declarations. Thus, the example is precisely equivalent to

```
class Message
{
   object sender;
   string text;
   public Message(): base() {}
}
```

10.10.5 Private constructors

When a class `T` declares only private instance constructors, it is not possible for classes outside the program text of `T` to derive from `T` or to directly create instances of `T`. Thus, if a class contains only static members and isn't intended to be instantiated, adding an empty private instance constructor will prevent instantiation. For example:

```
public class Trig
{
   private Trig() {}      // Prevent instantiation
   public const double PI = 3.14159265358979323846;
   public static double Sin(double x) {...}
   public static double Cos(double x) {...}
   public static double Tan(double x) {...}
}
```

The `Trig` class groups related methods and constants, but is not intended to be instantiated. Therefore it declares a single empty private instance constructor. At least one instance constructor must be declared to suppress the automatic generation of a default constructor.

10.10.6 Optional instance constructor parameters

The `this(...)` form of constructor initializer is commonly used in conjunction with overloading to implement optional instance constructor parameters. In the example

```
class Text
{
   public Text(): this(0, 0, null) {}
   public Text(int x, int y): this(x, y, null) {}
   public Text(int x, int y, string s) {
      // Actual constructor implementation
   }
}
```

the first two instance constructors merely provide the default values for the missing arguments. Both use a `this(...)` constructor initializer to invoke the third instance constructor, which actually does the work of initializing the new instance. The effect is that of optional constructor parameters:

```
Text t1 = new Text();               // Same as Text(0, 0, null)
Text t2 = new Text(5, 10);            // Same as Text(5, 10, null)
Text t3 = new Text(5, 20, "Hello");
```

10.11 Static constructors

A static constructor is a member that implements the actions required to initialize a class. Static constructors are declared using *static-constructor-declarations*:

static-constructor-declaration:
  attributes$_{opt}$ *static-constructor-modifiers* *identifier* ( ) *static-constructor-body*
static-constructor-modifiers:
  extern$_{opt}$ static
  static extern$_{opt}$
static-constructor-body:
  block
  ;

A *static-constructor-declaration* may include a set of *attributes* (Section 17) and an `extern` (Section 10.5.7) modifier.

The *identifier* of a *static-constructor-declaration* must name the class in which the static constructor is declared. If any other name is specified, a compile-time error occurs.

When a static constructor declaration includes an `extern` modifier, the static constructor is said to be an external static constructor. Because an external static constructor declaration provides no actual implementation, its *static-constructor-body* consists of a semicolon. For all other static constructor declarations, the *static-constructor-body* consists of a *block* that specifies the statements to execute in order to initialize the class. This corresponds exactly to the *method-body* of a static method with a `void` return type (Section 10.5.8).

Static constructors are not inherited, and cannot be called directly.

The static constructor for a class executes at most once in a given application domain. The execution of a static constructor is triggered by the first of the following events to occur within an application domain:

- An instance of the class is created.
- Any of the static members of the class are referenced.

If a class contains the `Main` method (Section 3.1) in which execution begins, the static constructor for that class executes before the `Main` method is called. If a class contains any static fields with initializers, those initializers are executed in textual order immediately prior to executing the static constructor.

The example

```
using System;
class Test
{
   static void Main() {
      A.F();
      B.F();
   }
}
class A
{
   static A() {
      Console.WriteLine("Init A");
   }
   public static void F() {
      Console.WriteLine("A.F");
   }
}
class B
{
   static B() {
      Console.WriteLine("Init B");
   }
   public static void F() {
      Console.WriteLine("B.F");
   }
}
```

must produce the output:

```
Init A
A.F
Init B
B.F
```

because the execution of `A`'s static constructor is triggered by the call to `A.F`, and the execution of `B`'s static constructor is triggered by the call to `B.F`.

It is possible to construct circular dependencies that allow static fields with variable initializers to be observed in their default value state.

The example

```
using System;
class A
{
   public static int X;
   static A() {
      X = B.Y + 1;
   }
}
class B
{
   public static int Y = A.X + 1;
   static B() {}
   static void Main() {
      Console.WriteLine("X = {0}, Y = {1}", A.X, B.Y);
   }
}
```

produces the output

```
X = 1, Y = 2
```

To execute the `Main` method, the system first runs the initializer for `B.Y`, prior to class `B`'s static constructor. `Y`'s initializer causes `A`'s static constructor to be run because the value of `A.X` is referenced. The static constructor of `A` in turn proceeds to compute the value of `X`, and in doing so fetches the default value of `Y`, which is zero. `A.X` is thus initialized to 1. The process of running `A`'s static field initializers and static constructor then completes, returning to the calculation of the initial value of `Y`, the result of which becomes 2.

10.12 Destructors

A destructor is a member that implements the actions required to destruct an instance of a class. A destructor is declared using a *destructor-declaration*:

destructor-declaration:
  attributes$_{opt}$ extern$_{opt}$ ~ *identifier* ( ) *destructor-body*
destructor-body:
  block
  ;

A *destructor-declaration* may include a set of *attributes* (Section 17).

The *identifier* of a *destructor-declarator* must name the class in which the destructor is declared. If any other name is specified, a compile-time error occurs.

When a destructor declaration includes an `extern` modifier, the destructor is said to be an external destructor. Because an external destructor declaration provides no actual implementation, its *destructor-body* consists of a semicolon. For all other destructors, the *destructor-body* consists of a *block* that specifies the statements to execute in order to destruct an instance of the class. A *destructor-body* corresponds exactly to the *method-body* of an instance method with a `void` return type (Section 10.5.8).

Destructors are not inherited. Thus, a class has no destructors other than the one that may be declared in that class.

Since a destructor is required to have no parameters, it cannot be overloaded, so a class can have, at most, one destructor.

Destructors are invoked automatically, and cannot be invoked explicitly. An instance becomes eligible for destruction when it is no longer possible for any code to use that instance. Execution of the destructor for the instance may occur at any time after the instance becomes eligible for destruction. When an instance is destructed, the destructors in that instance's inheritance chain are called, in order, from most derived to least derived. A destructor may be executed on any thread. For further discussion of the rules that govern when and how a destructor is executed, see Section 3.9.

The output of the example

```
using System;
class A
{
   ~A() {
      Console.WriteLine("A's destructor");
   }
}
class B: A
{
   ~B() {
      Console.WriteLine("B's destructor");
   }
}
class Test
{
    static void Main() {
      B b = new B();
      b = null;
      GC.Collect();
      GC.WaitForPendingFinalizers();
   }
}
```

is

```
B's destructor
A's destructor
```

since destructors in an inheritance chain are called in order, from most derived to least derived.

Destructors are implemented by overriding the virtual method `Finalize` on `System.Object`. C# programs are not permitted to override this method or call it (or overrides of it) directly. For instance, the program

```
class A
{
   override protected void Finalize() {}   // error
   public void F() {
      this.Finalize();                     // error
   }
}
```

contains two errors.

The compiler behaves as if this method, and overrides of it, do not exist at all. Thus, this program:

```
class A
{
   void Finalize() {}                     // permitted
}
```

is valid, and the method shown hides `System.Object`'s `Finalize` method.

For a discussion of the behavior when an exception is thrown from a destructor, see Section 16.3.

11. Structs

Structs are similar to classes in that they represent data structures that can contain data members and function members. However, unlike classes, structs are value types and do not require heap allocation. A variable of a struct type directly contains the data of the struct, whereas a variable of a class type contains a reference to the data, the latter known as an object.

Structs are particularly useful for small data structures that have value semantics. Complex numbers, points in a coordinate system, or key-value pairs in a dictionary are all good examples of structs. Key to these data structures is that they have few data members, that they do not require use of inheritance or referential identity, and that they can be conveniently implemented using value semantics where assignment copies the value instead of the reference.

As described in Section 4.1.3, the simple types provided by C#, such as `int`, `double`, and `bool`, are in fact all struct types. Just as these predefined types are structs, it is also possible to use structs and operator overloading to implement new "primitive" types in the C# language. Two examples of such types are given at the end of this chapter (Section 11.4).

11.1 Struct declarations

A *struct-declaration* is a *type-declaration* (Section 9.5) that declares a new struct:

struct-declaration:
  attributes$_{opt}$ *struct-modifiers*$_{opt}$ struct *identifier* *struct-interfaces*$_{opt}$ *struct-body* ;$_{opt}$

A *struct-declaration* consists of an optional set of *attributes* (Section 17), followed by an optional set of *struct-modifiers* (Section 11.1.1), followed by the keyword `struct` and an *identifier* that names the struct, followed by an optional *struct-interfaces* specification (Section 11.1.2), followed by a *struct-body* (Section 11.1.3), optionally followed by a semicolon.

11.1.1 Struct modifiers

A *struct-declaration* may optionally include a sequence of struct modifiers:

struct-modifiers:
  struct-modifier
  struct-modifiers *struct-modifier*
struct-modifier:
  new
  public
  protected
  internal
  private

It is a compile-time error for the same modifier to appear multiple times in a struct declaration.

The modifiers of a struct declaration have the same meaning as those of a class declaration (Section 10.1.1).

11.1.2 Struct interfaces

A struct declaration may include a *struct-interfaces* specification, in which case the struct is said to implement the given interface types.

struct-interfaces:
  ; *interface-type-list*

Interface implementations are discussed further in Section 13.4.

11.1.3 Struct body

The *struct-body* of a struct defines the members of the struct.

struct-body:
  { *struct-member-declarations*$_{opt}$ }

11.2 Struct members

The members of a struct consist of the members introduced by its *struct-member-declaration*s and the members inherited from the type `System.ValueType`.

struct-member-declarations:
  struct-member-declaration
  struct-member-declarations struct-member-declaration
struct-member-declaration:
  constant-declaration
  field-declaration
  method-declaration
  property-declaration
  event-declaration
  indexer-declaration
  operator-declaration
  constructor-declaration
  static-constructor-declaration
  type-declaration

Except for the differences noted in Section 11.3, the descriptions of class members provided in Section 10.2 through Section 10.11 apply to struct members as well.

11.3 Class and struct differences

Structs differ from classes in several important ways:

- Structs are value types (Section 11.3.1).
- All struct types implicitly inherit from the class `System.ValueType` (Section 11.3.2).
- Assignment to a variable of a struct type creates a *copy* of the value being assigned (Section 11.3.3).
- The default value of a struct is the value produced by setting all value type fields to their default value and all reference type fields to `null` (Section 11.3.4).
- Boxing and unboxing operations are used to convert between a struct type and `object` (Section 11.3.5).
- The meaning of `this` is different for structs (Section 11.3.6).
- Instance field declarations for a struct are not permitted to include variable initializers (Section 11.3.7).
- A struct is not permitted to declare a parameterless instance constructor (Section 11.3.8).
- A struct is not permitted to declare a destructor (Section 11.3.9).

11.3.1 Value semantics

Structs are value types (Section 4.1) and are said to have value semantics. Classes, on the other hand, are reference types (Section 4.2) and are said to have reference semantics.

A variable of a struct type directly contains the data of the struct, whereas a variable of a class type contains a reference to the data, the latter known as an object.

With classes, it is possible for two variables to reference the same object, and thus possible for operations on one variable to affect the object referenced by the other variable. With structs, the variables each have their own copy of the data (except in the case of ref and out parameter variables), and it is not possible for operations on one to affect the other. Furthermore, because structs are not reference types, it is not possible for values of a struct type to be `null`.

Given the declaration

```
struct Point
{
   public int x, y;
   public Point(int x, int y) {
      this.x = x;
      this.y = y;
   }
}
```

the code fragment

```
Point a = new Point(10, 10);
Point b = a;
a.x = 100;
System.Console.WriteLine(b.x);
```

outputs the value `10`. The assignment of `a` to `b` creates a copy of the value, and `b` is thus unaffected by the assignment to `a.x`. Had `Point` instead been declared as a class, the output would be `100` because `a` and `b` would reference the same object.

11.3.2 Inheritance

All struct types implicitly inherit from the class `System.ValueType`, which, in turn, inherits from class `object`. A struct declaration may specify a list of implemented interfaces, but it is not possible for a struct declaration to specify a base class.

Struct types are never abstract and are always implicitly sealed. The `abstract` and `sealed` modifiers are therefore not permitted in a struct declaration.

Since inheritance is not supported for structs, the declared accessibility of a struct member cannot be `protected` or `protected internal`.

Function members in a struct cannot be `abstract` or `virtual`, and the `override` modifier is allowed only to override methods inherited from `System.ValueType`.

11.3.3 Assignment

Assignment to a variable of a struct type creates a *copy* of the value being assigned. This differs from assignment to a variable of a class type, which copies the reference but not the object identified by the reference.

Similar to an assignment, when a struct is passed as a value parameter or returned as the result of a function member, a copy of the struct is created. A struct may be passed by reference to a function member using a `ref` or `out` parameter.

When a property or indexer of a struct is the target of an assignment, the instance expression associated with the property or indexer access must be classified as a variable. If the instance expression is classified as a value, a compile-time error occurs. This is described in further detail in Section 7.13.1.

11.3.4 Default values

As described in Section 5.2, several kinds of variables are automatically initialized to their default value when they are created. For variables of class types and other reference types, this default value is `null`. However, since structs are value types that cannot be `null`, the default value of a struct is the value produced by setting all value type fields to their default value and all reference type fields to `null`.

Referring to the `Point` struct declared above, the example

```
Point[] a = new Point[100];
```

initializes each `Point` in the array to the value produced by setting the `x` and `y` fields to zero.

The default value of a struct corresponds to the value returned by the default constructor of the struct (Section 4.1.1). Unlike a class, a struct is not permitted to declare a parameterless instance constructor. Instead, every struct implicitly has a parameterless instance constructor, which always returns the value that results from setting all value type fields to their default value and all reference type fields to `null`.

Structs should be designed to consider the default initialization state a valid state. In the example

```
using System;
struct KeyValuePair
{
   string key;
   string value;
   public KeyValuePair(string key, string value) {
      if (key == null || value == null) throw new ArgumentException();
      this.key = key;
      this.value = value;
   }
}
```

the user-defined instance constructor protects against null values only where it is explicitly called. In cases where a `KeyValuePair` variable is subject to default value initialization, the `key` and `value` fields will be null, and the struct must be prepared to handle this state.

11.3.5 Boxing and unboxing

A value of a class type can be converted to type `object` or to an interface type that is implemented by the class simply by treating the reference as another type at compile-time. Likewise, a value of type `object` or a value of an interface type can be converted back to a class type without changing the reference (but of course a run-time type check is required in this case).

Since structs are not reference types, these operations are implemented differently for struct types. When a value of a struct type is converted to type `object` or to an interface type that is implemented by the struct, a boxing operation takes place. Likewise, when a value of type `object` or a value of an interface type is converted back to a struct type, an unboxing operation takes place. A key difference from the same operations on class types is that boxing and unboxing *copies* the struct value either into or out of the boxed instance. Thus, following a boxing or unboxing operation, changes made to the unboxed struct are not reflected in the boxed struct.

For further details on boxing and unboxing, see Section 4.3.

11.3.6 Meaning of this

Within an instance constructor or instance function member of a class, `this` is classified as a value. Thus, while `this` can be used to refer to the instance for which the function member was invoked, it is not possible to assign to `this` in a function member of a class.

Within an instance constructor of a struct, `this` corresponds to an `out` parameter of the struct type, and within an instance function member of a struct, `this` corresponds to a `ref` parameter of the struct type. In both cases, `this` is classified as a variable, and it is possible to modify the entire struct for which the function member was invoked by assigning to `this` or by passing this as a `ref` or `out` parameter.

11.3.7 Field initializers

As described in Section 11.3.4, the default value of a struct consists of the value that results from setting all value type fields to their default value and all reference type fields to `null`. For this reason, a struct does not permit instance field declarations to include variable initializers. This restriction applies only to instance fields. Static fields of a struct are permitted to include variable initializers.

The example

```
struct Point
{
   public int x = 1;  // Error, initializer not permitted
   public int y = 1;  // Error, initializer not permitted
}
```

is in error because the instance field declarations include variable initializers.

11.3.8 Constructors

Unlike a class, a struct is not permitted to declare a parameterless instance constructor. Instead, every struct implicitly has a parameterless instance constructor that always returns the value that results from setting all value type fields to their default value and all reference type fields to null (Section 4.1.1). A struct can declare instance constructors having parameters. For example

```
struct Point
{
   int x, y;
   public Point(int x, int y) {
      this.x = x;
      this.y = y;
   }
}
```

Given the above declaration, the statements

```
Point p1 = new Point();
Point p2 = new Point(0, 0);
```

both create a `Point` with `x` and `y` initialized to zero.

A struct instance constructor is not permitted to include a constructor initializer of the form `base(...)`.

If the struct instance constructor doesn't specify a constructor initializer, the `this` variable corresponds to an `out` parameter of the struct type, and similar to an `out` parameter, this must be definitely assigned (Section 5.3) at every location where the constructor returns. If the struct instance constructor specifies a constructor initializer, the `this` variable corresponds to a `ref` parameter of the struct type, and similar to a `ref` parameter, `this` is considered definitely assigned on entry to the constructor body. Consider the instance constructor implementation below:

```
struct Point
{
   int x, y;
   public int X {
      set { x = value; }
   }
   public int Y {
      set { y = value; }
   }
   public Point(int x, int y) {
      X = x;      // error, this is not yet definitely assigned
      Y = y;      // error, this is not yet definitely assigned
   }
}
```

No instance member function (including the set accessors for the properties `X` and `Y`) can be called until all fields of the struct being constructed have been definitely assigned. Note, however, that if `Point` were a class instead of a struct, the instance constructor implementation would be permitted.

11.3.9 Destructors

A struct is not permitted to declare a destructor.

11.4 Struct examples

The following shows two significant examples of using `struct` types to create types that can be used similarly to the built-in types of the language, but with modified semantics.

11.4.1 Database integer type

The `DBInt` struct below implements an integer type that can represent the complete set of values of the `int` type, plus an additional state that indicates an unknown value. A type with these characteristics is commonly used in databases.

```
using System;
public struct DBInt
{
   // The Null member represents an unknown DBInt value.
   public static readonly DBInt Null = new DBInt();
   // When the defined field is true, this DBInt represents a known value
   // which is stored in the value field. When the defined field is false,
   // this DBInt represents an unknown value, and the value field is 0.
   int value;
   bool defined;
   // Private instance constructor. Creates a DBInt with a known value.
   DBInt(int value) {
      this.value = value;
      this.defined = true;
   }
   // The IsNull property is true if this DBInt represents an unknown value.
   public bool IsNull { get { return !defined; } }
   // The Value property is the known value of this DBInt, or 0 if this
   // DBInt represents an unknown value.
   public int Value { get { return value; } }
   // Implicit conversion from int to DBInt.
   public static implicit operator DBInt(int x) {
      return new DBInt(x);
   }
   // Explicit conversion from DBInt to int. Throws an exception if the
   // given DBInt represents an unknown value.
   public static explicit operator int(DBInt x) {
      if (!x.defined) throw new InvalidOperationException();
      return x.value;
   }
   public static DBInt operator +(DBInt x) {
      return x;
   }
   public static DBInt operator -(DBInt x) {
      return x.defined ? -x.value : Null;
   }
   public static DBInt operator +(DBInt x, DBInt y) {
      return x.defined && y.defined? x.value + y.value: Null;
   }
   public static DBInt operator -(DBInt x, DBInt y) {
      return x.defined && y.defined? x.value - y.value: Null;
   }
   public static DBInt operator *(DBInt x, DBInt y) {
      return x.defined && y.defined? x.value * y.value: Null;
   }
   public static DBInt operator /(DBInt x, DBInt y) {
      return x.defined && y.defined? x.value / y.value: Null;
   }
   public static DBInt operator %(DBInt x, DBInt y) {
      return x.defined && y.defined? x.value % y.value: Null;
   }
   public static DBBool operator ==(DBInt x, DBInt y) {
      return x.defined && y.defined? x.value == y.value: DBBool.Null;
   }
   public static DBBool operator !=(DBInt x, DBInt y) {
      return x.defined && y.defined? x.value != y.value: DBBool.Null;
   }
   public static DBBool operator >(DBInt x, DBInt y) {
      return x.defined && y.defined? x.value > y.value: DBBool.Null;
   }
   public static DBBool operator <(DBInt x, DBInt y) {
```

```
        return x.defined && y.defined? x.value < y.value: DBBool.Null;
    }
    public static DBBool operator >=(DBInt x, DBInt y) {
        return x.defined && y.defined? x.value >= y.value: DBBool.Null;
    }
    public static DBBool operator <=(DBInt x, DBInt y) {
        return x.defined && y.defined? x.value <= y.value: DBBool.Null;
    }
    public override bool Equals(object obj) {
        if (!(obj is DBInt)) return false;
        DBInt x = (DBInt)obj;
        return value == x.value && defined == x.defined;
    }
    public override int GetHashCode() {
        return value;
    }
    public override string ToString() {
        return defined? value.ToString(): "DBInt.Null";
    }
}
```

11.4.2 Database boolean type

The `DBBool` struct below implements a three-valued logical type. The possible values of this type are `DBBool.True`, `DBBool.False`, and `DBBool.Null`, where the `Null` member indicates an unknown value. Such three-valued logical types are commonly used in databases.

```
using System;
public struct DBBool
{
   // The three possible DBBool values.
   public static readonly DBBool Null = new DBBool(0);
   public static readonly DBBool False = new DBBool(-1);
   public static readonly DBBool True = new DBBool(1);
   // Private field that stores –1, 0, 1 for False, Null, True.
   sbyte value;
   // Private instance constructor. The value parameter must be –1, 0, or 1.
   DBBool(int value) {
      this.value = (sbyte)value;
   }
   // Properties to examine the value of a DBBool. Return true if this
   // DBBool has the given value, false otherwise.
   public bool IsNull { get { return value == 0; } }
   public bool IsFalse { get { return value < 0; } }
   public bool IsTrue { get { return value > 0; } }
   // Implicit conversion from bool to DBBool. Maps true to DBBool.True and
   // false to DBBool.False.
   public static implicit operator DBBool(bool x) {
      return x? True: False;
   }
   // Explicit conversion from DBBool to bool. Throws an exception if the
   // given DBBool is Null, otherwise returns true or false.
   public static explicit operator bool(DBBool x) {
      if (x.value == 0) throw new InvalidOperationException();
      return x.value > 0;
   }
   // Equality operator. Returns Null if either operand is Null, otherwise
   // returns True or False.
   public static DBBool operator ==(DBBool x, DBBool y) {
      if (x.value == 0 || y.value == 0) return Null;
      return x.value == y.value? True: False;
   }
   // Inequality operator. Returns Null if either operand is Null, otherwise
   // returns True or False.
   public static DBBool operator !=(DBBool x, DBBool y) {
      if (x.value == 0 || y.value == 0) return Null;
      return x.value != y.value? True: False;
   }
   // Logical negation operator. Returns True if the operand is False, Null
   // if the operand is Null, or False if the operand is True.
   public static DBBool operator !(DBBool x) {
      return new DBBool(-x.value);
   }
   // Logical AND operator. Returns False if either operand is False,
   // otherwise Null if either operand is Null, otherwise True.
   public static DBBool operator &(DBBool x, DBBool y) {
      return new DBBool(x.value < y.value? x.value: y.value);
   }
   // Logical OR operator. Returns True if either operand is True, otherwise
   // Null if either operand is Null, otherwise False.
   public static DBBool operator |(DBBool x, DBBool y) {
      return new DBBool(x.value > y.value? x.value: y.value);
   }
   // Definitely true operator. Returns true if the operand is True, false
   // otherwise.
   public static bool operator true(DBBool x) {
      return x.value > 0;
```

```
    }
    // Definitely false operator. Returns true if the operand is False, false
    // otherwise.
    public static bool operator false(DBBool x) {
       return x.value < 0;
    }
    public override bool Equals(object obj) {
       if (!(obj is DBBool)) return false;
       return value == ((DBBool)obj).value;
    }
    public override int GetHashCode() {
       return value;
    }
    public override string ToString() {
       if (value > 0) return "DBBool.True";
       if (value < 0) return "DBBool.False";
       return "DBBool.Null";
    }
 }
```

12. Arrays

An array is a data structure that contains a number of variables, which are accessed through computed indices. The variables contained in an array, also called the elements of the array, are all of the same type, and this type is called the element type of the array.

An array has a rank that determines the number of indices associated with each array element. The rank of an array is also referred to as the dimensions of the array. An array with a rank of one is called a single-dimensional array. An array with a rank greater than one is called a multi-dimensional array. Specific sized multidimensional arrays are often referred to as two-dimensional arrays, three-dimensional arrays, and so on.

Each dimension of an array has an associated length that is an integral number greater than or equal to zero. The dimension lengths are not part of the type of the array, but rather are established when an instance of the array type is created at run-time. The length of a dimension determines the valid range of indices for that dimension: For a dimension of length `N`, indices can range from `0` to `N - 1` inclusive. The total number of elements in an array is the product of the lengths of each dimension in the array. If one or more of the dimensions of an array have a length of zero, the array is said to be empty.

The element type of an array can be any type, including an array type.

12.1 Array types

An array type is written as a *non-array-type* followed by one or more *rank-specifiers*:

array-type:
  non-array-type *rank-specifiers*
non-array-type:
  type
rank-specifiers:
  rank-specifier
  rank-specifiers *rank-specifier*
rank-specifier:
  [*dim-separators*$_{opt}$]
dim-separators:
  ,
  dim-separators ,

A *non-array-type* is any type that is not itself an *array-type*.

The rank of an array type is given by the leftmost *rank-specifier* in the *array-type*: A *rank-specifier* indicates that the array is an array with a rank of one plus the number of "`,`" tokens in the *rank-specifier*.

The element type of an array type is the type that results from deleting the leftmost *rank-specifier*:

- An array type of the form `T[R]` is an array with rank `R` and a non-array element type `T`.
- An array type of the form `T[R][R1]...[RN]` is an array with rank `R` and an element type `T[R1]...[RN]`.

In effect, the *rank-specifiers* are read from left to right before the final non-array element type. The type `int[][,,][,]` is a single-dimensional array of three-dimensional arrays of two-dimensional arrays of `int`.

At run-time, a value of an array type can be `null` or a reference to an instance of that array type.

12.1.1 The System.Array type

The type `System.Array` is the abstract base type of all array types. An implicit reference conversion (Section 6.1.4) exists from any array type to `System.Array`, and an explicit reference conversion (Section 6.2.3) exists from `System.Array` to any array type. Note that `System.Array` is not itself an *array-type*. Rather, it is a *class-type* from which all *array-type*s are derived.

At run-time, a value of type `System.Array` can be `null` or a reference to an instance of any array type.

12.2 Array creation

Array instances are created by *array-creation-expression*s (Section 7.5.10.2) or by field or local variable declarations that include an *array-initializer* (Section 12.6).

When an array instance is created, the rank and length of each dimension are established and then remain constant for the entire lifetime of the instance. In other words, it is not possible to change the rank of an existing array instance, nor is it possible to resize its dimensions.

An array instance is always of an array type. The `System.Array` type is an abstract type that cannot be instantiated.

Elements of arrays created by *array-creation-expression*s are always initialized to their default value (Section 5.2).

12.3 Array element access

Array elements are accessed using *element-access* expressions (Section 7.5.6.1) of the form `A[I1, I2, ..., IN]`, where `A` is an expression of an array type and each `IX` is an expression of type `int`, `uint`, `long`, `ulong`, or of a type that can be implicitly converted to one or more of these types. The result of an array element access is a variable, namely the array element selected by the indices.

The elements of an array can be enumerated using a `foreach` statement (Section 8.8.4).

12.4 Array members

Every array type inherits the members declared by the `System.Array` type.

12.5 Array covariance

For any two *reference-types* `A` and `B`, if an implicit reference conversion (Section 6.1.4) or explicit reference conversion (Section 6.2.3) exists from `A` to `B`, then the same reference conversion also exists from the array type `A[R]` to the array type `B[R]`, where `R` is any given *rank-specifier* (but the same for both array types). This relationship is known as array covariance. Array covariance in particular means that a value of an array type `A[R]` may actually be a reference to an instance of an array type `B[R]`, provided an implicit reference conversion exists from `B` to `A`.

Because of array covariance, assignments to elements of reference type arrays include a run-time check that ensures that the value being assigned to the array element is actually of a permitted type (Section 7.13.1). For example:

```
class Test
{
   static void Fill(object[] array, int index, int count, object value) {
      for (int i = index; i < index + count; i++) array[i] = value;
   }
   static void Main() {
      string[] strings = new string[100];
      Fill(strings, 0, 100, "Undefined");
      Fill(strings, 0, 10, null);
      Fill(strings, 90, 10, 0);
   }
}
```

The assignment to `array[i]` in the `Fill` method implicitly includes a run-time check that ensures that the object referenced by `value` is either `null` or an instance of a type that is compatible with the actual element type of `array`. In `Main`, the first two invocations of `Fill` succeed, but the third invocation causes a `System.ArrayTypeMismatchException` to be thrown upon executing the first assignment to `array[i]`. The exception occurs because a boxed `int` cannot be stored in a `string` array.

Array covariance specifically does not extend to arrays of *value-types*. For example, no conversion exists that permits an `int[]` to be treated as an `object[]`.

12.6 Array initializers

Array initializers may be specified in field declarations (Section 10.4), local variable declarations (Section 8.5.1), and array creation expressions (Section 7.5.10.2):

array-initializer:
  { *variable-initializer-list*$_{opt}$ }
  { *variable-initializer-list* , }
variable-initializer-list:
  variable-initializer
  variable-initializer-list , *variable-initializer*
variable-initializer:
  expression
  array-initializer

An array initializer consists of a sequence of variable initializers, enclosed by "`{`"and "`}`" tokens and separated by "`,`" tokens. Each variable initializer is an expression or, in the case of a multi-dimensional array, a nested array initializer.

The context in which an array initializer is used determines the type of the array being initialized. In an array creation expression, the array type immediately precedes the initializer. In a field or variable declaration, the array type is the type of the field or variable being declared. When an array initializer is used in a field or variable declaration, such as:

```
int[] a = {0, 2, 4, 6, 8};
```

it is simply shorthand for an equivalent array creation expression:

```
int[] a = new int[] {0, 2, 4, 6, 8};
```

For a single-dimensional array, the array initializer must consist of a sequence of expressions that are assignment compatible with the element type of the array. The expressions initialize array elements in increasing order, starting with the element at index zero. The number of expressions in the array initializer determines the length of the array instance being created. For example, the array initializer above creates an `int[]` instance of length 5 and then initializes the instance with the following values:

```
a[0] = 0; a[1] = 2; a[2] = 4; a[3] = 6; a[4] = 8;
```

For a multi-dimensional array, the array initializer must have as many levels of nesting as there are dimensions in the array. The outermost nesting level corresponds to the leftmost dimension and the innermost nesting level corresponds to the rightmost dimension. The length of each dimension of the array is determined by the number of elements at the corresponding nesting level in the array initializer. For each nested array initializer, the number of elements must be the same as the other array initializers at the same level. The example:

```
int[,] b = {{0, 1}, {2, 3}, {4, 5}, {6, 7}, {8, 9}};
```

creates a two-dimensional array with a length of five for the leftmost dimension and a length of two for the rightmost dimension:

```
int[,] b = new int[5, 2];
```

and then initializes the array instance with the following values:

```
b[0, 0] = 0; b[0, 1] = 1;
b[1, 0] = 2; b[1, 1] = 3;
b[2, 0] = 4; b[2, 1] = 5;
b[3, 0] = 6; b[3, 1] = 7;
b[4, 0] = 8; b[4, 1] = 9;
```

When an array creation expression includes both explicit dimension lengths and an array initializer, the lengths must be constant expressions and the number of elements at each nesting level must match the corresponding dimension length. Here are some examples:

```
int i = 3;
int[] x = new int[3] {0, 1, 2};      // OK
int[] y = new int[i] {0, 1, 2};      // Error, i not a constant
int[] z = new int[3] {0, 1, 2, 3};   // Error, length/initializer mismatch
```

Here, the initializer for `y` results in a compile-time error because the dimension length expression is not a constant, and the initializer for `z` results in a compile-time error because the length and the number of elements in the initializer do not agree.

13. Interfaces

An interface defines a contract. A class or struct that implements an interface must adhere to its contract. An interface may inherit from multiple base interfaces, and a class or struct may implement multiple interfaces.

Interfaces can contain methods, properties, events, and indexers. The interface itself does not provide implementations for the members that it defines. The interface merely specifies the members that must be supplied by classes or structs that implement the interface.

13.1 Interface declarations

An *interface-declaration* is a *type-declaration* (Section 9.5) that declares a new interface type.

interface-declaration:
  attributes$_{opt}$ *interface-modifiers*$_{opt}$ interface *identifier* *interface-base*$_{opt}$ *interface-body* ;$_{opt}$

An *interface-declaration* consists of an optional set of attributes (Section 17), followed by an optional set of *interface-modifiers* (Section 13.1.1), followed by the keyword `interface` and an *identifier* that names the interface, optionally followed by an optional *interface-base* specification (Section 13.1.2), followed by a *interface-body* (Section 13.1.3), optionally followed by a semicolon.

13.1.1 Interface modifiers

An *interface-declaration* may optionally include a sequence of interface modifiers:

interface-modifiers:
  interface-modifier
  interface-modifiers *interface-modifier*
interface-modifier:
  new
  public
  protected
  internal
  private

It is a compile-time error for the same modifier to appear multiple times in an interface declaration.

The `new` modifier is only permitted on interfaces defined within a class. It specifies that the interface hides an inherited member by the same name, as described in Section 10.2.2.

The `public`, `protected`, `internal`, and `private` modifiers control the accessibility of the interface. Depending on the context in which the interface declaration occurs, only some of these modifiers may be permitted (Section 3.5.1).

13.1.2 Base interfaces

An interface can inherit from zero or more interfaces, which are called the explicit base interfaces of the interface. When an interface has one or more explicit base interfaces, then in the declaration of that interface, the interface identifier is followed by a colon and a comma separated list of base interface identifiers.

interface-base:
  ; *interface-type-list*

The explicit base interfaces of an interface must be at least as accessible as the interface itself (Section 3.5.4). For example, it is a compile-time error to specify a `private` or `internal` interface in the *interface-base* of a `public` interface.

It is a compile-time error for an interface to directly or indirectly inherit from itself.

The base interfaces of an interface are the explicit base interfaces and their base interfaces. In other words, the set of base interfaces is the complete transitive closure of the explicit base interfaces, their explicit base interfaces, and so on. An interface inherits all members of its base interfaces. In the example

```
interface IControl
{
   void Paint();
}
interface ITextBox: IControl
{
   void SetText(string text);
}
interface IListBox: IControl
{
   void SetItems(string[] items);
}
interface IComboBox: ITextBox, IListBox {}
```

the base interfaces of `IComboBox` are `IControl`, `ITextBox`, and `IListBox`.

In other words, the `IComboBox` interface above inherits members `SetText` and `SetItems` as well as `Paint`.

A class or struct that implements an interface also implicitly implements all of the interface's base interfaces.

13.1.3 Interface body

The *interface-body* of an interface defines the members of the interface.

interface-body:
  { *interface-member-declarations*$_{opt}$ }

13.2 Interface members

The members of an interface are the members inherited from the base interfaces and the members declared by the interface itself.

interface-member-declarations:
  interface-member-declaration
  interface-member-declarations *interface-member-declaration*
interface-member-declaration:
  interface-method-declaration
  interface-property-declaration
  interface-event-declaration
  interface-indexer-declaration

An interface declaration may declare zero or more members. The members of an interface must be methods, properties, events, or indexers. An interface cannot contain constants, fields, operators, instance constructors, destructors, or types, nor can an interface contain static members of any kind.

All interface members implicitly have public access. It is a compile-time error for interface member declarations to include any modifiers. In particular, interfaces members cannot be declared with the modifiers `abstract`, `public`, `protected`, `internal`, `private`, `virtual`, `override`, or `static`.

The example

```
public delegate void StringListEvent(IStringList sender);
public interface IStringList
{
   void Add(string s);
   int Count { get; }
   event StringListEvent Changed;
   string this[int index] { get; set; }
}
```

declares an interface that contains one each of the possible kinds of members: A method, a property, an event, and an indexer.

An *interface-declaration* creates a new declaration space (Section 3.3), and the *interface-member-declarations* immediately contained by the *interface-declaration* introduce new members into this declaration space. The following rules apply to *interface-member-declarations*:

- The name of a method must differ from the names of all properties and events declared in the same interface. In addition, the signature (Section 3.6) of a method must differ from the signatures of all other methods declared in the same interface.
- The name of a property or event must differ from the names of all other members declared in the same interface.
- The signature of an indexer must differ from the signatures of all other indexers declared in the same interface.

The inherited members of an interface are specifically not part of the declaration space of the interface. Thus, an interface is allowed to declare a member with the same name or signature as an inherited member. When this occurs, the derived interface member is said to *hide* the base interface member. Hiding an inherited member is not considered an error, but it does cause the compiler to issue a warning. To suppress the warning, the declaration of the derived interface member must include a `new` modifier to indicate that the derived member is intended to hide the base member. This topic is discussed further in Section 3.7.1.2.

If a `new` modifier is included in a declaration that doesn't hide an inherited member, a warning is issued to that effect. This warning is suppressed by removing the `new` modifier.

13.2.1 Interface methods

Interface methods are declared using *interface-method-declarations*:

interface-method-declaration:
  attributes$_{opt}$ new$_{opt}$ *return-type* *identifier* (*formal-parameter-list*$_{opt}$) ;

The *attributes*, *return-type*, *identifier*, and *formal-parameter-list* of an interface method declaration have the same meaning as those of a method declaration in a class (Section 10.5). An interface method declaration is not permitted to specify a method body, and the declaration therefore always ends with a semicolon.

13.2.2 Interface properties

Interface properties are declared using *interface-property-declarations*:

interface-property-declaration:
  attributes$_{opt}$ new$_{opt}$ *type* *identifier* { *interface-accessors* }
interface-accessors:
  attributes$_{opt}$ get ;
  attributes$_{opt}$ set ;
  attributes$_{opt}$ get ; *attributes*$_{opt}$ set ;
  attributes$_{opt}$ set ; *attributes*$_{opt}$ get ;

The *attributes*, *type*, and *identifier* of an interface property declaration have the same meaning as those of a property declaration in a class (Section 10.6).

The accessors of an interface property declaration correspond to the accessors of a class property declaration (Section 10.6.2), except that the accessor body must always be a semicolon. Thus, the accessors simply indicate whether the property is read-write, read-only, or write-only.

13.2.3 Interface events

Interface events are declared using *interface-event-declarations*:

interface-event-declaration:
  attributes$_{opt}$ new$_{opt}$ event *type* *identifier* ;

The *attributes*, *type*, and *identifier* of an interface event declaration have the same meaning as those of an event declaration in a class (Section 10.7).

13.2.4 Interface indexers

Interface indexers are declared using *interface-indexer-declarations*:

interface-indexer-declaration:
  attributes$_{opt}$ new$_{opt}$ *type* this [*formal-parameter-list*] { *interface-accessors* }

The *attributes*, *type*, and *formal-parameter-list* of an interface indexer declaration have the same meaning as those of an indexer declaration in a class (Section 10.8).

The accessors of an interface indexer declaration correspond to the accessors of a class indexer declaration (Section 10.8), except that the accessor body must always be a semicolon. Thus, the accessors simply indicate whether the indexer is read-write, read-only, or write-only.

13.2.5 Interface member access

Interface members are accessed through member access (Section 7.5.4) and indexer access (Section 7.5.6.2) expressions of the form `I.M` and `I[A]`, where `I` is an interface type, `M` is a method, property, or event of that interface type, and `A` is an indexer argument list.

For interfaces that are strictly single-inheritance (each interface in the inheritance chain has exactly zero or one direct base interface), the effects of the member lookup (Section 7.3), method invocation (Section 7.5.5.1), and indexer access (Section 7.5.6.2) rules are exactly the same as for classes and structs: More derived members hide less derived members with the same name or signature. However, for multiple-inheritance interfaces, ambiguities can occur when two or more unrelated base interfaces declare members with the same name or signature. This section shows several examples of such situations. In all cases, explicit casts can be used to resolve the ambiguities.

In the example

```
interface IList
{
   int Count { get; set; }
}
interface ICounter
{
   void Count(int i);
}
interface IListCounter: IList, ICounter {}
class C
{
   void Test(IListCounter x) {
      x.Count(1);                  // Error
      x.Count = 1;               // Error
      ((IList)x).Count = 1;      // Ok, invokes IList.Count.set
      ((ICounter)x).Count(1);      // Ok, invokes ICounter.Count
   }
}
```

the first two statements cause compile-time errors because the member lookup (Section 7.3) of `Count` in `IListCounter` is ambiguous. As illustrated by the example, the ambiguity is resolved by casting `x` to the appropriate base interface type. Such casts have no run-time costs — they merely consist of viewing the instance as a less derived type at compile-time.

In the example

```
interface IInteger
{
   void Add(int i);
}
interface IDouble
{
   void Add(double d);
}
interface INumber: IInteger, IDouble {}
class C
{
   void Test(INumber n) {
      n.Add(1);               // Error, both Add methods are applicable
      n.Add(1.0);               // Ok, only IDouble.Add is applicable
      ((IInteger)n).Add(1);   // Ok, only IInteger.Add is a candidate
      ((IDouble)n).Add(1);      // Ok, only IDouble.Add is a candidate
   }
}
```

the invocation `n.Add(1)` is ambiguous because a method invocation (Section 7.5.5.1) requires all overloaded candidate methods to be declared in the same type. However, the invocation `n.Add(1.0)` is permitted because only `IDouble.Add` is applicable. When explicit casts are inserted, there is only one candidate method, and thus no ambiguity.

In the example

```
interface IBase
{
   void F(int i);
}
interface ILeft: IBase
{
   new void F(int i);
}
interface IRight: IBase
{
   void G();
}
interface IDerived: ILeft, IRight {}
class A
{
   void Test(IDerived d) {
      d.F(1);               // Invokes ILeft.F
      ((IBase)d).F(1);      // Invokes IBase.F
      ((ILeft)d).F(1);      // Invokes ILeft.F
      ((IRight)d).F(1);      // Invokes IBase.F
   }
}
```

the `IBase.F` member is hidden by the `ILeft.F` member. The invocation `d.F(1)` thus selects `ILeft.F`, even though `IBase.F` appears to not be hidden in the access path that leads through `IRight`.

The intuitive rule for hiding in multiple-inheritance interfaces is simply this: If a member is hidden in any access path, it is hidden in all access paths. Because the access path from `IDerived` to `ILeft` to `IBase` hides `IBase.F`, the member is also hidden in the access path from `IDerived` to `IRight` to `IBase`.

13.3 Fully qualified interface member names

An interface member is sometimes referred to by its fully qualified name. The fully qualified name of an interface member consists of the name of the interface in which the member is declared, followed by a dot, followed by the name of the member. The fully qualified name of a member references the interface in which the member is declared. For example, given the declarations

```
interface IControl
{
   void Paint();
}
interface ITextBox: IControl
{
   void SetText(string text);
}
```

the fully qualified name of `Paint` is `IControl.Paint` and the fully qualified name of `SetText` is `ITextBox.SetText`.

In the example above, it is not possible to refer to `Paint` as `ITextBox.Paint`.

When an interface is part of a namespace, the fully qualified name of an interface member includes the namespace name. For example

```
namespace System
{
   public interface ICloneable
   {
      object Clone();
   }
}
```

Here, the fully qualified name of the `Clone` method is `System.ICloneable.Clone`.

13.4 Interface implementations

Interfaces may be implemented by classes and structs. To indicate that a class or struct implements an interface, the interface identifier is included in the base class list of the class or struct. For example:

```
interface ICloneable
{
   object Clone();
}
interface IComparable
{
   int CompareTo(object other);
}
class ListEntry: ICloneable, IComparable
{
   public object Clone() {...}
   public int CompareTo(object other) {...}
}
```

A class or struct that implements an interface also implicitly implements all of the interface's base interfaces. This is true even if the class or struct doesn't explicitly list all base interfaces in the base class list. For example:

```
interface IControl
{
   void Paint();
}
interface ITextBox: IControl
{
   void SetText(string text);
}
class TextBox: ITextBox
{
   public void Paint() {...}
   public void SetText(string text) {...}
}
```

Here, class `TextBox` implements both `IControl` and `ITextBox`.

13.4.1 Explicit interface member implementations

For purposes of implementing interfaces, a class or struct may declare explicit interface member implementations. An explicit interface member implementation is a method, property, event, or indexer declaration that references a fully qualified interface member name. For example

```
interface ICloneable
{
   object Clone();
}
interface IComparable
{
   int CompareTo(object other);
}
class ListEntry: ICloneable, IComparable
{
   object ICloneable.Clone() {...}
   int IComparable.CompareTo(object other) {...}
}
```

Here, `ICloneable.Clone` and `IComparable.CompareTo` are explicit interface member implementations.

In some cases, the name of an interface member may not be appropriate for the implementing class, in which case the interface member may be implemented using explicit interface member implementation. A class implementing a file abstraction, for example, would likely implement a `Close` member function that has the effect of releasing the file resource, and implement the `Dispose` method of the `IDisposable` interface using explicit interface member implementation:

```
interface IDisposable
{
   void Dispose();
}
class MyFile: IDisposable
{
   void IDisposable.Dispose() {
      Close();
   }
   public void Close() {
      // Do what's necessary to close the file
      System.GC.SuppressFinalize(this);
   }
}
```

It is not possible to access an explicit interface member implementation through its fully qualified name in a method invocation, property access, or indexer access. An explicit interface member implementation can only be accessed through an interface instance, and is in that case referenced simply by its member name.

It is a compile-time error for an explicit interface member implementation to include access modifiers, and it is a compile-time error to include the modifiers `abstract`, `virtual`, `override`, or `static`.

Explicit interface member implementations have different accessibility characteristics than other members. Because explicit interface member implementations are never accessible through their fully qualified name in a method invocation or a property access, they are in a sense private. However, since they can be accessed through an interface instance, they are in a sense also public.

Explicit interface member implementations serve two primary purposes:

- Because explicit interface member implementations are not accessible through class or struct instances, they allow interface implementations to be excluded from the public interface of a class or struct. This is particularly useful when a class or struct implements an internal interface that is of no interest to a consumer of that class or struct.
- Explicit interface member implementations allow disambiguation of interface members with the same signature. Without explicit interface member implementations it would be impossible for a class or struct to have different implementations of interface members with the same signature and return type, as would it be impossible for a class or struct to have any implementation at all of interface members with the same signature but with different return types.

For an explicit interface member implementation to be valid, the class or struct must name an interface in its base class list that contains a member whose fully qualified name, type, and parameter types exactly match those of the explicit interface member implementation. Thus, in the following class

```
class Shape: ICloneable
{
   object ICloneable.Clone() {...}
   int IComparable.CompareTo(object other) {...}   // invalid
}
```

the declaration of `IComparable.CompareTo` results in a compile-time error because `IComparable` is not listed in the base class list of `Shape` and is not a base interface of `ICloneable`. Likewise, in the declarations

```
class Shape: ICloneable
{
   object ICloneable.Clone() {...}
}
class Ellipse: Shape
{
   object ICloneable.Clone() {...}   // invalid
}
```

the declaration of `ICloneable.Clone` in `Ellipse` results in a compile-time error because `ICloneable` is not explicitly listed in the base class list of `Ellipse`.

The fully qualified name of an interface member must reference the interface in which the member was declared. Thus, in the declarations

```
interface IControl
{
   void Paint();
}
interface ITextBox: IControl
{
   void SetText(string text);
}
class TextBox: ITextBox
{
   void IControl.Paint() {...}
   void ITextBox.SetText(string text) {...}
}
```

the explicit interface member implementation of `Paint` must be written as `IControl.Paint`.

13.4.2 Interface mapping

A class or struct must provide implementations of all members of the interfaces that are listed in the base class list of the class or struct. The process of locating implementations of interface members in an implementing class or struct is known as interface mapping.

Interface mapping for a class or struct `C` locates an implementation for each member of each interface specified in the base class list of `C`. The implementation of a particular interface member `I.M`, where `I` is the interface in which the member `M` is declared, is determined by examining each class or struct `S`, starting with `C` and repeating for each successive base class of `C`, until a match is located:

- If `S` contains a declaration of an explicit interface member implementation that matches `I` and `M`, then this member is the implementation of `I.M`.
- Otherwise, if `S` contains a declaration of a non-static public member that matches `M`, then this member is the implementation of `I.M`.

A compile-time error occurs if implementations cannot be located for all members of all interfaces specified in the base class list of `C`. Note that the members of an interface include those members that are inherited from base interfaces.

For purposes of interface mapping, a class member `A` matches an interface member `B` when:

- `A` and `B` are methods, and the name, type, and formal parameter lists of `A` and `B` are identical.
- `A` and `B` are properties, the name and type of `A` and `B` are identical, and `A` has the same accessors as `B` (`A` is permitted to have additional accessors if it is not an explicit interface member implementation).
- `A` and `B` are events, and the name and type of `A` and `B` are identical.
- `A` and `B` are indexers, the type and formal parameter lists of `A` and `B` are identical, and `A` has the same accessors as `B` (`A` is permitted to have additional accessors if it is not an explicit interface member implementation).

Notable implications of the interface mapping algorithm are:

- Explicit interface member implementations take precedence over other members in the same class or struct when determining the class or struct member that implements an interface member.
- Neither non-public nor static members participate in interface mapping.

In the example

```
interface ICloneable
{
   object Clone();
}
class C: ICloneable
{
   object ICloneable.Clone() {...}
   public object Clone() {...}
}
```

the `ICloneable.Clone` member of `C` becomes the implementation of `Clone` in `ICloneable` because explicit interface member implementations take precedence over other members.

If a class or struct implements two or more interfaces containing a member with the same name, type, and parameter types, it is possible to map each of those interface members onto a single class or struct member. For example

```
interface IControl
{
   void Paint();
}
interface IForm
{
   void Paint();
}
class Page: IControl, IForm
{
```

```
   public void Paint() {...}
}
```

Here, the `Paint` methods of both `IControl` and `IForm` are mapped onto the `Paint` method in `Page`. It is of course also possible to have separate explicit interface member implementations for the two methods.

If a class or struct implements an interface that contains hidden members, then some members must necessarily be implemented through explicit interface member implementations. For example

```
interface IBase
{
   int P { get; }
}
interface IDerived: IBase
{
   new int P();
}
```

An implementation of this interface would require at least one explicit interface member implementation, and would take one of the following forms

```
class C: IDerived
{
   int IBase.P { get {...} }
   int IDerived.P() {...}
}
class C: IDerived
{
   public int P { get {...} }
   int IDerived.P() {...}
}
class C: IDerived
{
   int IBase.P { get {...} }
   public int P() {...}
}
```

When a class implements multiple interfaces that have the same base interface, there can be only one implementation of the base interface. In the example

```
interface IControl
{
   void Paint();
}
interface ITextBox: IControl
{
   void SetText(string text);
}
interface IListBox: IControl
{
   void SetItems(string[] items);
}
class ComboBox: IControl, ITextBox, IListBox
{
   void IControl.Paint() {...}
   void ITextBox.SetText(string text) {...}
   void IListBox.SetItems(string[] items) {...}
}
```

it is not possible to have separate implementations for the `IControl` named in the base class list, the `IControl` inherited by `ITextBox`, and the `IControl` inherited by `IListBox`. Indeed, there is no notion of a separate identity for these interfaces. Rather, the implementations of `ITextBox` and `IListBox` share the same implementation of `IControl`, and `ComboBox` is simply considered to implement three interfaces, `IControl`, `ITextBox`, and `IListBox`.

The members of a base class participate in interface mapping. In the example

```
interface Interface1
{
   void F();
}
class Class1
{
   public void F() {}
   public void G() {}
}
class Class2: Class1, Interface1
{
   new public void G() {}
}
```

the method `F` in `Class1` is used in `Class2`'s implementation of `Interface1`.

13.4.3 Interface implementation inheritance

A class inherits all interface implementations provided by its base classes.

Without explicitly re-implementing an interface, a derived class cannot in any way alter the interface mappings it inherits from its base classes. For example, in the declarations

```
interface IControl
{
   void Paint();
}
class Control: IControl
{
   public void Paint() {...}
}
class TextBox: Control
{
   new public void Paint() {...}
}
```

the `Paint` method in `TextBox` hides the `Paint` method in `Control`, but it does not alter the mapping of `Control.Paint` onto `IControl.Paint`, and calls to `Paint` through class instances and interface instances will have the following effects

```
Control c = new Control();
TextBox t = new TextBox();
IControl ic = c;
IControl it = t;
c.Paint();         // invokes Control.Paint();
t.Paint();         // invokes TextBox.Paint();
ic.Paint();         // invokes Control.Paint();
it.Paint();         // invokes Control.Paint();
```

However, when an interface method is mapped onto a virtual method in a class, it is possible for derived classes to override the virtual method and alter the implementation of the interface. For example, rewriting the declarations above to

```
interface IControl
{
   void Paint();
}
class Control: IControl
{
   public virtual void Paint() {...}
}
class TextBox: Control
{
   public override void Paint() {...}
}
```

the following effects will now be observed

```
Control c = new Control();
TextBox t = new TextBox();
IControl ic = c;
IControl it = t;
c.Paint();         // invokes Control.Paint();
t.Paint();         // invokes TextBox.Paint();
ic.Paint();         // invokes Control.Paint();
it.Paint();         // invokes TextBox.Paint();
```

Since explicit interface member implementations cannot be declared virtual, it is not possible to override an explicit interface member implementation. However, it is perfectly valid for an explicit interface member implementation to call another method, and that other method can be declared virtual to allow derived classes to override it. For example

```
interface IControl
{
   void Paint();
}
class Control: IControl
{
   void IControl.Paint() { PaintControl(); }
   protected virtual void PaintControl() {...}
}
class TextBox: Control
{
   protected override void PaintControl() {...}
}
```

Here, classes derived from `Control` can specialize the implementation of `IControl.Paint` by overriding the `PaintControl` method.

13.4.4 Interface re-implementation

A class that inherits an interface implementation is permitted to re-implement the interface by including it in the base class list.

A re-implementation of an interface follows exactly the same interface mapping rules as an initial implementation of an interface. Thus, the inherited interface mapping has no effect whatsoever on the interface mapping established for the re-implementation of the interface. For example, in the declarations

```
interface IControl
{
   void Paint();
}
class Control: IControl
{
   void IControl.Paint() {...}
}
class MyControl: Control, IControl
{
   public void Paint() {}
}
```

the fact that `Control` maps `IControl.Paint` onto `Control.IControl.Paint` doesn't affect the re-implementation in `MyControl`, which maps `IControl.Paint` onto `MyControl.Paint`.

Inherited public member declarations and inherited explicit interface member declarations participate in the interface mapping process for re-implemented interfaces. For example

```
interface IMethods
{
   void F();
   void G();
   void H();
   void I();
}
class Base: IMethods
{
   void IMethods.F() {}
   void IMethods.G() {}
   public void H() {}
   public void I() {}
}
class Derived: Base, IMethods
{
   public void F() {}
   void IMethods.H() {}
}
```

Here, the implementation of `IMethods` in `Derived` maps the interface methods onto `Derived.F`, `Base.IMethods.G`, `Derived.IMethods.H`, and `Base.I`.

When a class implements an interface, it implicitly also implements all of that interface's base interfaces. Likewise, a re-implementation of an interface is also implicitly a re-implementation of all of the interface's base interfaces. For example

```
interface IBase
{
   void F();
}
interface IDerived: IBase
{
   void G();
}
class C: IDerived
{
   void IBase.F() {...}
```

```
    void IDerived.G() {...}
}
class D: C, IDerived
{
    public void F() {...}
    public void G() {...}
}
```

Here, the re-implementation of `IDerived` also re-implements `IBase`, mapping `IBase.F` onto `D.F`.

13.4.5 Abstract classes and interfaces

Like a non-abstract class, an abstract class must provide implementations of all members of the interfaces that are listed in the base class list of the class. However, an abstract class is permitted to map interface methods onto abstract methods. For example

```
interface IMethods
{
   void F();
   void G();
}
abstract class C: IMethods
{
   public abstract void F();
   public abstract void G();
}
```

Here, the implementation of `IMethods` maps `F` and `G` onto abstract methods, which must be overridden in non-abstract classes that derive from `C`.

Note that explicit interface member implementations cannot be abstract, but explicit interface member implementations are of course permitted to call abstract methods. For example

```
interface IMethods
{
   void F();
   void G();
}
abstract class C: IMethods
{
   void IMethods.F() { FF(); }
   void IMethods.G() { GG(); }
   protected abstract void FF();
   protected abstract void GG();
}
```

Here, non-abstract classes that derive from `C` would be required to override `FF` and `GG`, thus providing the actual implementation of `IMethods`.

14. Enums

An enum type is a distinct value type (Section 4.1) that declares a set of named constants.

The example

```
enum Color
{
   Red,
   Green,
   Blue
}
```

declares an enum type named `Color` with members `Red`, `Green`, and `Blue`.

14.1 Enum declarations

An enum declaration declares a new enum type. An enum declaration begins with the keyword `enum`, and defines the name, accessibility, underlying type, and members of the enum.

enum-declaration:
  attributes$_{opt}$ *enum-modifiers*$_{opt}$ enum *identifier* *enum-base*$_{opt}$ *enum-body* ;$_{opt}$
enum-base:
  ; *integral-type*
enum-body:
  { *enum-member-declarations*$_{opt}$ }
  { *enum-member-declarations* , }

Each enum type has a corresponding integral type called the underlying type of the enum type. This underlying type must be able to represent all the enumerator values defined in the enumeration. An enum declaration may explicitly declare an underlying type of `byte`, `sbyte`, `short`, `ushort`, `int`, `uint`, `long` or `ulong`. Note that `char` cannot be used as an underlying type. An enum declaration that does not explicitly declare an underlying type has an underlying type of `int`.

The example

```
enum Color: long
{
   Red,
   Green,
   Blue
}
```

declares an enum with an underlying type of `long`. A developer might choose to use an underlying type of `long`, as in the example, to enable the use of values that are in the range of `long` but not in the range of `int`, or to preserve this option for the future.

14.2 Enum modifiers

An *enum-declaration* may optionally include a sequence of enum modifiers:

enum-modifiers:
  enum-modifier
  enum-modifiers *enum-modifier*
enum-modifier:
  new
  public
  protected
  internal
  private

It is a compile-time error for the same modifier to appear multiple times in an enum declaration.

The modifiers of an enum declaration have the same meaning as those of a class declaration (Section 10.1.1). Note, however, that the `abstract` and `sealed` modifiers are not permitted in an enum declaration. Enums cannot be abstract and do not permit derivation.

14.3 Enum members

The body of an enum type declaration defines zero or more enum members, which are the named constants of the enum type. No two enum members can have the same name.

enum-member-declarations:
  enum-member-declaration
  enum-member-declarations , *enum-member-declaration*
enum-member-declaration:
  $attributes_{opt}$ *identifier*
  $attributes_{opt}$ *identifier* = *constant-expression*

Each enum member has an associated constant value. The type of this value is the underlying type for the containing enum. The constant value for each enum member must be in the range of the underlying type for the enum. The example

```
enum Color: uint
{
   Red = -1,
   Green = -2,
   Blue = -3
}
```

results in a compile-time error because the constant values `-1`, `-2`, and `-3` are not in the range of the underlying integral type `uint`.

Multiple enum members may share the same associated value. The example

```
enum Color
{
   Red,
   Green,
   Blue,
   Max = Blue
}
```

shows an enum in which two enum members — `Blue` and `Max` — have the same associated value.

The associated value of an enum member is assigned either implicitly or explicitly. If the declaration of the enum member has a *constant-expression* initializer, the value of that constant expression, implicitly converted to the underlying type of the enum, is the associated value of the enum member. If the declaration of the enum member has no initializer, its associated value is set implicitly, as follows:

- If the enum member is the first enum member declared in the enum type, its associated value is zero.
- Otherwise, the associated value of the enum member is obtained by increasing the associated value of the textually preceding enum member by one. This increased value must be within the range of values that can be represented by the underlying type; otherwise, a compile-time error occurs.

The example

```
using System;
enum Color
{
   Red,
   Green = 10,
   Blue
}
class Test
{
   static void Main() {
      Console.WriteLine(StringFromColor(Color.Red));
      Console.WriteLine(StringFromColor(Color.Green));
      Console.WriteLine(StringFromColor(Color.Blue));
   }
   static string StringFromColor(Color c) {
```

```
        switch (c) {
           case Color.Red:
              return String.Format("Red = {0}", (int) c);
           case Color.Green:
              return String.Format("Green = {0}", (int) c);
           case Color.Blue:
              return String.Format("Blue = {0}", (int) c);
           default:
              return "Invalid color";
        }
     }
  }
```

prints out the enum member names and their associated values. The output is:

```
Red = 0
Green = 10
Blue = 11
```

for the following reasons:

- The enum member `Red` is automatically assigned the value zero (since it has no initializer and is the first enum member).
- The enum member `Green` is explicitly given the value `10`.
- The enum member `Blue` is automatically assigned the value one greater than the member that textually precedes it.

The associated value of an enum member may not, directly or indirectly, use the value of its own associated enum member. Other than this circularity restriction, enum member initializers may freely refer to other enum member initializers, regardless of their textual position. Within an enum member initializer, values of other enum members are always treated as having the type of their underlying type, so that casts are not necessary when referring to other enum members.

The example

```
enum Circular
{
   A = B,
   B
}
```

results in a compile-time error because the declarations of `A` and `B` are circular. `A` depends on `B` explicitly, and `B` depends on `A` implicitly.

Enum members are named and scoped in a manner exactly analogous to fields within classes. The scope of an enum member is the body of its containing enum type. Within that scope, enum members can be referred to by their simple name. From all other code, the name of an enum member must be qualified with the name of its enum type. Enum members do not have any declared accessibility — an enum member is accessible if its containing enum type is accessible.

14.4 The System.Enum type

The type `System.Enum` is the abstract base class of all enum types (this is distinct and different from the underlying type of the enum type), and the members inherited from `System.Enum` are available in any enum type. A boxing conversion (Section 4.3.1) exists from any enum type to `System.Enum`, and an unboxing conversion (Section 4.3.2) exists from `System.Enum` to any enum type.

Note that `System.Enum` is not itself an *enum-type*. Rather, it is a *class-type* from which all *enum-type*s are derived. The type `System.Enum` inherits from the type `System.ValueType` (Section 4.1.1), which, in turn, inherits from type `object`. At run-time, a value of type `System.Enum` can be `null` or a reference to a boxed value of any enum type.

14.5 Enum values and operations

Each enum type defines a distinct type; an explicit enumeration conversion (Section 6.2.2) is required to convert between an enum type and an integral type, or between two enum types. The set of values that an enum type can take on is not limited by its enum members. In particular, any value of the underlying type of an enum can be cast to the enum type, and is a distinct valid value of that enum type.

Enum members have the type of their containing enum type (except within other enum member initializers: see Section 14.3). The value of an enum member declared in enum type `E` with associated value `v` is `(E)v`.

The following operators can be used on values of enum types: `==`, `!=`, `<`, `>`, `<=`, `>=` (Section 7.9.5), `+` (Section 7.7.4), `-` (Section 7.7.5), `^`, `&`, `|` (Section 7.10.2), `~` (Section 7.6.4), `++`, `--` (Section 7.5.9 and Section 7.6.5), `sizeof` (Section A.5.4).

15. Delegates

Delegates enable scenarios that other languages — such as C++, Pascal, and Modula — have addressed with function pointers. Unlike C++ function pointers, however, delegates are fully object oriented, and unlike C++ pointers to member functions, delegates encapsulate both an object instance and a method.

A delegate declaration defines a class that is derived from the class `System.Delegate`. A delegate instance encapsulates an invocation list, which is a list one or more methods, each of which is referred to as a callable entity. For instance methods, a callable entity consists of an instance and a method on that instance. For static methods, a callable entity consists of just a method. Invoking a delegate instance with an appropriate set of arguments causes each of the delegate's callable entities to be invoked with the given set of arguments.

An interesting and useful property of a delegate instance is that it does not know or care about the classes of the methods it encapsulates; all that matters is that those methods be compatible (Section 15.1) with the delegate's type. This makes delegates perfectly suited for "anonymous" invocation.

15.1 Delegate declarations

A *delegate-declaration* is a *type-declaration* (Section 9.5) that declares a new delegate type.

delegate-declaration:
  attributes$_{opt}$ *delegate-modifiers*$_{opt}$ delegate *return-type* *identifier* (*formal-parameter-list*$_{opt}$) ;
delegate-modifiers:
  delegate-modifier
  delegate-modifiers *delegate-modifier*
delegate-modifier:
  new
  public
  protected
  internal
  private

It is a compile-time error for the same modifier to appear multiple times in a delegate declaration.

The `new` modifier is only permitted on delegates declared within another type, in which case it specifies that such a delegate hides an inherited member by the same name, as described in Section 10.2.2.

The `public`, `protected`, `internal`, and `private` modifiers control the accessibility of the delegate type. Depending on the context in which the delegate declaration occurs, some of these modifiers may not be permitted (Section 3.5.1).

The delegate's type name is *identifier*.

The optional *formal-parameter-list* specifies the parameters of the delegate, and *return-type* indicates the return type of the delegate. A method and a delegate type are compatible if both of the following are true:

- They have the same number or parameters, with the same types, in the same order, with the same parameter modifiers.
- Their return types are the same.

Delegate types in C# are name equivalent, not structurally equivalent. Specifically, two different delegate types that have the same parameter lists and return type are considered different delegate types. However, instances of two distinct but structurally equivalent delegate types may compare as equal (Section 7.9.8)).

For example:

```
delegate int D1(int i, double d);
class A
{
   public static int M1(int a, double b) {...}
}
class B
{
   delegate int D2(int c, double d);
   public static int M1(int f, double g) {...}
   public static void M2(int k, double l) {...}
   public static int M3(int g) {...}
   public static void M4(int g) {...}
}
```

The delegate types `D1` and `D2` are both compatible with the methods `A.M1` and `B.M1`, since they have the same return type and parameter list; however, these delegate types are two different types, so they are not interchangeable. The delegate types `D1` and `D2` are incompatible with the methods `B.M2`, `B.M3`, and `B.M4`, since they have different return types or parameter lists.

The only way to declare a delegate type is via a *delegate-declaration*. A delegate type is a class type that is derived from `System.Delegate`. Delegate types are implicitly `sealed`, so it is not permissible to derive any type from a delegate type. It is also not permissible to derive a non-delegate class type from `System.Delegate`. Note that `System.Delegate` is not itself a delegate type; it is a class type from which all delegate types are derived.

C# provides special syntax for delegate instantiation and invocation. Except for instantiation, any operation that can be applied to a class or class instance can also be applied to a delegate class or instance, respectively. In particular, it is possible to access members of the `System.Delegate` type via the usual member access syntax.

The set of methods encapsulated by a delegate instance is called an invocation list. When a delegate instance is created

(Section 15.2) from a single method, it encapsulates that method, and its invocation list contains only one entry. However, when two non-null delegate instances are combined, their invocation lists are concatenated — in the order left operand then right operand — to form a new invocation list, which contains two or more entries.

Delegates are combined using the binary + (Section 7.7.4) and += operators (Section 7.13.2). A delegate can be removed from a combination of delegates, using the binary - (Section 7.7.5) and -= operators (Section 7.13.2). Delegates can be compared for equality (Section 7.9.8).

The following example shows the instantiation of a number of delegates, and their corresponding invocation lists:

```
delegate void D(int x);
class C
{
   public static void M1(int i) {...}
   public static void M2(int i) {...}
}
class Test
{
   static void Main() {
      D cd1 = new D(C.M1);      // M1
      D cd2 = new D(C.M2);      // M2
      D cd3 = cd1 + cd2;      // M1 + M2
      D cd4 = cd3 + cd1;       // M1 + M2 + M1
      D cd5 = cd4 + cd3;       // M1 + M2 + M1 + M1 + M2
   }
}
```

When `cd1` and `cd2` are instantiated, they each encapsulate one method. When `cd3` is instantiated, it has an invocation list of two methods, `M1` and `M2`, in that order. `cd4`'s invocation list contains `M1`, `M2`, and `M1`, in that order. Finally, `cd5`'s invocation list contains `M1`, `M2`, `M1`, `M1`, and `M2`, in that order. For more examples of combining (as well as removing) delegates, see Section 15.3.

15.2 Delegate instantiation

An instance of a delegate is created by a *delegate-creation-expression* (Section 7.5.10.3). The newly created delegate instance then refers to either:

- The static method referenced in the *delegate-creation-expression*, or
- The target object (which cannot be `null`) and instance method referenced in the *delegate-creation-expression*, or
- Another delegate

For example:

```
delegate void D(int x);
class C
{
   public static void M1(int i) {...}
   public void M2(int i) {...}
}
class Test
{
   static void Main() {
      D cd1 = new D(C.M1);      // static method
      Test t = new C();
      D cd2 = new D(t.M2);      // instance method
      D cd3 = new D(cd2);      // another delegate
   }
}
```

Once instantiated, delegate instances always refer to the same target object and method. Remember, when two delegates are combined, or one is removed from another, a new delegate results with its own invocation list; the invocation lists of the delegates combined or removed remain unchanged.

15.3 Delegate invocation

C# provides special syntax for invoking a delegate. When a non-null delegate instance whose invocation list contains one entry is invoked, it invokes the one method with the same arguments it was given, and returns the same value as the referred to method. See Section 7.5.5.2 for detailed information on delegate invocation. If an exception occurs during the invocation of such a delegate, and that exception is not caught within the method that was invoked, the search for an exception catch clause continues in the method that called the delegate, as if that method had directly called the method to which that delegate referred.

Invocation of a delegate instance whose invocation list contains multiple entries proceeds by invoking each of the methods in the invocation list, synchronously, in order. Each method so called is passed the same set of arguments as was given to the delegate instance. If such a delegate invocation includes reference parameters (Section 10.5.1.2), each method invocation will occur with a reference to the same variable; changes to that variable by one method in the invocation list will be visible to methods further down the invocation list. If the delegate invocation includes output parameters or a return value, their final value will come from the invocation of the last delegate in the list.

If an exception occurs during processing of the invocation of such a delegate, and that exception is not caught within the method that was invoked, the search for an exception catch clause continues in the method that called the delegate, and any methods further down the invocation list are not invoked.

Attempting to invoke a delegate instance whose value is null results in an exception of type `System.NullReferenceException`.

The following example shows how to instantiate, combine, remove, and invoke delegates:

```
using System;
delegate void D(int x);
class C
{
   public static void M1(int i) {
      Console.WriteLine("C.M1: " + i);
   }
   public static void M2(int i) {
      Console.WriteLine("C.M2: " + i);
   }
   public void M3(int i) {
      Console.WriteLine("C.M3: " + i);
   }
}
class Test
{
   static void Main() {
      D cd1 = new D(C.M1);
      cd1(-1);            // call M1
      D cd2 = new D(C.M2);
      cd2(-2);            // call M2
      D cd3 = cd1 + cd2;
      cd3(10);            // call M1 then M2
      cd3 += cd1;
      cd3(20);            // call M1, M2, then M1
      C c = new C();
      D cd4 = new D(c.M3);
      cd3 += cd4;
      cd3(30);            // call M1, M2, M1, then M3
      cd3 -= cd1;         // remove last M1
      cd3(40);            // call M1, M2, then M3
      cd3 -= cd4;
      cd3(50);            // call M1 then M2
      cd3 -= cd2;
      cd3(60);            // call M1
      cd3 -= cd2;         // impossible removal is benign
      cd3(60);            // call M1
      cd3 -= cd1;         // invocation list is empty so cd3 is null
      //      cd3(70);      // System.NullReferenceException thrown
      cd3 -= cd1;         // impossible removal is benign
   }
}
```

As shown in the statement `cd3 += cd1;`, a delegate can be present in an invocation list multiple times. In this case, it is simply invoked once per occurrence. In an invocation list such as this, when that delegate is removed, the last occurrence in the invocation list is the one actually removed.

Immediately prior to the execution of the final statement, `cd3 -= cd1;`, the delegate `cd3` refers to an empty invocation list. Attempting to remove a delegate from an empty list (or to remove a non-existent delegate from a non-empty list) is not an error.

The output produced is:

```
C.M1: -1
C.M2: -2
C.M1: 10
C.M2: 10
C.M1: 20
C.M2: 20
C.M1: 20
C.M1: 30
C.M2: 30
C.M1: 30
C.M3: 30
C.M1: 40
C.M2: 40
C.M3: 40
C.M1: 50
C.M2: 50
C.M1: 60
C.M1: 60
```

16. Exceptions

Exceptions in C# provide a structured, uniform, and type-safe way of handling both system level and application level error conditions. The exception mechanism in C# is quite similar to that of C++, with a few important differences:

- In C#, all exceptions must be represented by an instance of a class type derived from `System.Exception`. In C++, any value of any type can be used to represent an exception.
- In C#, a finally block (Section 8.10) can be used to write termination code that executes in both normal execution and exceptional conditions. Such code is difficult to write in C++ without duplicating code.
- In C#, system-level exceptions such as overflow, divide-by-zero, and null dereferences have well defined exception classes and are on a par with application-level error conditions.

16.1 Causes of exceptions

Exception can be thrown in two different ways.

- A `throw` statement (Section 8.9.5) throws an exception immediately and unconditionally. Control never reaches the statement immediately following the `throw`.
- Certain exceptional conditions that arise during the processing of C# statements and expression cause an exception in certain circumstances when the operation cannot be completed normally. For example, an integer division operation (Section 7.7.2) throws a `System.DivideByZeroException` if the denominator is zero. See Section 16.4 for a list of the various exceptions that can occur in this way.

16.2 The System.Exception class

The `System.Exception` class is the base type of all exceptions. This class has a few notable properties that all exceptions share:

- `Message` is a read-only property of type `string` that contains a human-readable description of the reason for the exception.
- `InnerException` is a read-only property of type `Exception`. If its value is non-`null`, it refers to the exception that caused the current exception — that is, the current exception was raised in a catch block handling the `InnerException`. Otherwise, its value is null, indicating that this exception was not caused by another exception. The number of exception objects chained together in this manner can be arbitrary.

The value of these properties can be specified in calls to the instance constructor for `System.Exception`.

16.3 How exceptions are handled

Exceptions are handled by a `try` statement (Section 8.10).

When an exception occurs, the system searches for the nearest `catch` clause that can handle the exception, as determined by the run-time type of the exception. First, the current method is searched for a lexically enclosing `try` statement, and the associated catch clauses of the try statement are considered in order. If that fails, the method that called the current method is searched for a lexically enclosing `try` statement that encloses the point of the call to the current method. This search continues until a `catch` clause is found that can handle the current exception, by naming an exception class that is of the same class, or a base class, of the run-time type of the exception being thrown. A `catch` clause that doesn't name an exception class can handle any exception.

Once a matching catch clause is found, the system prepares to transfer control to the first statement of the catch clause. Before execution of the catch clause begins, the system first executes, in order, any `finally` clauses that were associated with try statements more nested that than the one that caught the exception.

If no matching catch clause is found, one of two things occurs:

- If the search for a matching catch clause reaches a static constructor (Section 10.11) or static field initializer, then a `System.TypeInitializationException` is thrown at the point that triggered the invocation of the static constructor. The inner exception of the `System.TypeInitializationException` contains the exception that was originally thrown.
- If the search for matching catch clauses reaches the code that initially started the thread, then execution of the thread is terminated. The impact of such termination is implementation-defined.

Exceptions that occur during destructor execution are worth special mention. If an exception occurs during destructor execution, and that exception is not caught, then the execution of that destructor is terminated and the destructor of the base class (if any) is called. If there is no base class (as in the case of the `object` type) or if there is no base class destructor, then the exception is discarded.

16.4 Common Exception Classes

The following exceptions are thrown by certain C# operations.

`System.ArithmeticException`	A base class for exceptions that occur during arithmetic operations, such as `System.DivideByZeroException` and `System.OverflowException`.
`System.ArrayTypeMismatchException`	Thrown when a store into an array fails because the actual type of the stored element is incompatible with the actual type of the array.
`System.DivideByZeroException`	Thrown when an attempt to divide an integral value by zero occurs.
`System.IndexOutOfRangeException`	Thrown when an attempt to index an array via an index that is less than zero or outside the bounds of the array.
`System.InvalidCastException`	Thrown when an explicit conversion from a base type or interface to a derived type fails at run time.
`System.NullReferenceException`	Thrown when a `null` reference is used in a way that causes the referenced object to be required.
`System.OutOfMemoryException`	Thrown when an attempt to allocate memory (via `new`) fails.
`System.OverflowException`	Thrown when an arithmetic operation in a `checked` context overflows.
`System.StackOverflowException`	Thrown when the execution stack is exhausted by having too many pending method calls; typically indicative of very deep or unbounded recursion.
`System.TypeInitializationException`	Thrown when a static constructor throws an exception, and no catch clauses exists to catch it.

17. Attributes

Much of the C# language enables the programmer to specify declarative information about the entities defined in the program. For example, the accessibility of a method in a class is specified by decorating it with the *method-modifiers* `public`, `protected`, `internal`, and `private`.

C# enables programmers to invent new kinds of declarative information, called attributes. Programmers can then attach attributes to various program entities, and retrieve attribute information in a run-time environment. For instance, a framework might define a `HelpAttribute` attribute that can be placed on certain program elements (such as classes and methods) to provide a mapping from those program elements to their documentation.

Attributes are defined through the declaration of attribute classes (Section 17.1), which may have positional and named parameters (Section 17.1.2). Attributes are attached to entities in a C# program using attribute specifications (Section 17.2), and can be retrieved at run-time as attribute instances (Section 17.3).

17.1 Attribute classes

A class that derives from the abstract class `System.Attribute`, whether directly or indirectly, is an attribute class. The declaration of an attribute class defines a new kind of attribute that can be placed on a declaration. By convention, attribute classes are named with a suffix of `Attribute`. Uses of an attribute may either include or omit this suffix.

17.1.1 Attribute usage

The attribute `AttributeUsage` (Section 17.4.1) is used to describe how an attribute class can be used.

`AttributeUsage` has a positional parameter (Section 17.1.2) that enables an attribute class to specify the kinds of declarations on which it can be used. The example

```
using System;
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Interface)]
public class SimpleAttribute: Attribute
{
   ...
}
```

defines an attribute class named `SimpleAttribute` that can be placed on *class-declarations* and *interface-declarations* only. The example

```
[Simple] class Class1 {...}
[Simple] interface Interface1 {...}
```

shows several uses of the `Simple` attribute. Although this attribute is defined with the name `SimpleAttribute`, when this attribute is used, the `Attribute` suffix may be omitted, resulting in the short name `Simple`. Thus, the example above is semantically equivalent to the following:

```
[SimpleAttribute] class Class1 {...}
[SimpleAttribute] interface Interface1 {...}
```

`AttributeUsage` has a named parameter (Section 17.1.2) called `AllowMultiple`, which indicates whether the attribute can be specified more than once for a given entity. If `AllowMultiple` for an attribute class is true, then that attribute class is a multi-use attribute class, and can be specified more than once on an entity. If `AllowMultiple` for an attribute class is false or it is unspecified, then that attribute class is a single-use attribute class, and can be specified at most once on an entity.

The example

```
using System;
[AttributeUsage(AttributeTargets.Class, AllowMultiple = true)]
public class AuthorAttribute: Attribute
{
   private string name;
   public AuthorAttribute(string name) {
      this.name = name;
   }
   public string Name {
      get { return name; }
   }
}
```

defines a multi-use attribute class named `AuthorAttribute`. The example

```
[Author("Brian Kernighan"), Author("Dennis Ritchie")]
class Class1
{
   ...
}
```

shows a class declaration with two uses of the `Author` attribute.

`AttributeUsage` has another named parameter called `Inherited`, which indicates whether the attribute, when specified on a base class, is also inherited by classes that derive from that base class. If `Inherited` for an attribute class is true, then that attribute is inherited. If `Inherited` for an attribute class is false or it is unspecified, then that attribute is not inherited.

An attribute class `X` not having an `AttributeUsage` attribute attached to it, as in

```
using System;
class X: Attribute {...}
```

is equivalent to the following:

```
using System;
[AttributeUsage(
   AttributeTargets.All,
   AllowMultiple = false,
   Inherited = true)
]
class X: Attribute {...}
```

17.1.2 Positional and named parameters

Attribute classes can have positional parameters and named parameters. Each public instance constructor for an attribute class defines a valid sequence of positional parameters for that attribute class. Each non-static public read-write field and property for an attribute class defines a named parameter for the attribute class.

The example

```
using System;
[AttributeUsage(AttributeTargets.Class)]
public class HelpAttribute: Attribute
{
   public HelpAttribute(string url) {      // Positional parameter
      ...
   }
   public string Topic {                  // Named parameter
      get {...}
      set {...}
   }
   public string Url {
      get {...}
   }
```

```
}
```

defines an attribute class named `HelpAttribute` that has one positional parameter, `url`, and one named parameter, `Topic`. Although it is non-static and public, the property `Url` does not define a named parameter, since it is not read-write.

This attribute class might be used as follows:

```
[Help("http://www.example.com/.../Class1.htm")]
class Class1
{
   ...
}
[Help("http://www.example.com/.../Misc.htm", Topic = "Class2")]
class Class2
{
   ...
}
```

17.1.3 Attribute parameter types

The types of positional and named parameters for an attribute class are limited to the attribute parameter types, which are:

- One of the following types: `bool`, `byte`, `char`, `double`, `float`, `int`, `long`, `short`, `string`.
- The type `object`.
- The type `System.Type`.
- An enum type, provided it has public accessibility and the types in which it is nested (if any) also have public accessibility (Section 17.2).
- Single-dimensional arrays of the above types.

17.2 Attribute specification

Attribute specification is the application of a previously defined attribute to a declaration. An attribute is a piece of additional declarative information that is specified for a declaration. Attributes can be specified at global scope (to specify attributes on the containing assembly or module) and for *type-declarations* (Section 9.5), *class-member-declarations* (Section 10.2), *interface-member-declarations* (Section 13.2), *struct-member-declarations* (Section 11.2), *enum-member-declarations* (Section 14.3), *accessor-declarations* for properties (Section 10.6.2), *event-accessor-declarations* (Section 10.7.1), and *formal-parameter-lists* (Section 10.5.1).

Attributes are specified in attribute sections. An attribute section consists of a pair of square brackets, which surround a comma-separated list of one or more attributes. The order in which attributes are specified in such a list, and the order in which sections attached to the same program entity are arranged, is not significant. For instance, the attribute specifications `[A][B]`, `[B][A]`, `[A, B]`, and `[B, A]` are equivalent.

global-attributes:
  global-attribute-sections
global-attribute-sections:
  global-attribute-section

  global-attribute-sections *global-attribute-section*

global-attribute-section:
  [*global-attribute-target-specifier* *attribute-list*]

  [*global-attribute-target-specifier* *attribute-list* ,]

global-attribute-target-specifier:
  global-attribute-target :
global-attribute-target:
  assembly

  module

attributes:
  attribute-sections
attribute-sections:
  attribute-section

  attribute-sections *attribute-section*

attribute-section:
  [*attribute-target-specifier*$_{opt}$ *attribute-list*]

  [*attribute-target-specifier*$_{opt}$ *attribute-list* ,]

attribute-target-specifier:
  attribute-target :
attribute-target:
  field

  event

  method

  param

  property

  return

  type

attribute-list:
  attribute

  attribute-list , *attribute*

attribute:
  attribute-name *attribute-arguments*$_{opt}$

attribute-name:
  type-name
attribute-arguments:
  (*positional-argument-list*$_{opt}$)

  (*positional-argument-list* , *named-argument-list*)

  (*named-argument-list*)

positional-argument-list:
  positional-argument

  positional-argument-list , *positional-argument*

positional-argument:
  attribute-argument-expression
named-argument-list:
  named-argument

  named-argument-list , *named-argument*

named-argument:
  identifier = *attribute-argument-expression*
attribute-argument-expression:
  expression

An attribute consists of an *attribute-name* and an optional list of positional and named arguments. The positional arguments (if any) precede the named arguments. A positional argument consists of an *attribute-argument-expression*; a named argument consists of a name, followed by an equal sign, followed by an *attribute-argument-expression*, which, together, are constrained by the same rules as simple assignment. The order of named arguments is not significant.

The *attribute-name* identifies an attribute class. If the form of *attribute-name* is *type-name* then this name must refer to an attribute class. Otherwise, a compile-time error occurs. The example

```
class Class1 {}
[Class1] class Class2 {}   // Error
```

results in a compile-time error because it attempts to use `Class1` as an attribute class when `Class1` is not an attribute class.

Certain contexts permit the specification of an attribute on more than one target. A program can explicitly specify the target by including an *attribute-target-specifier*. When an attribute is placed at the global level, a *global-attribute-target-specifier* is required. In all other locations, a reasonable default is applied, but an *attribute-target-specifier* can be used to affirm or override the default in certain ambiguous cases (or to just affirm the default in non-ambiguous cases). Thus, typically, *attribute-target-specifier*s can be omitted except at the global level. The potentially ambiguous contexts are resolved as follows:

- An attribute specified at global scope can apply either to the target assembly or the target module. No default exists for this context, so an *attribute-target-specifier* is always required in this context. The presence of the `assembly` *attribute-target-specifier* indicates that the attribute applies to the target assembly; the presence of the `module` *attribute-target-specifier* indicates that the attribute applies to the target module.
- An attribute specified on a delegate declaration can apply either to the delegate being declared or to its return value. In the absence of an *attribute-target-specifier*, the attribute applies to the delegate. The presence of the `type` *attribute-target-specifier* indicates that the attribute applies to the delegate; the presence of the `return` *attribute-target-specifier* indicates that the attribute applies to the return value.
- An attribute specified on a method declaration can apply either to the method being declared or to its return value. In the absence of an *attribute-target-specifier*, the attribute applies to the method. The presence of the `method` *attribute-target-specifier* indicates that the attribute applies to the method; the presence of the `return` *attribute-target-specifier* indicates that the attribute applies to the return value.
- An attribute specified on an operator declaration can apply either to the operator being declared or to its return value. In the absence of an *attribute-target-specifier*, the attribute applies to the operator. The presence of the `method` *attribute-target-specifier* indicates that the attribute applies to the operator; the presence of the `return` *attribute-target-specifier* indicates that the attribute applies to the return value.
- An attribute specified on an event declaration that omits event accessors can apply to the event being declared, to the associated field (if the event is not abstract), or to the associated add and remove methods. In the absence of an *attribute-target-specifier*, the attribute applies to the event. The presence of the `event` *attribute-target-specifier* indicates that the

attribute applies to the event; the presence of the `field` *attribute-target-specifier* indicates that the attribute applies to the field; and the presence of the `method` *attribute-target-specifier* indicates that the attribute applies to the methods.

- An attribute specified on a get accessor declaration for a property or indexer declaration can apply either to the associated method or to its return value. In the absence of an *attribute-target-specifier*, the attribute applies to the method. The presence of the `method` *attribute-target-specifier* indicates that the attribute applies to the method; the presence of the `return` *attribute-target-specifier* indicates that the attribute applies to the return value.
- An attribute specified on a set accessor for a property or indexer declaration can apply either to the associated method or to its lone implicit parameter. In the absence of an *attribute-target-specifier*, the attribute applies to the method. The presence of the `method` *attribute-target-specifier* indicates that the attribute applies to the method; the presence of the `param` *attribute-target-specifier* indicates that the attribute applies to the parameter.
- An attribute specified on an add or remove accessor declaration for an event declaration can apply either to the associated method or to its lone parameter. In the absence of an *attribute-target-specifier*, the attribute applies to the method. The presence of the `method` *attribute-target-specifier* indicates that the attribute applies to the method; the presence of the `param` *attribute-target-specifier* indicates that the attribute applies to the parameter.

In other contexts, inclusion of an *attribute-target-specifier* is permitted but unnecessary. For instance, a class declaration may either include or omit the specifier type:

```
[type: Author("Brian Kernighan")]
class Class1 {}
[Author("Dennis Ritchie")]
class Class2 {}
```

It is an error to specify an invalid *attribute-target-specifier*. For instance, the specifier param cannot be used on a class declaration:

```
[param: Author("Brian Kernighan")]      // Error
class Class1 {}
```

By convention, attribute classes are named with a suffix of `Attribute`. An *attribute-name* of the form *type-name* may either include or omit this suffix. If an attribute class is found both with and without this suffix, an ambiguity is present, and a compile-time error results. If the *attribute-name* is spelled such that its right-most identifier is a verbatim identifier (Section 2.4.2), then only an attribute without a suffix is matched, thus enabling such an ambiguity to be resolved. The example

```
using System;
[AttributeUsage(AttributeTargets.All)]
public class X: Attribute
{}
[AttributeUsage(AttributeTargets.All)]
public class XAttribute: Attribute
{}
[X]                  // Error: ambiguity
class Class1 {}
[XAttribute]         // Refers to XAttribute
class Class2 {}
[@X]                  // Refers to X
class Class3 {}
[@XAttribute]         // Refers to XAttribute
class Class4 {}
```

shows two attribute classes named `X` and `XAttribute`. The attribute `[X]` is ambiguous, since it could refer to either `X` or `XAttribute`. Using a verbatim identifier allows the exact intent to be specified in such rare cases. The attribute `[XAttribute]` is not ambiguous (although it would be if there was an attribute class named `XAttributeAttribute`!). If the declaration for class `X` is removed, then both attributes refer to the attribute class named `XAttribute`, as follows:

```
using System;
[AttributeUsage(AttributeTargets.All)]
```

```
public class XAttribute: Attribute
{}
[X]                  // Refers to XAttribute
class Class1 {}
[XAttribute]         // Refers to XAttribute
class Class2 {}
[@X]                  // Error: no attribute named "X"
class Class3 {}
```

It is a compile-time error to use a single-use attribute class more than once on the same entity. The example

```
using System;
[AttributeUsage(AttributeTargets.Class)]
public class HelpStringAttribute: Attribute
{
   string value;
   public HelpStringAttribute(string value) {
      this.value = value;
   }
   public string Value {
      get {...}
   }
}
[HelpString("Description of Class1")]
[HelpString("Another description of Class1")]
public class Class1 {}
```

results in a compile-time error because it attempts to use `HelpString`, which is a single-use attribute class, more than once on the declaration of `Class1`.

An expression `E` is an *attribute-argument-expression* if all of the following statements are true:

- The type of `E` is an attribute parameter type (Section 17.1.3).
- At compile-time, the value of `E` can be resolved to one of the following:
    - A constant value.
    - A `System.Type` object.
    - A one-dimensional array of *attribute-argument-expressions*.

For example:

```
using System;
[AttributeUsage(AttributeTargets.Class)]
public class TestAttribute: Attribute
{
   public int P1 {
      get {...}
      set {...}
   }
   public Type P2 {
      get {...}
      set {...}
   }
   public object P3 {
      get {...}
      set {...}
   }
}
[Test(P1 = 1234, P3 = new int[] {1, 3, 5}, P2 = typeof(float))]
class MyClass {}
```

17.3 Attribute instances

An attribute instance is an instance that represents an attribute at run-time. An attribute is defined with an attribute class, positional arguments, and named arguments. An attribute instance is an instance of the attribute class that is initialized with the positional and named arguments.

Retrieval of an attribute instance involves both compile-time and run-time processing, as described in the following sections.

17.3.1 Compilation of an attribute

The compilation of an *attribute* with attribute class `T`, *positional-argument-list* `P` and *named-argument-list* `N`, consists of the following steps:

- Follow the compile-time processing steps for compiling an object-creation-expression of the form new `T(P)`. These steps either result in a compile-time error or determine an instance constructor `C` on `T` that can be invoked at run-time.
- If `C` does not have public accessibility, then a compile-time error occurs.
- For each *named-argument* `Arg` in `N`:
  - Let `Name` be the *identifier* of the *named-argument* `Arg`.
  - `Name` must identify a non-static read-write public field or property on `T`. If `T` has no such field or property, then a compile-time error occurs.
- Keep the following information for run-time instantiation of the attribute: the attribute class `T`, the instance constructor `C` on `T`, the *positional-argument-list* `P` and the *named-argument-list* `N`.

17.3.2 Run-time retrieval of an attribute instance

Compilation of an *attribute* yields an attribute class `T`, an instance constructor `C` on T, a *positional-argument-list* `P`, and a *named-argument-list* `N`. Given this information, an attribute instance can be retrieved at run-time using the following steps:

- Follow the run-time processing steps for executing an *object-creation-expression* of the form `new T(P)`, using the instance constructor `C` as determined at compile-time. These steps either result in an exception, or produce an instance `O` of `T`.
- For each *named-argument* `Arg` in `N, in order`:
  - Let `Name` be the *identifier* of the *named-argument* `Arg`. If `Name` does not identify a non-static public read-write field or property on `O`, then an exception is thrown.
  - Let `Value` be the result of evaluating the *attribute-argument-expression* of `Arg`.
  - If `Name` identifies a field on `O`, then set this field to `Value`.
  - Otherwise, `Name` identifies a property on `O`. Set this property to `Value`.
  - The result is `O`, an instance of the attribute class `T` that has been initialized with the *positional-argument-list* `P` and the *named-argument-list* `N`.

17.4 Reserved attributes

A small number of attributes affect the language in some way. These attributes include:

- `System.AttributeUsageAttribute` (Section 17.4.1), which is used to describe the ways in which an attribute class can be used.
- `System.Diagnostics.ConditionalAttribute` (Section 17.4.2), which is used to define conditional methods.
- `System.ObsoleteAttribute` (Section 17.4.3), which is used to mark a member as obsolete.

17.4.1 The AttributeUsage attribute

The attribute `AttributeUsage` is used to describe the manner in which the attribute class can be used.

A class that is decorated with the `AttributeUsage` attribute must derive from `System.Attribute`, either directly or indirectly. Otherwise, a compile-time error occurs.

```
namespace System
{
   [AttributeUsage(AttributeTargets.Class)]
   public class AttributeUsageAttribute: Attribute
   {
      public AttributeUsageAttribute(AttributeTargets validOn) {...}
      public virtual bool AllowMultiple { get {...} set {...} }
      public virtual bool Inherited { get {...} set {...} }
      public virtual AttributeTargets ValidOn { get {...} }
   }
   public enum AttributeTargets
   {
      Assembly    = 0x0001,
      Module       = 0x0002,
      Class       = 0x0004,
      Struct       = 0x0008,
      Enum          = 0x0010,
      Constructor = 0x0020,
      Method       = 0x0040,
      Property    = 0x0080,
      Field       = 0x0100,
      Event       = 0x0200,
      Interface    = 0x0400,
      Parameter    = 0x0800,
      Delegate    = 0x1000,
      ReturnValue = 0x2000,
      All = Assembly | Module | Class | Struct | Enum | Constructor |
         Method | Property | Field | Event | Interface | Parameter |
         Delegate | ReturnValue
   }
}
```

17.4.2 The Conditional attribute

The attribute `Conditional` enables the definition of conditional methods. The `Conditional` attribute indicates a condition by testing a conditional compilation symbol. Calls to a conditional method are either included or omitted depending on whether this symbol is defined at the point of the call. If the symbol is defined, the call is included; otherwise, the call (including evaluation of the parameters of the call) is omitted.

```
namespace System.Diagnostics
{
   [AttributeUsage(AttributeTargets.Method, AllowMultiple = true)]
   public class ConditionalAttribute: Attribute
   {
      public ConditionalAttribute(string conditionString) {...}
      public string ConditionString { get {...} }
   }
}
```

A conditional method is subject to the following restrictions:

- The conditional method must be a method in a class or struct declaration. A compile-time error occurs if the `Conditional` attribute is specified on a method in an interface declaration.
- The conditional method must have a return type of `void`.
- The conditional method must not be marked with the `override` modifier. A conditional method may be marked with the `virtual` modifier, however. Overrides of such a method are implicitly conditional, and must not be explicitly marked with a `Conditional` attribute.
- The conditional method must not be an implementation of an interface method. Otherwise, a compile-time error occurs.

In addition, a compile-time error occurs if a conditional method is used in a *delegate-creation-expression*. The example

```
#define DEBUG
using System;
using System.Diagnostics;
class Class1
{
   [Conditional("DEBUG")]
   public static void M() {
      Console.WriteLine("Executed Class1.M");
   }
}
class Class2
{
   public static void Test() {
      Class1.M();
   }
}
```

declares `Class1.M` as a conditional method. `Class2`'s `Test` method calls this method. Since the conditional compilation symbol `DEBUG` is defined, if `Class2.Test` is called, it will call `M`. If the symbol `DEBUG` had not been defined, then `Class2.Test` would not call `Class1.M`.

It is important to note that the inclusion or exclusion of a call to a conditional method is controlled by the conditional compilation symbols at the point of the call. In the example

File `class1.cs`:

```
using System.Diagnostics;
class Class1
{
   [Conditional("DEBUG")]
   public static void F() {
      Console.WriteLine("Executed Class1.F");
   }
```

```
}
```

File `class2.cs`:

```
#define DEBUG
class Class2
{
   public static void G() {
      Class1.F();            // F is called
   }
}
```

File `class3.cs`:

```
#undef DEBUG
class Class3
{
   public static void H() {
      Class1.F();            // F is not called
   }
}
```

the classes `Class2` and `Class3` each contain calls to the conditional method `Class1.F`, which is conditional based on whether or not `DEBUG` is defined. Since this symbol is defined in the context of `Class2` but not `Class3`, the call to `F` in `Class2` is included, while the call to F in `Class3` is omitted.

The use of conditional methods in an inheritance chain can be confusing. Calls made to a conditional method through `base`, of the form `base.M`, are subject to the normal conditional method call rules. In the example

File `class1.cs`:

```
using System;
using System.Diagnostics;
class Class1
{
   [Conditional("DEBUG")]
   public virtual void M() {
      Console.WriteLine("Class1.M executed");
   }
}
```

File `class2.cs`:

```
using System;
class Class2: Class1
{
   public override void M() {
      Console.WriteLine("Class2.M executed");
      base.M();                  // base.M is not called!
   }
}
```

File `class3.cs`:

```
#define DEBUG
using System;
class Class3
{
   public static void Test() {
      Class2 c = new Class2();
      c.M();                     // M is called
   }
}
```

`Class2` includes a call to the `M` defined in its base class. This call is omitted because the base method is conditional based on the presence of the symbol `DEBUG`, which is undefined. Thus, the method writes to the console "`Class2.M executed`" only. Judicious use of *pp-declaration*s can eliminate such problems.

17.4.3 The Obsolete attribute

The attribute `Obsolete` is used to mark types and members of types that should no longer be used.

```
namespace System
{
   [AttributeUsage(
      AttributeTargets.Class |
      AttributeTargets.Struct |
       AttributeTargets.Enum |
      AttributeTargets.Interface |
      AttributeTargets.Delegate |
      AttributeTargets.Method |
      AttributeTargets.Constructor |
      AttributeTargets.Property |
      AttributeTargets.Field |
      AttributeTargets.Event,
      Inherited = false)
   ]
   public class ObsoleteAttribute: Attribute
   {
      public ObsoleteAttribute() {...}
      public ObsoleteAttribute(string message) {...}
      public ObsoleteAttribute(string message, bool error) {...}
      public string Message { get {...} }
      public bool IsError { get {...} }
   }
}
```

If a program uses a type or member that is decorated with the `Obsolete` attribute, the compiler issues a warning or an error. Specifically, the compiler issues a warning if no error parameter is provided, or if the error parameter is provided and has the value false. The compiler issues an error if the error parameter is specified and has the value true.

In the example

```
[Obsolete("This class is obsolete; use class B instead")]
class A
{
   public void F() {}
}
class B
{
   public void F() {}
}
class Test
{
   static void Main() {
      A a = new A();       // Warning
      a.F();
   }
}
```

the class `A` is decorated with the `Obsolete` attribute. Each use of `A` in `Main` results in a warning that includes the specified message, "This class is obsolete; use class B instead."

17.5 Attributes for Interoperation

Note This section is applicable only to the Microsoft .NET implementation of C#.

17.5.1 Interoperation with COM and Win32 components

The .NET runtime provides a large number of attributes that enable C# programs to interoperate with components written using COM and Win32 DLLs. For example, the `DllImport` attribute can be used on a `static extern` method to indicate that the implementation of the method is to be found in a Win32 DLL. These attributes are found in the `System.Runtime.InteropServices` namespace, and detailed documentation for these attributes is found in the .NET runtime documentation.

17.5.2 Interoperation with other .NET languages

The following section describes the `IndexerName` attribute.

17.5.2.1 The IndexerName attribute

Indexers are implemented in .NET using indexed properties, and have a name in the .NET metadata. If no `IndexerName` attribute is present for an indexer, then the name `Item` is used by default. The `IndexerName` attribute enables a developer to override this default and specify a different name.

```
namespace System.Runtime.CompilerServices.CSharp
{
   [AttributeUsage(AttributeTargets.Property)]
   public class IndexerNameAttribute: Attribute
   {
      public IndexerNameAttribute(string indexerName) {...}
      public string Value { get {...} }
   }
}
```

A. Unsafe code

The core C# language, as defined in the preceding chapters, differs notably from C and C++ in its omission of pointers as a data type. Instead, C# provides references and the ability to create objects that are managed by a garbage collector. This design, coupled with other features, makes C# a much safer language than C or C++. In the core C# language it is simply not possible to have an uninitialized variable, a "dangling" pointer, or an expression that indexes an array beyond its bounds. Whole categories of bugs that routinely plague C and C++ programs are thus eliminated.

While practically every pointer type construct in C or C++ has a reference type counterpart in C#, nonetheless, there are situations where access to pointer types becomes a necessity. For example, interfacing with the underlying operating system, accessing a memory-mapped device, or implementing a time-critical algorithm may not be possible or practical without access to pointers. To address this need, C# provides the ability to write unsafe code.

In unsafe code it is possible to declare and operate on pointers, to perform conversions between pointers and integral types, to take the address of variables, and so forth. In a sense, writing unsafe code is much like writing C code within a C# program.

Unsafe code is in fact a "safe" feature from the perspective of both developers and users. Unsafe code must be clearly marked with the modifier `unsafe`, so developers can't possibly use unsafe features accidentally, and the execution engine works to ensure that unsafe code cannot be executed in an untrusted environment.

A.1 Unsafe contexts

The unsafe features of C# are available only in *unsafe contexts*. An unsafe context is introduced by including an `unsafe` modifier in the declaration of a type or member, or by employing an *unsafe-statement*:

- A declaration of a class, struct, interface, or delegate may include an `unsafe` modifier, in which case the entire textual extent of that type declaration (including the body of the class, struct, or interface) is considered an unsafe context.
- A declaration of a field, method, property, event, indexer, operator, instance constructor, destructor, or static constructor may include an `unsafe` modifier, in which case the entire textual extent of that member declaration is considered an unsafe context.
- An *unsafe-statement* enables the use of an unsafe context within a *block*. The entire textual extent of the associated *block* is considered an unsafe context.

The associated grammar extensions are shown below. For brevity, ellipses (...) are used to represent productions that appear in preceding chapters.

class-modifier:
  ...
  unsafe
struct-modifier:
  ...
  unsafe
interface-modifier:
  ...
  unsafe
delegate-modifier:
  ...
  unsafe
field-modifier:
  ...
  unsafe
method-modifier:
  ...
  unsafe
property-modifier:
  ...
  unsafe
event-modifier:
  ...
  unsafe
indexer-modifier:
  ...
  unsafe
operator-modifier:
  ...
  unsafe
constructor-modifier:
  ...
  unsafe
destructor-declaration:
  attributes$_{opt}$ extern$_{opt}$ unsafe$_{opt}$ ~ *identifier* ( ) *destructor-body*
  attributes$_{opt}$ unsafe$_{opt}$ extern$_{opt}$ ~ *identifier* ( ) *destructor-body*
static-constructor-declaration:
  attributes$_{opt}$ extern$_{opt}$ unsafe$_{opt}$ static *identifier* ( ) *static-constructor-body*
  attributes$_{opt}$ unsafe$_{opt}$ extern$_{opt}$ static *identifier* ( ) *static-constructor-body*
embedded-statement:
  ...
  unsafe-statement
unsafe-statement:
  unsafe *block*

In the example

```
public unsafe struct Node
{
   public int Value;
   public Node* Left;
   public Node* Right;
}
```

the `unsafe` modifier specified in the struct declaration causes the entire textual extent of the struct declaration to become an unsafe context. Thus, it is possible to declare the `Left` and `Right` fields to be of a pointer type. The example above could also be written

```
public struct Node
{
   public int Value;
   public unsafe Node* Left;
   public unsafe Node* Right;
}
```

Here, the `unsafe` modifiers in the field declarations cause those declarations to be considered unsafe contexts.

Other than establishing an unsafe context, thus permitting the use of pointer types, the `unsafe` modifier has no effect on a type or a member. In the example

```
public class A
{
   public unsafe virtual void F() {
      char* p;
      ...
   }
}
public class B: A
{
   public override void F() {
      base.F();
      ...
   }
}
```

the `unsafe` modifier on the `F` method in `A` simply causes the textual extent of `F` to become an unsafe context in which the unsafe features of the language can be used. In the override of `F` in `B`, there is no need to re-specify the `unsafe` modifier — unless, of course, the `F` method in `B` itself needs access to unsafe features.

The situation is slightly different when a pointer type is part of the method's signature

```
public unsafe class A
{
   public virtual void F(char* p) {...}
}
public class B: A
{
   public unsafe override void F(char* p) {...}
}
```

Here, because `F`'s signature includes a pointer type, it can only be written in an unsafe context. However, the unsafe context can be introduced by either making the entire class unsafe, as is the case in `A`, or by including an `unsafe` modifier in the method declaration, as is the case in `B`.

A.2 Pointer types

In an unsafe context, a *type* (Section 4) may be a *pointer-type* as well as a *value-type* or a *reference-type*.

type:
  value-type
  reference-type
  pointer-type

A *pointer-type* is written as an *unmanaged-type* or the keyword `void`, followed by a `*` token:

pointer-type:
  unmanaged-type `*`
  void `*`
unmanaged-type:
  type

The type specified before the `*` in a pointer type is called the referent type of the pointer type. It represents the type of the variable to which a value of the pointer type points.

Unlike references (values of reference types), pointers are not tracked by the garbage collector — the garbage collector has no knowledge of pointers and the data to which they point. For this reason a pointer is not permitted to point to a reference or to a struct that contains references, and the referent type of a pointer must be an *unmanaged-type*.

An *unmanaged-type* is any type that isn't a *reference-type* and doesn't contain *reference-type* fields at any level of nesting. In other words, an *unmanaged-type* is one of the following:

- `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, `decimal`, or `bool`.
- Any *enum-type*.
- Any *pointer-type*.
- Any user-defined *struct-type* that contains fields of *unmanaged-types* only.

The intuitive rule for mixing of pointers and references is that referents of references (objects) are permitted to contain pointers, but referents of pointers are not permitted to contain references.

Some examples of pointer types are given in the table below:

Example	Description
`byte*`	Pointer to `byte`
`char*`	Pointer to `char`
`int**`	Pointer to pointer to `int`
`int*[]`	Single-dimensional array of pointers to `int`
`void*`	Pointer to unknown type

For a given implementation, all pointer types must have the same size and representation.

Unlike C and C++, when multiple pointers are declared in the same declaration, in C# the `*` is written along with the underlying type only, not as a prefix punctuator on each pointer name. For example

```
int* pi, pj;   // NOT as int *pi, *pj;
```

The value of a pointer having type `T*` represents the *address* of a variable of type `T`. The pointer indirection operator `*` (Section A.5.1) may be used to access this variable. For example, given

a variable `P` of type `int*`, the expression `*P` denotes the `int` variable found at the address contained in `P`.

Like an object reference, a pointer may be `null`. Applying the indirection operator to a `null` pointer results in implementation-defined behavior. A pointer with value `null` is represented by all-bits-zero.

The `void*` type represents a pointer to an unknown type. Because the referent type is unknown, the indirection operator cannot be applied to a pointer of type `void*, nor can any arithmetic be performed on such a pointer`. However, a pointer of type `void*` can be cast to any other pointer type (and vice versa).

Pointer types are a separate category of types. Unlike reference types and value types, pointer types do not inherit from `object` and no conversions exist between pointer types and `object`. In particular, boxing and unboxing (Section 4.3) are not supported for pointers. However, conversions are permitted between different pointer types and between pointer types and the integral types. This is described in Section A.4.

A pointer type may be used as the type of a volatile field (Section 10.4.3).

Although pointers can be passed as `ref` or `out` parameters, doing so can cause undefined behavior, since the pointer may well be set to point to a local variable which no longer exists when the called method returns, or the fixed object to which it used to point, is no longer fixed. For example:

```
using System;
class Test
{
   static int value = 20;
   unsafe static void F(out int* pi1, ref int* pi2) {
      int i = 10;
      pi1 = &i;
      fixed (int* pj = &value) {
         // ...
         pi2 = pj;
      }
   }
   static void Main() {
      int i = 10;
      unsafe {
         int* px1;
         int* px2 = &i;
         F(out px1, ref px2);
         Console.WriteLine("*px1 = {0}, *px2 = {1}",
            *px1, *px2);   // undefined behavior
      }
   }
}
```

A method can return a value of some type, and that type can be a pointer. For example, when given a pointer to a contiguous sequence of `int`s, that sequence's element count, and some other `int` value, the following method returns the address of that value in that sequence, if a match occurs; otherwise it returns `null`:

```
unsafe static int* Find(int* pi, int size, int value) {
   for (int i = 0; i < size; ++i) {
      if (*pi == value)
         return pi;
      ++pi;
   }
   return null;
}
```

In an unsafe context, several constructs are available for operating on pointers:

- The `*` operator may be used to perform pointer indirection (Section A.5.1).
- The `->` operator may be used to access a member of a struct through a pointer (Section A.5.2).
- The `[]` operator may be used to index a pointer (Section A.5.3).
- The `&` operator may be used to obtain the address of a variable (Section A.5.4).
- The `++` and `--` operators may be used to increment and decrement pointers (Section A.5.5).

- The `+` and `-` operators may be used to perform pointer arithmetic (Section A.5.6).
- The `==`, `!=`, `<`, `>`, `<=`, and `=>` operators may be used to compare pointers (Section A.5.7).
- The `stackalloc` operator may be used to allocate memory from the call stack (Section A.7).
- The `fixed` statement may be used to temporarily fix a variable so its address can be obtained (Section A.6).

A.3 Fixed and moveable variables

The address-of operator (Section A.5.4) and the `fixed` statement (Section A.6) divide variables into two categories: Fixed variables and moveable variables.

Fixed variables reside in storage locations that are unaffected by operation of the garbage collector. (Examples of fixed variables include local variables, value parameters, and variables created by dereferencing pointers.) On the other hand, moveable variables reside in storage locations that are subject to relocation or disposal by the garbage collector. (Examples of moveable variables include fields in objects and elements of arrays.)

The `&` operator (Section A.5.4) permits the address of a fixed variable to be obtained without restrictions. However, because a moveable variable is subject to relocation or disposal by the garbage collector, the address of a moveable variable can only be obtained using a `fixed` statement (Section A.6), and that address remains valid only for the duration of that `fixed` statement.

In precise terms, a fixed variable is one of the following:

- A variable resulting from a *simple-name* (Section 7.5.2) that refers to a local variable or a value parameter.
- A variable resulting from a *member-access* (Section 7.5.4) of the form `V.I`, where `V` is a fixed variable of a *struct-type*.
- A variable resulting from a *pointer-indirection-expression* (Section A.5.1) of the form `*P`, a *pointer-member-access* (Section A.5.2) of the form `P->I`, or a *pointer-element-access* (Section A.5.3) of the form `P[E]`.

All other variables are classified as moveable variables.

Note that a static field is classified as a moveable variable. Also note that a `ref` or `out` parameter is classified as a moveable variable, even if the argument given for the parameter is a fixed variable. Finally, note that a variable produced by dereferencing a pointer is always classified as a fixed variable.

A.4 Pointer conversions

In an unsafe context, the set of available implicit conversions (Section 6.1) is extended to include the following implicit pointer conversions:

- From any *pointer-type* to the type `void*`.
- From the null type to any *pointer-type*.

Additionally, in an unsafe context, the set of available explicit conversions (Section 6.2) is extended to include the following explicit pointer conversions:

- From any *pointer-type* to any other *pointer-type*.
- From `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, or `ulong` to any *pointer-type*.
- From any *pointer-type* to `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, or `ulong`.

Finally, in an unsafe context, the set of standard implicit conversions (Section 6.3.1) includes the following pointer conversion:

- From any *pointer-type* to the type `void*`.

Conversions between two pointer types never change the actual pointer value. In other words, a conversion from one pointer type to another has no effect on the underlying address given by the pointer.

When one pointer type is converted to another, if the resulting pointer is not correctly aligned for the pointed-to type, the behavior is undefined if the result is dereferenced. In general, the concept "correctly aligned" is transitive: if a pointer to type `A` is correctly aligned for a pointer to type `B`, which, in turn, is correctly aligned for a pointer to type `C`, then a pointer to type `A` is correctly aligned for a pointer to type `C`.

Consider the following case in which a variable having one type is accessed via a pointer to a different type:

```
char c = 'A';
char* pc = &c;
void* pv = pc;
int* pi = (int*)pv;
int i = *pi;      // undefined
*pi = 123456;      // undefined
```

When a pointer type is converted to a pointer to byte, the result points to the lowest addressed byte of the variable. Successive increments of the result, up to the size of the variable, yield pointers to the remaining bytes of that variable. For example, the following method displays each of the eight bytes in a double as a hexadecimal value:

```
using System;
class Test
{
   unsafe static void Main() {
      double d = 123.456e23;
      unsafe {
         byte* pb = (byte*)&d;
         for (int i = 0; i < sizeof(double); ++i)
            Console.Write("{0:X2} ", *pb++);
         Console.WriteLine();
      }
   }
}
```

Of course, the output produced depends on endianness.

Mappings between pointers and integers are implementation-defined. However, on 32- and 64-bit CPU architectures with a linear address space, conversions of pointers to or from integral types typically behave exactly like conversions of `uint` or `ulong` values, respectively, to or from those integral types.

A.5 Pointers in expressions

In an unsafe context an expression may yield a result of a pointer type, but outside an unsafe context it is a compile-time error for an expression to be of a pointer type. In precise terms, outside an unsafe context a compile-time error occurs if any *simple-name* (Section 7.5.2), *member-access* (Section 7.5.4), *invocation-expression* (Section 7.5.5), or *element-access* (Section 7.5.6) is of a pointer type.

In an unsafe context, the *primary-no-array-creation-expression* (Section 7.5) and *unary-expression* (Section 7.6) productions permit the following additional constructs:

primary-no-array-creation-expression:
  ...
  pointer-member-access
  pointer-element-access
  sizeof-expression
unary-expression:
  ...
  pointer-indirection-expression
  addressof-expression

These constructs are described in the following sections. The precedence and associativity of the unsafe operators is implied by the grammar.

A.5.1 Pointer indirection

A *pointer-indirection-expression* consists of an asterisk (`*`) followed by a *unary-expression*.

pointer-indirection-expression:
  `*` *unary-expression*

The unary `*` operator denotes *pointer indirection* and is used to obtain the variable to which a pointer points. The result of evaluating `*P`, where `P` is an expression of a pointer type `T*`, is a variable of type `T`. It is a compile-time error to apply the unary `*` operator to an expression of type `void*` or to an expression that isn't of a pointer type.

The effect of applying the unary `*` operator to a `null` pointer is implementation-defined. In particular, there is no guarantee that this operation throws a `System.NullReferenceException`.

If an invalid value has been assigned to the pointer, the behavior of the unary `*` operator is undefined. Among the invalid values for dereferencing a pointer by the unary `*` operator are an address inappropriately aligned for the type pointed to (see example in Section A.4), and the address of a variable after the end of its lifetime.

For purposes of definite assignment analysis, a variable produced by evaluating an expression of the form `*P` is considered initially assigned (Section 5.3.1).

A.5.2 Pointer member access

A *pointer-member-access* consists of a *primary-expression*, followed by a "`->`" token, followed by an *identifier*.

pointer-member-access:
  primary-expression `->` *identifier*

In a pointer member access of the form `P->I`, `P` must be an expression of a pointer type other than `void*`, and `I` must denote an accessible member of the type to which `P` points.

A pointer member access of the form `P->I` is evaluated exactly as `(*P).I`. For a description of the pointer indirection operator (`*`), see Section A.5.1. For a description of the member access operator (`.`), see Section 7.5.4.

In the example

```
using System;
struct Point
{
   public int x;
   public int y;
   public override string ToString() {
      return "(" + x + "," + y + ")";
   }
}
class Test
{
   static void Main() {
      Point point;
      unsafe {
         Point* p = &point;
         p->x = 10;
         p->y = 20;
         Console.WriteLine(p->ToString());
      }
   }
}
```

the `->` operator is used to access fields and invoke a method of a struct through a pointer. Because the operation `P->I` is precisely equivalent to `(*P).I`, the `Main` method could equally well have been written:

```
class Test
{
   static void Main() {
      Point point;
      unsafe {
         Point* p = &point;
         (*p).x = 10;
         (*p).y = 20;
         Console.WriteLine((*p).ToString());
      }
   }
}
```

A.5.3 Pointer element access

A *pointer-element-access* consists of a *primary-no-array-creation-expression* followed by an expression enclosed in "`[`" and "`]`".

pointer-element-access:
  primary-no-array-creation-expression [*expression*]

In a pointer element access of the form `P[E]`, `P` must be an expression of a pointer type other than `void*`, and `E` must be an expression of a type that can be implicitly converted to `int`, `uint`, `long`, or `ulong`.

A pointer element access of the form `P[E]` is evaluated exactly as `*(P + E)`. For a description of the pointer indirection operator (`*`), see Section A.5.1. For a description of the pointer addition operator (`+`), see Section A.5.6.

In the example

```
class Test
{
   static void Main() {
      unsafe {
         char* p = stackalloc char[256];
         for (int i = 0; i < 256; i++) p[i] = (char)i;
      }
   }
}
```

a pointer element access is used to initialize the character buffer in a `for` loop. Because the operation `P[E]` is precisely equivalent to `*(P + E)`, the example could equally well have been written:

```
class Test
{
   static void Main() {
      unsafe {
         char* p = stackalloc char[256];
         for (int i = 0; i < 256; i++) *(p + i) = (char)i;
      }
   }
}
```

The pointer element access operator does not check for out-of-bounds errors and the behavior when accessing an out-of-bounds element is undefined. This is the same as C and C++.

A.5.4 The address-of operator

An *addressof-expression* consists of an ampersand (`&`) followed by a *unary-expression*.

addressof-expression:
  & *unary-expression*

Given an expression `E` that is of a type `T` and is classified as a fixed variable (Section A.3), the construct `&E` computes the address of the variable given by `E`. The type of the result is `T*` and is classified as a value. A compile-time error occurs if `E` is not classified as a variable, if `E` is classified as a volatile field, or if `E` denotes a moveable variable. In the last case, a fixed statement (Section A.6) can be used to temporarily "fix" the variable before obtaining its address.

The `&` operator does not require its argument to be definitely assigned, but following an `&` operation, the variable to which the operator is applied is considered definitely assigned in the execution path in which the operation occurs. It is the responsibility of the programmer to ensure that correct initialization of the variable actually does take place in this situation.

In the example

```
using System;
class Test
{
   static void Main() {
      int i;
      unsafe {
         int* p = &i;
         *p = 123;
      }
      Console.WriteLine(i);
   }
}
```

`i` is considered definitely assigned following the `&i` operation used to initialize `p`. The assignment to `*p` in effect initializes `i`, but the inclusion of this initialization is the responsibility of the programmer, and no compile-time error would occur if the assignment was removed.

The rules of definite assignment for the `&` operator exist such that redundant initialization of local variables can be avoided. For example, many external APIs take a pointer to a structure that is filled in by the API. Calls to such APIs typically pass the address of a local struct variable, and without the rule, redundant initialization of the struct variable would be required.

As stated in Section 7.5.4, outside an instance constructor or static constructor for a struct or class that defines a `readonly` field, that field is considered a value, not a variable. As such, its address cannot be taken. Similarly, the address of a constant cannot be taken.

A.5.5 Pointer increment and decrement

In an unsafe context, the `++` (Section 7.5.9) and -- operators (Section 7.6.5) can be applied to pointer variables of all types except `void*`. Thus, for every pointer type `T*`, the following operators are implicitly defined:

```
T* operator ++(T* x);
T* operator --(T* x);
```

The operators produce the same results as x + 1 and x - 1, respectively (Section A.5.6). In other words, for a pointer variable of type `T*`, the `++` operator adds `sizeof(T)` to the address contained in the variable, and the –– operator subtracts `sizeof(T)` from the address contained in the variable.

If a pointer increment or decrement operation overflows the domain of the pointer type, the result is implementation-defined, but no exceptions are produced.

A.5.6 Pointer arithmetic

In an unsafe context, the `+` (Section 7.7.4) and `-` operators (Section 7.7.5) can be applied to values of all pointer types except `void*`. Thus, for every pointer type `T*`, the following operators are implicitly defined:

```
T* operator +(T* x, int y);
T* operator +(T* x, uint y);
T* operator +(T* x, long y);
T* operator +(T* x, ulong y);
T* operator +(int x, T* y);
T* operator +(uint x, T* y);
T* operator +(long x, T* y);
T* operator +(ulong x, T* y);
T* operator -(T* x, int y);
T* operator -(T* x, uint y);
T* operator -(T* x, long y);
T* operator -(T* x, ulong y);
long operator -(T* x, T* y);
```

Given an expression `P` of a pointer type `T*` and an expression `N` of type `int`, `uint`, `long`, or `ulong`, the expressions `P + N` and `N + P` compute the pointer value of type `T*` that results from adding `N * sizeof(T)` to the address given by `P`. Likewise, the expression `P - N` computes the pointer value of type `T*` that results from subtracting `N * sizeof(T)` from the address given by `P`.

Given two expressions, `P` and `Q`, of a pointer type `T*`, the expression `P - Q` computes the difference between the addresses given by `P` and `Q` and then divides that difference by `sizeof(T)`. The type of the result is always `long`. In effect, `P - Q` is computed as `((long)(P) - (long)(Q)) / sizeof(T)`.

For example:

```
using System;
class Test
{
   static void Main() {
      unsafe {
         int* values = stackalloc int[20];
         int* p = &values[1];
         int* q = &values[15];
         Console.WriteLine("p - q = {0}", p - q);
         Console.WriteLine("q - p = {0}", q - p);
      }
   }
}
```

which produces the output:

```
p - q = -14
q - p = 14
```

If a pointer arithmetic operation overflows the domain of the pointer type, the result is truncated in an implementation-defined fashion, but no exceptions are produced.

A.5.7 Pointer comparison

In an unsafe context, the `==`, `!=`, `<`, `>`, `<=`, and `=>` operators (Section 7.9) can be applied to values of all pointer types. The pointer comparison operators are:

```
bool operator ==(void* x, void* y);
bool operator !=(void* x, void* y);
bool operator <(void* x, void* y);
bool operator >(void* x, void* y);
bool operator <=(void* x, void* y);
bool operator >=(void* x, void* y);
```

Because an implicit conversion exists from any pointer type to the `void*` type, operands of any pointer type can be compared using these operators. The comparison operators compare the addresses given by the two operands as if they were unsigned integers.

A.5.8 The sizeof operator

The `sizeof` operator returns the number of bytes occupied by a variable of a given type. The type specified as an operand to `sizeof` must be an *unmanaged-type* (Section A.2).

sizeof-expression:
  sizeof (*unmanaged-type*)

The result of the `sizeof` operator is a value of type `int`. For certain predefined types, the `sizeof` operator yields a constant value as shown in the table below.

Expression	Result
`sizeof(sbyte)`	1
`sizeof(byte)`	1
`sizeof(short)`	2
`sizeof(ushort)`	2
`sizeof(int)`	4
`sizeof(uint)`	4
`sizeof(long)`	8
`sizeof(ulong)`	8
`sizeof(char)`	2
`sizeof(float)`	4
`sizeof(double)`	8
`sizeof(bool)`	1

For all other types, the result of the `sizeof` operator is implementation-defined and is classified as a value, not a constant.

The order in which members are packed into a struct is unspecified.

For alignment purposes, there may be unnamed padding at the beginning of a struct, within a struct, and at the end of the struct. The contents of the bits used as padding are indeterminate.

When applied to an operand that has struct type, the result is the total number of bytes in a variable of that type, including any padding.

A.6 The fixed statement

In an unsafe context, the *embedded-statement* (Section 8) production permits an additional construct, the `fixed` statement, which is used to "fix" a moveable variable such that its address remains constant for the duration of the statement.

embedded-statement:
  ...
  fixed-statement
fixed-statement:
  fixed (*pointer-type* *fixed-pointer-declarators*) *embedded-statement*
fixed-pointer-declarators:
  fixed-pointer-declarator
  fixed-pointer-declarators , *fixed-pointer-declarator*
fixed-pointer-declarator:
  identifier = *fixed-pointer-initializer*
fixed-pointer-initializer:
  & *variable-reference*
  expression

Each *fixed-pointer-declarator* declares a local variable of the given *pointer-type* and initializes that local variable with the address computed by the corresponding *fixed-pointer-initializer*. A local variable declared in a `fixed` statement is accessible in any *fixed-pointer-initializer*s occurring to the right of that variable's declaration, and in the *embedded-statement* of the `fixed` statement. A local variable declared by a `fixed` statement is considered read-only. A compile-time error occurs if the embedded statement attempts to modify this local variable (via assignment or the ++ and -- operators) or pass it as a `ref` or `out` parameter.

A *fixed-pointer-initializer* can be one of the following:

- The token "`&`" followed by a *variable-reference* (Section 5.4) to a moveable variable (Section A.3) of an unmanaged type `T`, provided the type `T*` is implicitly convertible to the pointer type given in the `fixed` statement. In this case, the initializer computes the address of the given variable, and the variable is guaranteed to remain at a fixed address for the duration of the `fixed` statement.
- An expression of an *array-type* with elements of an unmanaged type `T`, provided the type `T*` is implicitly convertible to the pointer type given in the `fixed` statement. In this case, the initializer computes the address of the first element in the array, and the entire array is guaranteed to remain at a fixed address for the duration of the `fixed` statement. The behavior of the `fixed` statement is implementation-defined if the array expression is null or if the array has zero elements.
- An expression of type `string`, provided the type `char*` is implicitly convertible to the pointer type given in the `fixed` statement. In this case, the initializer computes the address of the first character in the string, and the entire string is guaranteed to remain at a fixed address for the duration of the `fixed` statement. The behavior of the `fixed` statement is implementation-defined if the string expression is null.

For each address computed by a *fixed-pointer-initializer* the `fixed` statement ensures that the variable referenced by the address is not subject to relocation or disposal by the garbage collector for the duration of the `fixed` statement. For example, if the address computed by a *fixed-pointer-initializer* references a field of an object or an element of an array instance, the `fixed` statement guarantees that the containing object instance is not relocated or disposed of during the lifetime of the statement.

It is the programmer's responsibility to ensure that pointers created by `fixed` statements do not survive beyond execution of those statements. For example, when pointers created by `fixed` statements are passed to external APIs, it is the programmer's responsibility to ensure that the APIs retain no memory of these pointers.

Fixed objects may cause fragmentation of the heap (because they can't be moved). For that reason, objects should be fixed only when absolutely necessary and then only for the shortest amount of time possible.

The example

```
class Test
{
   static int x;
   int y;
   unsafe static void F(int* p) {
      *p = 1;
   }
   static void Main() {
      Test t = new Test();
```

```
      int[] a = new int[10];
      unsafe {
         fixed (int* p = &x) F(p);
         fixed (int* p = &t.y) F(p);
         fixed (int* p = &a[0]) F(p);
         fixed (int* p = a) F(p);
      }
   }
}
```

demonstrates several uses of the `fixed` statement. The first statement fixes and obtains the address of a static field, the second statement fixes and obtains the address of an instance field, and the third statement fixes and obtains the address of an array element. In each case it would have been an error to use the regular `&` operator since the variables are all classified as moveable variables.

The third and fourth `fixed` statements in the example above produce identical results. In general, for an array instance `a`, specifying `&a[0]` in a `fixed` statement is the same as simply specifying `a`.

Here's another example of the `fixed` statement, this time using `string`:

```
class Test
{
   static string name = "xx";
   unsafe static void F(char* p) {
      for (int i = 0; p[i] != '\0'; ++i)
         Console.WriteLine(p[i]);
   }
   static void Main() {
      unsafe {
         fixed (char* p = name) F(p);
         fixed (char* p = "xx") F(p);
      }
   }
}
```

In an unsafe context array elements of single-dimensional arrays are stored in increasing index order, starting with index `0` and ending with index `Length - 1`. For multi-dimensional arrays, array elements are stored such that the indices of the rightmost dimension are increased first, then the next left dimension, and so on to the left. Within a `fixed` statement that obtains a pointer `p` to an array instance `a`, the pointer values ranging from `p` to `p + a.Length - 1` represent addresses of the elements in the array. Likewise, the variables ranging from `p[0]` to `p[a.Length - 1]` represent the actual array elements. Given the way in which arrays are stored, we can treat an array of any dimension as though it were linear.

For example:

```
using System;
class Test
{
   static void Main() {
      int[,,] a = new int[2,3,4];
      unsafe {
         fixed (int* p = a) {
            for (int i = 0; i < a.Length; ++i)   // treat as linear
               p[i] = i;
         }
      }
      for (int i = 0; i < 2; ++i)
         for (int j = 0; j < 3; ++j) {
            for (int k = 0; k < 4; ++k)
               Console.Write("[{0},{1},{2}] = {3,2} ", i, j, k, a[i,j,k]);
            Console.WriteLine();
         }
   }
}
```

which produces the output:

```
[0,0,0] =  0 [0,0,1] =  1 [0,0,2] =  2 [0,0,3] =  3
[0,1,0] =  4 [0,1,1] =  5 [0,1,2] =  6 [0,1,3] =  7
[0,2,0] =  8 [0,2,1] =  9 [0,2,2] = 10 [0,2,3] = 11
[1,0,0] = 12 [1,0,1] = 13 [1,0,2] = 14 [1,0,3] = 15
[1,1,0] = 16 [1,1,1] = 17 [1,1,2] = 18 [1,1,3] = 19
[1,2,0] = 20 [1,2,1] = 21 [1,2,2] = 22 [1,2,3] = 23
```

In the example

```
class Test
{
   unsafe static void Fill(int* p, int count, int value) {
      for (; count != 0; count--) *p++ = value;
   }
   static void Main() {
      int[] a = new int[100];
      unsafe {
         fixed (int* p = a) Fill(p, 100, -1);
      }
   }
}
```

a `fixed` statement is used to fix an array so its address can be passed to a method that takes a pointer.

A `char*` value produced by fixing a string instance always points to a null-terminated string. Within a fixed statement that obtains a pointer `p` to a string instance `s`, the pointer values ranging from `p` to `p + s.Length - 1` represent addresses of the characters in the string, and the pointer value `p + s.Length` always points to a null character (the character with value `'\0'`).

Modifying objects of managed type through fixed pointers can result in undefined behavior. For example, because strings are immutable, it is the programmer's responsibility to ensure that the characters referenced by a pointer to a fixed string are not modified.

The automatic null-termination of strings is particularly convenient when calling external APIs that expect "C-style" strings. Note, however, that a string instance is permitted to contain null characters. If such null characters are present, the string will appear truncated when treated as a null-terminated `char*`.

A.7 Stack allocation

In an unsafe context, a local variable declaration (Section 8.5.1) may include a stack allocation initializer that allocates memory from the call stack.

local-variable-initializer:
  expression
  array-initializer
  stackalloc-initializer
stackalloc-initializer:
  stackalloc *unmanaged-type* [*expression*]

The *unmanaged-type* indicates the type of the items that will be stored in the newly allocated location, and the *expression* indicates the number of these items. Taken together, these specify the required allocation size. Since the size of a stack allocation cannot be negative, it is a compile-time error to specify the number of items as a *constant-expression* that evaluates to a negative value.

A stack allocation initializer of the form `stackalloc T[E]` requires `T` to be an unmanaged type (Section A.2) and `E` to be an expression of type `int`. The construct allocates `E * sizeof(T)` bytes from the call stack and returns a pointer, of type `T*`, to the newly allocated block. If `E` is a negative value, then the behavior is undefined. If `E` is zero, then no allocation is made, and the pointer returned is implementation-defined. If there is not enough memory available to allocate a block of the given size, a `System.StackOverflowException` is thrown.

The content of the newly allocated memory is undefined.

Stack allocation initializers are not permitted in `catch` or `finally` blocks (Section 8.10).

There is no way to explicitly free memory allocated using `stackalloc`. All stack allocated memory blocks created during the execution of a function member are automatically discarded when that function member returns. This corresponds to the `alloca` function, an extension commonly found in C and C++ implementations.

In the example

```
using System;
class Test
{
   static string IntToString(int value) {
      int n = value >= 0? value: -value;
      unsafe {
         char* buffer = stackalloc char[16];
         char* p = buffer + 16;
         do {
            *--p = (char)(n % 10 + '0');
            n /= 10;
         } while (n != 0);
         if (value < 0) *--p = '-';
         return new string(p, 0, (int)(buffer + 16 - p));
      }
   }
   static void Main() {
      Console.WriteLine(IntToString(12345));
      Console.WriteLine(IntToString(-999));
   }
}
```

a `stackalloc` initializer is used in the `IntToString` method to allocate a buffer of 16 characters on the stack. The buffer is automatically discarded when the method returns.

A.8 Dynamic memory allocation

Except for the `stackalloc` operator, C# provides no predefined constructs for managing non-garbage collected memory. Such services are typically provided by supporting class libraries or imported directly from the underlying operating system. For example, the `Memory` class below illustrates how the heap functions of an underlying operating system might be accessed from C#:

```
using System;
using System.Runtime.InteropServices;
public unsafe class Memory
{
   // Handle for the process heap. This handle is used in all calls to the
   // HeapXXX APIs in the methods below.
   static int ph = GetProcessHeap();
   // Private instance constructor to prevent instantiation.
   private Memory() {}
   // Allocates a memory block of the given size. The allocated memory is
   // automatically initialized to zero.
   public static void* Alloc(int size) {
      void* result = HeapAlloc(ph, HEAP_ZERO_MEMORY, size);
      if (result == null) throw new OutOfMemoryException();
      return result;
   }
   // Copies count bytes from src to dst. The source and destination
   // blocks are permitted to overlap.
   public static void Copy(void* src, void* dst, int count) {
      byte* ps = (byte*)src;
      byte* pd = (byte*)dst;
      if (ps > pd) {
         for (; count != 0; count--) *pd++ = *ps++;
      }
      else if (ps < pd) {
         for (ps += count, pd += count; count != 0; count--) *--pd = *--ps;
      }
   }
   // Frees a memory block.
   public static void Free(void* block) {
      if (!HeapFree(ph, 0, block)) throw new InvalidOperationException();
   }
   // Re-allocates a memory block. If the reallocation request is for a
   // larger size, the additional region of memory is automatically
   // initialized to zero.
   public static void* ReAlloc(void* block, int size) {
      void* result = HeapReAlloc(ph, HEAP_ZERO_MEMORY, block, size);
      if (result == null) throw new OutOfMemoryException();
      return result;
   }
   // Returns the size of a memory block.
   public static int SizeOf(void* block) {
      int result = HeapSize(ph, 0, block);
      if (result == -1) throw new InvalidOperationException();
      return result;
   }
   // Heap API flags
   const int HEAP_ZERO_MEMORY = 0x00000008;
   // Heap API functions
   [DllImport("kernel32")]
   static extern int GetProcessHeap();
   [DllImport("kernel32")]
   static extern void* HeapAlloc(int hHeap, int flags, int size);
   [DllImport("kernel32")]
   static extern bool HeapFree(int hHeap, int flags, void* block);
   [DllImport("kernel32")]
   static extern void* HeapReAlloc(int hHeap, int flags,
      void* block, int size);
   [DllImport("kernel32")]
   static extern int HeapSize(int hHeap, int flags, void* block);
```

```
}
```

An example that uses the `Memory` class is given below:

```
class Test
{
   static void Main() {
      unsafe {
         byte* buffer = (byte*)Memory.Alloc(256);
         try {
            for (int i = 0; i < 256; i++) buffer[i] = (byte)i;
            byte[] array = new byte[256];
            fixed (byte* p = array) Memory.Copy(buffer, p, 256);
         }
         finally {
            Memory.Free(buffer);
         }
         for (int i = 0; i < 256; i++) Console.WriteLine(array[i]);
      }
   }
}
```

The example allocates 256 bytes of memory through `Memory.Alloc` and initializes the memory block with values increasing from 0 to 255. It then allocates a 256 element byte array and uses `Memory.Copy` to copy the contents of the memory block into the byte array. Finally, the memory block is freed using `Memory.Free` and the contents of the byte array are output on the console.

B. Documentation comments

C# provides a mechanism for programmers to document their code using a special comment syntax that contains XML text. In source code files, comments having a certain form can be used to direct a tool to produce XML from those comments and the source code elements, which they precede. Comments using such syntax are called documentation comments. They must immediately precede a user-defined type (such as a class, delegate, or interface) or a member (such as a field, event, property, or method). The XML generation tool is called the documentation generator. (This generator could be, but need not be, the C# compiler itself.) The output produced by the documentation generator is called the documentation file. A documentation file is used as input to a documentation viewer; a tool intended to produce some sort of visual display of type information and its associated documentation.

This specification suggests a set of tags to be used in documentation comments, but use of these tags is not required, and other tags may be used if desired, as long the rules of well-formed XML are followed.

B.1 Introduction

Comments having a special form can be used to direct a tool to produce XML from those comments and the source code elements, which they precede. Such comments are single-line comments that start with three slashes (`///`), or delimited comments that start with a slash and two stars (`/**`). They must immediately precede a user-defined type (such as a class, delegate, or interface) or a member (such as a field, event, property, or method) that they annotate. Attribute sections (Section 17.2) are considered part of declarations, so documentation comments must precede attributes applied to a type or member.

Syntax:

single-line-doc-comment:
  /// input-characters$_{opt}$

  delimited-doc-comment:

  */** delimited-comment-charactersopt */*

In a *single-line-doc-comment*, if there is a *whitespace* character following the `///` characters on each of the *single-line-doc-comment*s adjacent to the current *single-line-doc-comment*, then that *whitespace* character is not included in the XML output.

In a *delimited-doc-comment*, if the first non-whitespace character on the second line is an asterisk and the same pattern of optional whitespace characters and an asterisk character is repeated at the beginning of each of the line within the *delimited-doc-comment*, then the characters of the repeated pattern are not included in the XML output. The pattern may include whitespace characters after, as well as before, the asterisk character.

Example:

```
/// <remarks>Class <c>Point</c> models a point in a two-dimensional
/// plane.</remarks>
///
public class Point
{
   /// <remarks>method <c>draw</c> renders the point.</remarks>
   void draw() {...}
}
```

The text within documentation comments must be well formed according to the rules of XML (http://www.w3.org/TR/REC-xml). If the XML is ill formed, a warning is generated and the documentation file will contain a comment saying that an error was encountered.

Although developers are free to create their own set of tags, a recommended set is defined in Appendix B.2. Some of the recommended tags have special meanings:

- The `<param>` tag is used to describe parameters. If such a tag is used, the documentation generator must verify that the specified parameter exists and that all parameters are described in documentation comments. If such verification fails, the documentation generator issues a warning.
- The `cref` attribute can be attached to any tag to provide a reference to a code element. The documentation generator must verify that this code element exists. If the verification fails, the documentation generator issues a warning. When looking for a name described in a `cref` attribute, the documentation generator must respect namespace visibility according to `using` statements appearing within the source code.
- The `<summary>` tag is intended to be used by a documentation viewer to display additional information about a type or member.
- The `<include>` tag includes information from an external XML file.

Note carefully that the documentation file does not provide full information about the type and members (for example, it does not contain any type information). To get such information about a type or member, the documentation file must be used in conjunction with reflection on the actual type or member.

B.2 Recommended tags

The documentation generator must accept and process any tag that is valid according to the rules of XML. The following tags provide commonly used functionality in user documentation. (Of course, other tags are possible.)

Tag	Section	Purpose
`<c>`	B.2.1	Set text in a code-like font
`<code>`	B.2.2	Set one or more lines of source code or program output
`<example>`	B.2.3	Indicate an example
`<exception>`	B.2.4	Identifies the exceptions a method can throw
`<include>`	B.2.5	Includes XML from an external file
`<list>`	B.2.6	Create a list or table
`<para>`	B.2.7	Permit structure to be added to text
`<param>`	B.2.8	Describe a parameter for a method or constructor
`<paramref>`	B.2.9	Identify that a word is a parameter name
`<permission>`	B.2.10	Document the security accessibility of a member
`<remarks>`	B.2.11	Describe a type
`<returns>`	B.2.12	Describe the return value of a method
`<see>`	B.2.13	Specify a link
`<seealso>`	B.2.14	Generate a *See Also* entry
`<summary>`	B.2.15	Describe a member of a type
`<value>`	B.2.16	Describe a property

B.2.1 <c>

This tag provides a mechanism to indicate that a fragment of text within a description should be set in a special font such as that used for a block of code. For lines of actual code, use `<code>` (B.2.2).

Syntax:

```
<c>text</c>
```

Example:

```
/// <remarks>Class <c>Point</c> models a point in a two-dimensional
/// plane.</remarks>
public class Point
{
   // ...
}
```

B.2.2 <code>

This tag is used to set one or more lines of source code or program output in some special font. For small code fragments in narrative, use <c> (B.2.1).

Syntax:

```
<code>source code or program output</code>
```

Example:

```
/// <summary>This method changes the point's location by
///   the given x- and y-offsets.
/// <example>For example:
/// <code>
///   Point p = new Point(3,5);
///   p.Translate(-1,3);
/// </code>
/// results in <c>p</c>'s having the value (2,8).
/// </example>
/// </summary>
public void Translate(int xor, int yor) {
   X += xor;
   Y += yor;
}
```

B.2.3 <example>

This tag allows example code within a comment, to specify how a method or other library member may be used. Ordinarily, this would also involve use of the tag `<code>` (B.2.2) as well.

Syntax:

```
<example>description</example>
```

Example:

See `<code>` (B.2.2) for an example.

B.2.4 <exception>

This tag provides a way to document the exceptions a method can throw.

Syntax:

```
<exception cref="member">description</exception>
```

where

`cref="member"`
: The name of a member. The documentation generator checks that the given member exists and translates *member* to the canonical element name in the documentation file.

`description`
: A description of the circumstances in which the exception is thrown.

Example:

```
public class DataBaseOperations
{
   /// <exception cref="MasterFileFormatCorruptException"></exception>
   /// <exception cref="MasterFileLockedOpenException"></exception>
   public static void ReadRecord(int flag) {
      if (flag == 1)
         throw new MasterFileFormatCorruptException();
      else if (flag == 2)
         throw new MasterFileLockedOpenException();
      // ...
   }
}
```

B.2.5 <include>

This tag allows including information from an XML document that is external to the source code file. The external file must be a well-formed XML document, and an `XPath` expression is applied to that document to specify what XML from that document to include. The `<include>` tag is then replaced with the selected XML from the external document.

Syntax:

```
<include file="filename" path="xpath" />
```

where

`file="filename"`
: The file name of an external XML file. The file name is interpreted relative to the file that contains the include tag.

`path="xpath"`
: An `XPath` expression that selects some of the XML in the external XML file.

Example:

If the source code contained a declaration like:

```
/// <include file="docs.xml" path='extradoc/class[@name="IntList"]/*' />
public class IntList { ... }
```

and the external file "docs.xml" had the following contents:

```
<?xml version="1.0"?>
<extradoc>
   <class name="IntList">
      <summary>
         Contains a list of integers.
      </summary>
   </class>
   <class name="StringList">
      <summary>
         Contains a list of integers.
      </summary>
   </class>
</extradoc>
```

then the same documentation is output as if the source code contained:

```
/// <summary>
///    Contains a list of integers.
/// </summary>
public class IntList { ... }
```

B.2.6 <list>

This tag is used to create a list or table of items. It may contain a `<listheader>` block to define the heading row of either a table or definition list. (When defining a table, only an entry for `term` in the heading need be supplied.)

Each item in the list is specified with an `<item>` block. When creating a definition list, both `term` and `description` must be specified. However, for a table, bulleted list, or numbered list, only `description` need be specified.

Syntax:

```
<list type="bullet" | "number" | "table">
   <listheader>
      <term>term</term>
      <description>description</description>
   </listheader>
   <item>
      <term>term</term>
      <description>description</description>
   </item>
   ...
   <item>
      <term>term</term>
      <description>description</description>
   </item>
</list>
```

where

`term`
: The term to define, whose definition is in `description`.

`description`
: Either an item in a bullet or numbered list, or the definition of a `term`.

Example:

```
public class MyClass
{
   /// <remarks>Here is an example of a bulleted list:
   /// <list type="bullet">
   /// <item>
   /// <description>Item 1.</description>
   /// </item>
   /// <item>
   /// <description>Item 2.</description>
   /// </item>
   /// </list>
   /// </remarks>
   public static void Main () {
      // ...
   }
}
```

B.2.7 <para>

This tag is for use inside other tags, such as `<remarks>` (B.2.11) or `<returns>` (B.2.12), and permits structure to be added to text.

Syntax:

```
<para>content</para>
```

where

`content`
The text of the paragraph.

Example:

```
/// <summary>This is the entry point of the Point class testing program.
/// <para>This program tests each method and operator, and
/// is intended to be run after any non-trvial maintenance has
/// been performed on the Point class.</para></summary>
public static void Main() {
   // ...
}
```

B.2.8 <param>

This tag is used to describe a parameter for a method, constructor, or indexer.

Syntax:

```
<param name="name">description</param>
```

where

`name`
: The name of the parameter.

`description`
: A description of the parameter.

Example:

```
/// <summary>This method changes the point's location to
///   the given coordinates.</summary>
/// <param><c>xor</c> is the new x-coordinate.</param>
/// <param><c>yor</c> is the new y-coordinate.</param>
public void Move(int xor, int yor) {
   X = xor;
   Y = yor;
}
```

B.2.9 <paramref>

This tag is used to indicate that a word is a parameter. The documentation file can be processed to format this parameter in some distinct way.

Syntax:

```
<paramref name="name"/>
```

where

`name`
The name of the parameter.

Example:

```
/// <summary>This constructor initializes the new Point to
///   (<paramref name="xor"/>,<paramref name="yor"/>).</summary>
/// <param><c>xor</c> is the new Point's x-coordinate.</param>
/// <param><c>yor</c> is the new Point's y-coordinate.</param>
public Point(int xor, int yor) {
   X = xor;
   Y = yor;
}
```

B.2.10 <permission>

This tag allows the security accessibility of a member to be documented.

Syntax:

```
<permission cref="member">description</permission>
```

where

`cref="member"`
: The name of a member. The documentation generator checks that the given code element exists and translates *member* to the canonical element name in the documentation file.

`description`
: A description of the access to the member.

Example:

```
/// <permission cref="System.Security.PermissionSet">Everyone can
/// access this method.</permission>
public static void Test() {
   // ...
}
```

B.2.11 <remarks>

This tag is used to specify overview information about a type. (Use `<summary>` (B.2.15) to describe the members of a type.)

Syntax:

```
<remarks>description</remarks>
```

where

`description`
The text of the remarks.

Example:

```
/// <remarks>Class <c>Point</c> models a point in a
/// two-dimensional plane.</remarks>
public class Point
{
   // ...
}
```

B.2.12 <returns>

This tag is used to describe the return value of a method.

Syntax:

```
<returns>description</returns>
```

where

`description`
A description of the return value.

Example:

```
/// <summary>Report a point's location as a string.</summary>
/// <returns>A string representing a point's location, in the form (x,y),
///   without any leading, trailing, or embedded whitespace.</returns>
public override string ToString() {
   return "(" + X + "," + Y + ")";
}
```

B.2.13 <see>

This tag allows a link to be specified within text. Use `<seealso>` (B.2.14) to indicate text that is to appear in a *See Also* section.

Syntax:

```
<see cref="member"/>
```

where

`cref="member"`
The name of a member. The documentation generator checks that the given code element exists and changes *member* to the element name in the generated documentation file.

Example:

```
/// <summary>This method changes the point's location to
///   the given coordinates.</summary>
/// <see cref="Translate"/>
public void Move(int xor, int yor) {
   X = xor;
   Y = yor;
}
/// <summary>This method changes the point's location by
///   the given x- and y-offsets.
/// </summary>
/// <see cref="Move"/>
public void Translate(int xor, int yor) {
   X += xor;
   Y += yor;
}
```

B.2.14 <seealso>

This tag allows an entry to be generated for the *See Also* section. Use `<see>` (B.2.13) to specify a link from within text.

Syntax:

```
<seealso cref="member"/>
```

where

`cref="member"`
The name of a member. The documentation generator checks that the given code element exists and changes *member* to the element name in the generated documentation file.

Example:

```
/// <summary>This method determines whether two Points have the same
///   location.</summary>
/// <seealso cref="operator=="/>
/// <seealso cref="operator!="/>
public override bool Equals(object o) {
   // ...
}
```

B.2.15 <summary>

This tag can be used to describe a member for a type. Use `<remarks>` (B.2.11) to describe the type itself.

Syntax:

```
<summary>description</summary>
```

where

`description`
A summary of the member.

Example:

```
/// <summary>This constructor initializes the new Point to (0,0).</summary>
public Point() : this(0,0) {
}
```

B.2.16 <value>

This tag allows a property to be described.

Syntax:

```
<value>property description</value>
```

where

`property description`
A description for the property.

Example:

```
/// <value>Property <c>X</c> represents the point's x-coordinate.</value>
public int X
{
   get { return x; }
   set { x = value; }
}
```

B.3 Processing the documentation file

The documentation generator generates an ID string for each element in the source code that is tagged with a documentation comment. This ID string uniquely identifies a source element. A documentation viewer can use an ID string to identify the corresponding metadata/reflection item to which the documentation applies.

The documentation file is not a hierarchical representation of the source code; rather, it is a flat list with a generated ID string for each element.

B.3.1 ID string format

The documentation generator observes the following rules when it generates the ID strings:

- No white space is placed in the string.
- The first part of the string identifies the kind of member being documented, via a single character followed by a colon. The following kinds of members are defined:

Character	Description
E	Event
F	Field
M	Method (including constructors, destructors, and operators)
N	Namespace
P	Property (including indexers)
T	Type (such as class, delegate, enum, interface, and struct)
!	Error string; the rest of the string provides information about the error. For example, the documentation generator generates error information for links that cannot be resolved.

- The second part of the string is the fully qualified name of the element, starting at the root of the namespace. The name of the element, its enclosing type(s), and namespace are separated by periods. If the name of the item itself has periods, they are replaced by `#` (`U+0023`) characters. (It is assumed that no element has this character in its name.)
- For methods and properties with arguments, the argument list follows, enclosed in parentheses. For those without arguments, the parentheses are omitted. The arguments are separated by commas. The encoding of each argument is the same as a CLI signature, as follows: Arguments are represented by their fully qualified name. For example, `int` becomes `System.Int32`, `string` becomes `System.String`, `object` becomes `System.Object`, and so on. Arguments having the `out` or `ref` modifier have an `@` following their type name. Arguments passed by value or via `params` have no special notation. Arguments that are arrays are represented as `[` *lowerbound* `:` *size* `,` … `,` *lowerbound* `:` *size* `]` where the number of commas is the rank less one, and the lower bounds and size of each dimension, if known, are represented in decimal. If a lower bound or size is not specified, it is omitted. If the lower bound and size for a particular dimension are omitted, the "`:`" is omitted as well. Jagged arrays are represented by one "`[]`" per level. Arguments that have pointer types other than void are represented using a `*` following the type name. A void pointer is represented using a type name of `System.Void`.

B.3.2 ID string examples

The following examples each show a fragment of C# code, along with the ID string produced from each source element capable of having a documentation comment:

- Types are represented using their fully qualified name.

```
enum Color { Red, Blue, Green }
namespace Acme
{
   interface IProcess {...}
   struct ValueType {...}
   class Widget: IProcess
   {
      public class NestedClass {...}
      public interface IMenuItem {...}
      public delegate void Del(int i);
      public enum Direction { North, South, East, West }
   }
}
"T:Color"
"T:Acme.IProcess"
"T:Acme.ValueType"
"T:Acme.Widget"
"T:Acme.Widget.NestedClass"
"T:Acme.Widget.IMenuItem"
"T:Acme.Widget.Del"
"T:Acme.Widget.Direction"
```

- Fields are represented by their fully qualified name.

```
namespace Acme
{
   struct ValueType
   {
      private int total;
   }
   class Widget: IProcess
   {
      public class NestedClass
      {
         private int value;
      }
      private string message;
      private static Color defaultColor;
      private const double PI = 3.14159;
      protected readonly double monthlyAverage;
      private long[] array1;
      private Widget[,] array2;
      private unsafe int *pCount;
      private unsafe float **ppValues;
   }
}
"F:Acme.ValueType.total"
"F:Acme.Widget.NestedClass.value"
"F:Acme.Widget.message"
"F:Acme.Widget.defaultColor"
"F:Acme.Widget.PI"
"F:Acme.Widget.monthlyAverage"
```

```
"F:Acme.Widget.array1"
"F:Acme.Widget.array2"
"F:Acme.Widget.pCount"
"F:Acme.Widget.ppValues"
```

- Constructors.

```
namespace Acme
{
   class Widget: IProcess
   {
      static Widget() {...}
      public Widget() {...}
      public Widget(string s) {...}
   }
}
"M:Acme.Widget.#cctor"
"M:Acme.Widget.#ctor"
"M:Acme.Widget.#ctor(System.String)"
```

- Destructors.

```
namespace Acme
{
   class Widget: IProcess
   {
      ~Widget() {...}
   }
}
"M:Acme.Widget.Finalize"
```

- Methods.

```
namespace Acme
{
   struct ValueType
   {
      public void M(int i) {...}
   }
   class Widget: IProcess
   {
      public class NestedClass
      {
         public void M(int i) {...}
      }
      public static void M0() {...}
      public void M1(char c, out float f, ref ValueType v) {...}
      public void M2(short[] x1, int[,] x2, long[][] x3) {...}
      public void M3(long[][] x3, Widget[][,,] x4) {...}
      public unsafe void M4(char *pc, Color **pf) {...}
      public unsafe void M5(void *pv, double *[][,] pd) {...}
      public void M6(int i, params object[] args) {...}
   }
}
"M:Acme.ValueType.M(System.Int32)"
"M:Acme.Widget.NestedClass.M(System.Int32)"
"M:Acme.Widget.M0"
"M:Acme.Widget.M1(System.Char,System.Single@,Acme.ValueType@)"
"M:Acme.Widget.M2(System.Int16[],System.Int32[0:,0:],System.Int64[][])"
```

```
"M:Acme.Widget.M3(System.Int64[][],Acme.Widget[0:,0:,0:][])"
"M:Acme.Widget.M4(System.Char*,Color**)"
"M:Acme.Widget.M5(System.Void*,System.Double*[0:,0:][])"
"M:Acme.Widget.M6(System.Int32,System.Object[])"
```

- Properties and indexers.

```
namespace Acme
{
   class Widget: IProcess
   {
      public int Width { get {...} set {...} }
      public int this[int i] { get {...} set {...} }
      public int this[string s, int i] { get {...} set {...} }
   }
}
"P:Acme.Widget.Width"
"P:Acme.Widget.Item(System.Int32)"
"P:Acme.Widget.Item(System.String,System.Int32)"
```

- Events.

```
namespace Acme
{
   class Widget: IProcess
   {
      public event Del AnEvent;
   }
}
"E:Acme.Widget.AnEvent"
```

- Unary operators.

```
namespace Acme
{
   class Widget: IProcess
   {
      public static Widget operator+(Widget x) {...}
   }
}
"M:Acme.Widget.op_UnaryPlus(Acme.Widget)"
```

The complete set of unary operator function names used is as follows: `op_UnaryPlus`, `op_UnaryNegation`, `op_LogicalNot`, `op_OnesComplement`, `op_Increment`, `op_Decrement`, `op_True`, and `op_False`.

- Binary operators.

```
namespace Acme
{
   class Widget: IProcess
   {
      public static Widget operator+(Widget x1, Widget x2) {...}
   }
}
"M:Acme.Widget.op_Addition(Acme.Widget,Acme.Widget)"
```

The complete set of binary operator function names used is as follows: `op_Addition`, `op_Subtraction`, `op_Multiply`, `op_Division`, `op_Modulus`, `op_BitwiseAnd`, `op_BitwiseOr`, `op_ExclusiveOr`, `op_LeftShift`, `op_RightShift`, `op_Equality`, `op_Inequality`, `op_LessThan`, `op_LessThanOrEqual`, `op_GreaterThan`, and `op_GreaterThanOrEqual`.

- Conversion operators have a trailing "~" followed by the return type.

```
namespace Acme
{
   class Widget: IProcess
   {
      public static explicit operator int(Widget x) {...}
      public static implicit operator long(Widget x) {...}
   }
}
"M:Acme.Widget.op_Explicit(Acme.Widget)~System.Int32"
"M:Acme.Widget.op_Implicit(Acme.Widget)~System.Int64"
```

B.4 An example

The following topics provide an example.

B.4.1 C# source code

The following example shows the source code of a `Point` class:

```
namespace Graphics
{

/// <remarks>Class <c>Point</c> models a point in a two-dimensional plane.
/// </remarks>
public class Point
{
   /// <summary>Instance variable <c>x</c> represents the point's
   ///   x-coordinate.</summary>
   private int x;
   /// <summary>Instance variable <c>y</c> represents the point's
   ///   y-coordinate.</summary>
   private int y;
   /// <value>Property <c>X</c> represents the point's x-coordinate.</value>
   public int X
   {
      get { return x; }
      set { x = value; }
   }
   /// <value>Property <c>Y</c> represents the point's y-coordinate.</value>
   public int Y
   {
      get { return y; }
      set { y = value; }
   }
   /// <summary>This constructor initializes the new Point to
   ///   (0,0).</summary>
   public Point() : this(0,0) {}
   /// <summary>This constructor initializes the new Point to
   ///   (<paramref name="xor"/>,<paramref name="yor"/>).</summary>
   /// <param><c>xor</c> is the new Point's x-coordinate.</param>
   /// <param><c>yor</c> is the new Point's y-coordinate.</param>
   public Point(int xor, int yor) {
      X = xor;
      Y = yor;
   }
   /// <summary>This method changes the point's location to
   ///   the given coordinates.</summary>
   /// <param><c>xor</c> is the new x-coordinate.</param>
   /// <param><c>yor</c> is the new y-coordinate.</param>
   /// <see cref="Translate"/>
   public void Move(int xor, int yor) {
      X = xor;
      Y = yor;
   }
   /// <summary>This method changes the point's location by
   ///   the given x- and y-offsets.
   /// <example>For example:
   /// <code>
   ///   Point p = new Point(3,5);
   ///   p.Translate(-1,3);
   /// </code>
   /// results in <c>p</c>'s having the value (2,8).
   /// </example>
   /// </summary>
   /// <param><c>xor</c> is the relative x-offset.</param>
   /// <param><c>yor</c> is the relative y-offset.</param>
   /// <see cref="Move"/>
   public void Translate(int xor, int yor) {
      X += xor;
      Y += yor;
   }
```

```
    /// <summary>This method determines whether two Points have the same
    ///   location.</summary>
    /// <param><c>o</c> is the object to be compared to the current object.
    /// </param>
    /// <returns>True if the Points have the same location and they have
    ///   the exact same type; otherwise, false.</returns>
    /// <seealso cref="operator=="/>
    /// <seealso cref="operator!="/>
    public override bool Equals(object o) {
       if (o == null) {
          return false;
       }
       if (this == o) {
          return true;
       }
       if (GetType() == o.GetType()) {
          Point p = (Point)o;
          return (X == p.X) && (Y == p.Y);
       }
       return false;
    }
    /// <summary>Report a point's location as a string.</summary>
    /// <returns>A string representing a point's location, in the form (x,y),
    ///   without any leading, training, or embedded whitespace.</returns>
    public override string ToString() {
       return "(" + X + "," + Y + ")";
    }
    /// <summary>This operator determines whether two Points have the same
    ///   location.</summary>
    /// <param><c>p1</c> is the first Point to be compared.</param>
    /// <param><c>p2</c> is the second Point to be compared.</param>
    /// <returns>True if the Points have the same location and they have
    ///   the exact same type; otherwise, false.</returns>
    /// <seealso cref="Equals"/>
    /// <seealso cref="operator!="/>
    public static bool operator==(Point p1, Point p2) {
       if ((object)p1 == null || (object)p2 == null) {
          return false;
       }

       if (p1.GetType() == p2.GetType()) {
          return (p1.X == p2.X) && (p1.Y == p2.Y);
       }

       return false;
    }
    /// <summary>This operator determines whether two Points have the same
    ///   location.</summary>
    /// <param><c>p1</c> is the first Point to be compared.</param>
    /// <param><c>p2</c> is the second Point to be compared.</param>
    /// <returns>True if the Points do not have the same location and the
    ///   exact same type; otherwise, false.</returns>
    /// <seealso cref="Equals"/>
    /// <seealso cref="operator=="/>
    public static bool operator!=(Point p1, Point p2) {
       return !(p1 == p2);
    }
    /// <summary>This is the entry point of the Point class testing
    /// program.
    /// <para>This program tests each method and operator, and
    /// is intended to be run after any non-trvial maintenance has
    /// been performed on the Point class.</para></summary>
    public static void Main() {
       // class test code goes here
    }
 }
 }
```

B.4.2 Resulting XML

Here is the output produced by one documentation generator when given the source code for class `Point`, shown above:

```
<?xml version="1.0"?>
<doc>
    <assembly>
        <name>Point</name>
    </assembly>
    <members>
        <member name="T:Graphics.Point">
            <remarks>Class <c>Point</c> models a point in a two-dimensional
            plane.
            </remarks>
        </member>
        <member name="F:Graphics.Point.x">
            <summary>Instance variable <c>x</c> represents the point's
            x-coordinate.</summary>
        </member>
        <member name="F:Graphics.Point.y">
            <summary>Instance variable <c>y</c> represents the point's
            y-coordinate.</summary>
        </member>
        <member name="M:Graphics.Point.#ctor">
            <summary>This constructor initializes the new Point to
        (0,0).</summary>
        </member>
        <member name="M:Graphics.Point.#ctor(System.Int32,System.Int32)">
            <summary>This constructor initializes the new Point to
            (<paramref name="xor"/>,<paramref name="yor"/>).</summary>
            <param><c>xor</c> is the new Point's x-coordinate.</param>
            <param><c>yor</c> is the new Point's y-coordinate.</param>
        </member>
        <member name="M:Graphics.Point.Move(System.Int32,System.Int32)">
            <summary>This method changes the point's location to
            the given coordinates.</summary>
            <param><c>xor</c> is the new x-coordinate.</param>
            <param><c>yor</c> is the new y-coordinate.</param>
            <see cref="M:Graphics.Point.Translate(System.Int32,System.Int32)"/>
        </member>
        <member
            name="M:Graphics.Point.Translate(System.Int32,System.Int32)">
            <summary>This method changes the point's location by
            the given x- and y-offsets.
            <example>For example:
            <code>
            Point p = new Point(3,5);
            p.Translate(-1,3);
            </code>
            results in <c>p</c>'s having the value (2,8).
            </example>
            </summary>
            <param><c>xor</c> is the relative x-offset.</param>
            <param><c>yor</c> is the relative y-offset.</param>
            <see cref="M:Graphics.Point.Move(System.Int32,System.Int32)"/>
        </member>
        <member name="M:Graphics.Point.Equals(System.Object)">
            <summary>This method determines whether two Points have the same
            location.</summary>
            <param><c>o</c> is the object to be compared to the current
            object.
            </param>
            <returns>True if the Points have the same location and they have
            the exact same type; otherwise, false.</returns>
            <seealso
       cref="M:Graphics.Point.op_Equality(Graphics.Point,Graphics.Point)"/>
```

```
            <seealso
      cref="M:Graphics.Point.op_Inequality(Graphics.Point,Graphics.Point)"/>
        </member>
        <member name="M:Graphics.Point.ToString">
            <summary>Report a point's location as a string.</summary>
            <returns>A string representing a point's location, in the form
            (x,y),
            without any leading, training, or embedded whitespace.</returns>
        </member>
        <member
       name="M:Graphics.Point.op_Equality(Graphics.Point,Graphics.Point)">
            <summary>This operator determines whether two Points have the
            same
            location.</summary>
            <param><c>p1</c> is the first Point to be compared.</param>
            <param><c>p2</c> is the second Point to be compared.</param>
            <returns>True if the Points have the same location and they have
            the exact same type; otherwise, false.</returns>
            <seealso cref="M:Graphics.Point.Equals(System.Object)"/>
            <seealso
     cref="M:Graphics.Point.op_Inequality(Graphics.Point,Graphics.Point)"/>
        </member>
        <member
      name="M:Graphics.Point.op_Inequality(Graphics.Point,Graphics.Point)">
            <summary>This operator determines whether two Points have the
            same
            location.</summary>
            <param><c>p1</c> is the first Point to be compared.</param>
            <param><c>p2</c> is the second Point to be compared.</param>
            <returns>True if the Points do not have the same location and
            the
            exact same type; otherwise, false.</returns>
            <seealso cref="M:Graphics.Point.Equals(System.Object)"/>
            <seealso
      cref="M:Graphics.Point.op_Equality(Graphics.Point,Graphics.Point)"/>
        </member>
        <member name="M:Graphics.Point.Main">
            <summary>This is the entry point of the Point class testing
            program.
            <para>This program tests each method and operator, and
            is intended to be run after any non-trvial maintenance has
            been performed on the Point class.</para></summary>
        </member>
        <member name="P:Graphics.Point.X">
            <value>Property <c>X</c> represents the point's
            x-coordinate.</value>
        </member>
        <member name="P:Graphics.Point.Y">
            <value>Property <c>Y</c> represents the point's
            y-coordinate.</value>
        </member>
    </members>
</doc>
```

C. Grammar

This appendix contains summaries of the lexical and syntactic grammars found in the main document, and of the grammar extensions for unsafe code. Grammar productions appear here in the same order that they appear in the main document.

C.1 Lexical grammar

input:
  input-section$_{opt}$
input-section:
  input-section-part
  input-section *input-section-part*
input-section-part:
  input-elements$_{opt}$ *new-line*
  pp-directive
input-elements:
  input-element
  input-elements *input-element*
input-element:
  whitespace
  comment
  token

C.1.1 Line terminators

new-line:
  Carriage return character (U+000D)
  Line feed character (U+000A)
  Carriage return character (U+000D) followed by line feed character (U+000A)
  Line separator character (U+2028)
  Paragraph separator character (U+2029)

C.1.2 White space

whitespace:
  Any character with Unicode class Zs
  Horizontal tab character (U+0009)
  Vertical tab character (U+000B)
  Form feed character (U+000C)

C.1.3 Comments

comment:
  single-line-comment
  delimited-comment
single-line-comment:
  // *input-characters*$_{opt}$
input-characters:
  input-character
  input-characters *input-character*
input-character:
  Any Unicode character except a *new-line-character*
new-line-character:
  Carriage return character (U+000D)
  Line feed character (U+000A)
  Line separator character (U+2028)
  Paragraph separator character (U+2029)
delimited-comment:
  /* *delimited-comment-characters*$_{opt}$ */
delimited-comment-characters:
  delimited-comment-character
  delimited-comment-characters *delimited-comment-character*

delimited-comment-character:
  not-asterisk
  * *not-slash*
not-asterisk:
  Any Unicode character except *
not-slash:
  Any Unicode character except /

C.1.4 Tokens

token:
  identifier
  keyword
  integer-literal
  real-literal
  character-literal
  string-literal
  operator-or-punctuator

C.1.5 Unicode character escape sequences

unicode-escape-sequence:
  \u *hex-digit* *hex-digit* *hex-digit* *hex-digit*
  \U *hex-digit* *hex-digit* *hex-digit* *hex-digit* *hex-digit* *hex-digit* *hex-digit* *hex-digit*

C.1.6 Identifiers

identifier:
  available-identifier
  @ *identifier-or-keyword*
available-identifier:
  An *identifier-or-keyword* that is not a *keyword*
identifier-or-keyword:
  identifier-start-character *identifier-part-characters*$_{opt}$
identifier-start-character:
  letter-character
  _ (the underscore character U+005F)
identifier-part-characters:
  identifier-part-character
  identifier-part-characters *identifier-part-character*
identifier-part-character:
  letter-character
  decimal-digit-character
  connecting-character
  combining-character
  formatting-character
letter-character:
  A Unicode character of classes Lu, Ll, Lt, Lm, Lo, or Nl
  A *unicode-escape-sequence* representing a character of classes Lu, Ll, Lt, Lm, Lo, or Nl
combining-character:
  A Unicode character of classes Mn or Mc
  A *unicode-escape-sequence* representing a character of classes Mn or Mc
decimal-digit-character:
  A Unicode character of the class Nd
  A *unicode-escape-sequence* representing a character of the class Nd
connecting-character:
  A Unicode character of the class Pc
  A *unicode-escape-sequence* representing a character of the class Pc
formatting-character:
  A Unicode character of the class Cf
  A *unicode-escape-sequence* representing a character of the class Cf

C.1.7 Keywords

keyword: one of

abstract	as	base	bool	break	byte	case
catch	char	checked	class	const	continue	decimal
default	delegate	do	double	else	enum	event
explicit	extern	false	finally	fixed	float	for
foreach	goto	if	implicit	in	int	interface
internal	is	lock	long	namespace	new	null
object	operator	out	override	params	private	protected
public	readonly	ref	return	sbyte	sealed	short
sizeof	stackalloc	static	string	struct	switch	this
throw	true	try	typeof	uint	ulong	unchecked
unsafe	ushort	using	virtual	void	volatile	while

C.1.8 Literals

literal:
  boolean-literal
  integer-literal
  real-literal
  character-literal
  string-literal
  null-literal
boolean-literal:
  true
  false
integer-literal:
  decimal-integer-literal
  hexadecimal-integer-literal
decimal-integer-literal:
  decimal-digits *integer-type-suffix*$_{opt}$
decimal-digits:
  decimal-digit
  decimal-digits *decimal-digit*
decimal-digit: one of
  0 1 2 3 4 5 6 7 8 9
integer-type-suffix: one of
  U u L l UL Ul uL ul LU Lu lU lu
hexadecimal-integer-literal:
  0x *hex-digits* *integer-type-suffix*$_{opt}$
  0X *hex-digits* *integer-type-suffix*$_{opt}$
hex-digits:
  hex-digit
  hex-digits *hex-digit*
hex-digit: one of
  0 1 2 3 4 5 6 7 8 9 A B C D E F a b c d e f
real-literal:
  decimal-digits . *decimal-digits* *exponent-part*$_{opt}$ *real-type-suffix*$_{opt}$
  . *decimal-digits* *exponent-part*$_{opt}$ *real-type-suffix*$_{opt}$
  decimal-digits *exponent-part* *real-type-suffix*$_{opt}$
  decimal-digits *real-type-suffix*
exponent-part:
  e *sign*$_{opt}$ *decimal-digits*
  E *sign*$_{opt}$ *decimal-digits*
sign: one of
  + -
real-type-suffix: one of
  F f D d M m
character-literal:

' *character* '

character:
single-character
simple-escape-sequence
hexadecimal-escape-sequence
unicode-escape-sequence

single-character:
Any character except ' (U+0027), \ (U+005C), and *new-line-character*

simple-escape-sequence: one of
\' \" \\ \0 \a \b \f \n \r \t \v

hexadecimal-escape-sequence:
\x *hex-digit* $hex\text{-}digit_{opt}$ $hex\text{-}digit_{opt}$ $hex\text{-}digit_{opt}$

string-literal:
regular-string-literal
verbatim-string-literal

regular-string-literal:
" $regular\text{-}string\text{-}literal\text{-}characters_{opt}$ "

regular-string-literal-characters:
regular-string-literal-character
regular-string-literal-characters *regular-string-literal-character*

regular-string-literal-character:
single-regular-string-literal-character
simple-escape-sequence
hexadecimal-escape-sequence
unicode-escape-sequence

single-regular-string-literal-character:
Any character except " (U+0022), \ (U+005C), and *new-line-character*

verbatim-string-literal:
@" $verbatim\text{-}string\text{-}literal\text{-}characters_{opt}$ "

verbatim-string-literal-characters:
verbatim-string-literal-character
verbatim-string-literal-characters *verbatim-string-literal-character*

verbatim-string-literal-character:
single-verbatim-string-literal-character
quote-escape-sequence

single-verbatim-string-literal-character:
any character except "

quote-escape-sequence:
""

null-literal:
null

C.1.9 Operators and punctuators

operator-or-punctuator: one of
{ } [ ] ( ) . , : ;
+ - * / % & | ^ ! ~
= < > ? ++ -- && || << >>
== != <= >= += -= *= /= %= &=
|= ^= <<= >>= ->

C.1.10 Pre-processing directives

pp-directive:
pp-declaration
pp-conditional
pp-line
pp-diagnostic
pp-region

pp-new-line:
$whitespace_{opt}$ $single\text{-}line\text{-}comment_{opt}$ *new-line*

conditional-symbol:

Any *identifier-or-keyword* except true or false

pp-expression:
whitespace$_{opt}$ *pp-or-expression* *whitespace*$_{opt}$

pp-or-expression:
pp-and-expression
pp-or-expression *whitespace*$_{opt}$ || *whitespace*$_{opt}$ *pp-and-expression*

pp-and-expression:
pp-equality-expression
pp-and-expression *whitespace*$_{opt}$ && *whitespace*$_{opt}$ *pp-equality-expression*

pp-equality-expression:
pp-unary-expression
pp-equality-expression *whitespace*$_{opt}$ == *whitespace*$_{opt}$ *pp-unary-expression*
pp-equality-expression *whitespace*$_{opt}$!= *whitespace*$_{opt}$ *pp-unary-expression*

pp-unary-expression:
pp-primary-expression
! *whitespace*$_{opt}$ *pp-unary-expression*

pp-primary-expression:
true
false
conditional-symbol
(*whitespace*$_{opt}$ *pp-expression* *whitespace*$_{opt}$)

pp-declaration:
whitespace$_{opt}$ # *whitespace*$_{opt}$ define *whitespace* *conditional-symbol* *pp-new-line*
whitespace$_{opt}$ # *whitespace*$_{opt}$ undef *whitespace* *conditional-symbol* *pp-new-line*

pp-conditional:
pp-if-section *pp-elif-sections*$_{opt}$ *pp-else-section*$_{opt}$ *pp-endif*

pp-if-section:
whitespace$_{opt}$ # *whitespace*$_{opt}$ if *whitespace* *pp-expression* *pp-new-line* *conditional-section*$_{opt}$

pp-elif-sections:
pp-elif-section
pp-elif-sections *pp-elif-section*

pp-elif-section:
whitespace$_{opt}$ # *whitespace*$_{opt}$ elif *whitespace* *pp-expression* *pp-new-line* *conditional-section*$_{opt}$

pp-else-section:
whitespace$_{opt}$ # *whitespace*$_{opt}$ else *pp-new-line* *conditional-section*$_{opt}$

pp-endif:
whitespace$_{opt}$ # *whitespace*$_{opt}$ endif *pp-new-line*

conditional-section:
input-section
skipped-section

skipped-section:
skipped-section-part
skipped-section *skipped-section-part*

skipped-section-part:
skipped-characters$_{opt}$ *new-line*
pp-directive

skipped-characters:
whitespace$_{opt}$ *not-number-sign* *input-characters*$_{opt}$

not-number-sign:
Any *input-character* except #

pp-line:
whitespace$_{opt}$ # *whitespace*$_{opt}$ line *whitespace* *line-indicator* *pp-new-line*

line-indicator:
decimal-digits *whitespace* *file-name*
decimal-digits
default

file-name:
" *file-name-characters* "

file-name-characters:

file-name-character
file-name-characters *file-name-character*
file-name-character:
Any *input-character* except "
pp-diagnostic:
whitespace$_{opt}$ # *whitespace*$_{opt}$ error *pp-message*
whitespace$_{opt}$ # *whitespace*$_{opt}$ warning *pp-message*
pp-message:
new-line
whitespace *input-characters*$_{opt}$ *new-line*
pp-region:
pp-start-region *conditional-section*$_{opt}$ *pp-end-region*
pp-start-region:
whitespace$_{opt}$ # *whitespace*$_{opt}$ region *pp-message*
pp-end-region:
whitespace$_{opt}$ # *whitespace*$_{opt}$ endregion *pp-message*

C.2 Syntactic grammar

C.2.1 Basic concepts

namespace-name:
namespace-or-type-name
type-name:
namespace-or-type-name
namespace-or-type-name:
identifier
namespace-or-type-name . *identifier*

C.2.2 Types

type:
value-type
reference-type
value-type:
struct-type
enum-type
struct-type:
type-name
simple-type
simple-type:
numeric-type
bool
numeric-type:
integral-type
floating-point-type
decimal
integral-type:
sbyte
byte
short
ushort
int
uint
long
ulong
char
floating-point-type:
float
double
enum-type:
type-name

reference-type:
  class-type
  interface-type
  array-type
  delegate-type
class-type:
  type-name
  object
  string
interface-type:
  type-name
array-type:
  non-array-type *rank-specifiers*
non-array-type:
  type
rank-specifiers:
  rank-specifier
  rank-specifiers *rank-specifier*
rank-specifier:
  [*dim-separators*$_{opt}$]
dim-separators:
  ,
  dim-separators ,
delegate-type:
  type-name

C.2.3 Variables

variable-reference:
  expression

C.2.4 Expressions

argument-list:
  argument
  argument-list , *argument*
argument:
  expression
  ref *variable-reference*
  out *variable-reference*
primary-expression:
  primary-no-array-creation-expression
  array-creation-expression
primary-no-array-creation-expression:
  literal
  simple-name
  parenthesized-expression
  member-access
  invocation-expression
  element-access
  this-access
  base-access
  post-increment-expression
  post-decrement-expression
  object-creation-expression
  delegate-creation-expression
  typeof-expression
  sizeof-expression
  checked-expression
  unchecked-expression
simple-name:
  identifier

parenthesized-expression:
(*expression*)
member-access:
primary-expression . *identifier*
predefined-type . *identifier*
predefined-type: one of
bool byte char decimal double float int long
object sbyte short string uint ulong ushort
invocation-expression:
primary-expression (*argument-list*$_{opt}$)
element-access:
primary-no-array-creation-expression [*expression-list*]
expression-list:
expression
expression-list , *expression*
this-access:
this
base-access:
base . *identifier*
base [*expression-list*]
post-increment-expression:
primary-expression ++
post-decrement-expression:
primary-expression --
object-creation-expression:
new *type* (*argument-list*$_{opt}$)
array-creation-expression:
new *non-array-type* [*expression-list*] *rank-specifiers*$_{opt}$ *array-initializer*$_{opt}$
new *array-type* *array-initializer*
delegate-creation-expression:
new *delegate-type* (*expression*)
typeof-expression:
typeof (*type*)
typeof (void)
checked-expression:
checked (*expression*)
unchecked-expression:
unchecked (*expression*)
unary-expression:
primary-expression
+ *unary-expression*
- *unary-expression*
! *unary-expression*
~ *unary-expression*
* *unary-expression*
pre-increment-expression
pre-decrement-expression
cast-expression
pre-increment-expression:
++ *unary-expression*
pre-decrement-expression:
-- *unary-expression*
cast-expression:
(*type*) *unary-expression*
multiplicative-expression:
unary-expression
multiplicative-expression * *unary-expression*
multiplicative-expression / *unary-expression*
multiplicative-expression % *unary-expression*
additive-expression:
multiplicative-expression

additive-expression + *multiplicative-expression*
additive-expression – *multiplicative-expression*
shift-expression:
additive-expression
shift-expression << *additive-expression*
shift-expression >> *additive-expression*
relational-expression:
shift-expression
relational-expression < *shift-expression*
relational-expression > *shift-expression*
relational-expression <= *shift-expression*
relational-expression >= *shift-expression*
relational-expression is *type*
relational-expression as *type*
equality-expression:
relational-expression
equality-expression == *relational-expression*
equality-expression != *relational-expression*
and-expression:
equality-expression
and-expression & *equality-expression*
exclusive-or-expression:
and-expression
exclusive-or-expression ^ *and-expression*
inclusive-or-expression:
exclusive-or-expression
inclusive-or-expression | *exclusive-or-expression*
conditional-and-expression:
inclusive-or-expression
conditional-and-expression && *inclusive-or-expression*
conditional-or-expression:
conditional-and-expression
conditional-or-expression || *conditional-and-expression*
conditional-expression:
conditional-or-expression
conditional-or-expression ? *expression* : *expression*
assignment:
unary-expression *assignment-operator* *expression*
assignment-operator: one of
= += -= *= /= %= &= |= ^= <<= >>=
expression:
conditional-expression
assignment
constant-expression:
expression
boolean-expression:
expression

C.2.5 Statements

statement:
labeled-statement
declaration-statement
embedded-statement
embedded-statement:
block
empty-statement
expression-statement
selection-statement
iteration-statement
jump-statement
try-statement

checked-statement
unchecked-statement
lock-statement
using-statement

block:
{ *statement-list*$_{opt}$ }

statement-list:
statement
statement-list *statement*

empty-statement:
;

labeled-statement:
identifier : *statement*

declaration-statement:
local-variable-declaration ;
local-constant-declaration ;

local-variable-declaration:
type *local-variable-declarators*

local-variable-declarators:
local-variable-declarator
local-variable-declarators , *local-variable-declarator*

local-variable-declarator:
identifier
identifier = *local-variable-initializer*

local-variable-initializer:
expression
array-initializer

local-constant-declaration:
const *type* *constant-declarators*

constant-declarators:
constant-declarator
constant-declarators , *constant-declarator*

constant-declarator:
identifier = *constant-expression*

expression-statement:
statement-expression ;

statement-expression:
invocation-expression
object-creation-expression
assignment
post-increment-expression
post-decrement-expression
pre-increment-expression
pre-decrement-expression

selection-statement:
if-statement
switch-statement

if-statement:
if (*boolean-expression*) *embedded-statement*
if (*boolean-expression*) *embedded-statement* else *embedded-statement*

boolean-expression:
expression

switch-statement:
switch (*expression*) *switch-block*

switch-block:
{ *switch-sections*$_{opt}$ }

switch-sections:
switch-section
switch-sections *switch-section*

switch-section:
switch-labels *statement-list*

switch-labels:

switch-label
switch-labels *switch-label*
switch-label:
case *constant-expression* :
default :
iteration-statement:
while-statement
do-statement
for-statement
foreach-statement
while-statement:
while (*boolean-expression*) *embedded-statement*
do-statement:
do *embedded-statement* while (*boolean-expression*) ;
for-statement:
for (*for-initializer*$_{opt}$; *for-condition*$_{opt}$; *for-iterator*$_{opt}$) *embedded-statement*
for-initializer:
local-variable-declaration
statement-expression-list
for-condition:
boolean-expression
for-iterator:
statement-expression-list
statement-expression-list:
statement-expression
statement-expression-list , *statement-expression*
foreach-statement:
foreach (*type* *identifier* in *expression*) *embedded-statement*
jump-statement:
break-statement
continue-statement
goto-statement
return-statement
throw-statement
break-statement:
break ;
continue-statement:
continue ;
goto-statement:
goto *identifier* ;
goto case *constant-expression* ;
goto default ;
return-statement:
return *expression*$_{opt}$;
throw-statement:
throw *expression*$_{opt}$;
try-statement:
try *block* *catch-clauses*
try *block* *finally-clause*
try *block* *catch-clauses* *finally-clause*
catch-clauses:
specific-catch-clauses *general-catch-clause*$_{opt}$
specific-catch-clauses$_{opt}$ *general-catch-clause*
specific-catch-clauses:
specific-catch-clause
specific-catch-clauses *specific-catch-clause*
specific-catch-clause:
catch (*class-type* *identifier*$_{opt}$) *block*
general-catch-clause:
catch *block*
finally-clause:

finally *block*
checked-statement:
checked *block*
unchecked-statement:
unchecked *block*
lock-statement:
lock (*expression*) *embedded-statement*
using-statement:
using (*resource-acquisition*) *embedded-statement*
resource-acquisition:
local-variable-declaration
expression

C.2.6 Namespaces

compilation-unit:
using-directives$_{opt}$ *global-attributes*$_{opt}$ *namespace-member-declarations*$_{opt}$
namespace-declaration:
namespace *qualified-identifier* *namespace-body* ;$_{opt}$
qualified-identifier:
identifier
qualified-identifier . *identifier*
namespace-body:
{ *using-directives*$_{opt}$ *namespace-member-declarations*$_{opt}$ }
using-directives:
using-directive
using-directives *using-directive*
using-directive:
using-alias-directive
using-namespace-directive
using-alias-directive:
using *identifier* = *namespace-or-type-name* ;
using-namespace-directive:
using *namespace-name* ;
namespace-member-declarations:
namespace-member-declaration
namespace-member-declarations *namespace-member-declaration*
namespace-member-declaration:
namespace-declaration
type-declaration
type-declaration:
class-declaration
struct-declaration
interface-declaration
enum-declaration
delegate-declaration

C.2.7 Classes

class-declaration:
attributes$_{opt}$ *class-modifiers*$_{opt}$ class *identifier* *class-base*$_{opt}$ *class-body* ;$_{opt}$
class-modifiers:
class-modifier
class-modifiers *class-modifier*
class-modifier:
new
public
protected
internal
private
abstract
sealed

class-base:
: *class-type*
: *interface-type-list*
: *class-type* , *interface-type-list*
interface-type-list:
interface-type
interface-type-list , *interface-type*
class-body:
{ *class-member-declarations*$_{opt}$ }
class-member-declarations:
class-member-declaration
class-member-declarations *class-member-declaration*
class-member-declaration:
constant-declaration
field-declaration
method-declaration
property-declaration
event-declaration
indexer-declaration
operator-declaration
constructor-declaration
destructor-declaration
static-constructor-declaration
type-declaration
constant-declaration:
attributes$_{opt}$ *constant-modifiers*$_{opt}$ const *type* *constant-declarators* ;
constant-modifiers:
constant-modifier
constant-modifiers *constant-modifier*
constant-modifier:
new
public
protected
internal
private
constant-declarators:
constant-declarator
constant-declarators , *constant-declarator*
constant-declarator:
identifier = *constant-expression*
field-declaration:
attributes$_{opt}$ *field-modifiers*$_{opt}$ *type* *variable-declarators* ;
field-modifiers:
field-modifier
field-modifiers *field-modifier*
field-modifier:
new
public
protected
internal
private
static
readonly
volatile
variable-declarators:
variable-declarator
variable-declarators , *variable-declarator*
variable-declarator:
identifier
identifier = *variable-initializer*
variable-initializer:

expression
array-initializer

method-declaration:
method-header *method-body*

method-header:
attributes$_{opt}$ *method-modifiers*$_{opt}$ *return-type* *member-name* (*formal-parameter-list*$_{opt}$)

method-modifiers:
method-modifier
method-modifiers *method-modifier*

method-modifier:
new
public
protected
internal
private
static
virtual
sealed
override
abstract
extern

return-type:
type
void

member-name:
identifier
interface-type . *identifier*

method-body:
block
;

formal-parameter-list:
fixed-parameters
fixed-parameters , *parameter-array*
parameter-array

fixed-parameters:
fixed-parameter
fixed-parameters , *fixed-parameter*

fixed-parameter:
attributes$_{opt}$ *parameter-modifier*$_{opt}$ *type* *identifier*

parameter-modifier:
ref
out

parameter-array:
attributes$_{opt}$ params *array-type* *identifier*

property-declaration:
attributes$_{opt}$ *property-modifiers*$_{opt}$ *type* *member-name* { *accessor-declarations* }

property-modifiers:
property-modifier
property-modifiers *property-modifier*

property-modifier:
new
public
protected
internal
private
static
virtual
sealed
override
abstract
extern

member-name:
  identifier
  interface-type . *identifier*
accessor-declarations:
  get-accessor-declaration *set-accessor-declaration*$_{opt}$
  set-accessor-declaration *get-accessor-declaration*$_{opt}$
get-accessor-declaration:
  attributes$_{opt}$ get *accessor-body*
set-accessor-declaration:
  attributes$_{opt}$ set *accessor-body*
accessor-body:
  block
  ;
event-declaration:
  attributes$_{opt}$ *event-modifiers*$_{opt}$ event *type* *variable-declarators* ;
  attributes$_{opt}$ *event-modifiers*$_{opt}$ event *type* *member-name* { *event-accessor-declarations* }
event-modifiers:
  event-modifier
  event-modifiers *event-modifier*
event-modifier:
  new
  public
  protected
  internal
  private
  static
  virtual
  sealed
  override
  abstract
  extern
event-accessor-declarations:
  add-accessor-declaration *remove-accessor-declaration*
  remove-accessor-declaration *add-accessor-declaration*
add-accessor-declaration:
  attributes$_{opt}$ add *block*
remove-accessor-declaration:
  attributes$_{opt}$ remove *block*
indexer-declaration:
  attributes$_{opt}$ *indexer-modifiers*$_{opt}$ *indexer-declarator* { *accessor-declarations* }
indexer-modifiers:
  indexer-modifier
  indexer-modifiers *indexer-modifier*
indexer-modifier:
  new
  public
  protected
  internal
  private
  virtual
  sealed
  override
  abstract
  extern
indexer-declarator:
  type this [*formal-parameter-list*]
  type *interface-type* . this [*formal-parameter-list*]
operator-declaration:
  attributes$_{opt}$ *operator-modifiers* *operator-declarator* *operator-body*
operator-modifiers:

operator-modifier
operator-modifiers *operator-modifier*
operator-modifier:
public
static
extern
operator-declarator:
unary-operator-declarator
binary-operator-declarator
conversion-operator-declarator
unary-operator-declarator:
type operator *overloadable-unary-operator* (*type* *identifier*)
overloadable-unary-operator: one of
+ - ! ~ ++ -- true false
binary-operator-declarator:
type operator *overloadable-binary-operator* (*type* *identifier* , *type* *identifier*)
overloadable-binary-operator: one of
+ - * / % & | ^ << >> == != > < >= <=
conversion-operator-declarator:
implicit operator *type* (*type* *identifier*)
explicit operator *type* (*type* *identifier*)
operator-body:
block
;
constructor-declaration:
attributes$_{opt}$ *constructor-modifiers*$_{opt}$ *constructor-declarator* *constructor-body*
constructor-modifiers:
constructor-modifier
constructor-modifiers *constructor-modifier*
constructor-modifier:
public
protected
internal
private
extern
constructor-declarator:
identifier (*formal-parameter-list*$_{opt}$) *constructor-initializer*$_{opt}$
constructor-initializer:
: base (*argument-list*$_{opt}$)
: this (*argument-list*$_{opt}$)
constructor-body:
block
;
static-constructor-declaration:
attributes$_{opt}$ *static-constructor-modifiers* *identifier* ( ) *static-constructor-body*
static-constructor-modifiers
extern$_{opt}$ static
static extern$_{opt}$
static-constructor-body:
block
;
destructor-declaration:
attributes$_{opt}$ extern$_{opt}$ ~ *identifier* ( ) *destructor-body*
destructor-body:
block
;

C.2.8 Structs

struct-declaration:
attributes$_{opt}$ *struct-modifiers*$_{opt}$ struct *identifier* *struct-interfaces*$_{opt}$ *struct-body* ;$_{opt}$

struct-modifiers:
  struct-modifier
  struct-modifiers *struct-modifier*
struct-modifier:
  new
  public
  protected
  internal
  private
struct-interfaces:
  : *interface-type-list*
struct-body:
  { *struct-member-declarations*$_{opt}$ }
struct-member-declarations:
  struct-member-declaration
  struct-member-declarations *struct-member-declaration*
struct-member-declaration:
  constant-declaration
  field-declaration
  method-declaration
  property-declaration
  event-declaration
  indexer-declaration
  operator-declaration
  constructor-declaration
  static-constructor-declaration
  type-declaration

C.2.9 Arrays

array-type:
  non-array-type *rank-specifiers*
non-array-type:
  type
rank-specifiers:
  rank-specifier
  rank-specifiers *rank-specifier*
rank-specifier:
  [*dim-separators*$_{opt}$]
dim-separators:
  ,
  dim-separators ,
array-initializer:
  { *variable-initializer-list*$_{opt}$ }
  { *variable-initializer-list* , }
variable-initializer-list:
  variable-initializer
  variable-initializer-list , *variable-initializer*
variable-initializer:
  expression
  array-initializer

C.2.10 Interfaces

interface-declaration:
  attributes$_{opt}$ *interface-modifiers*$_{opt}$ interface *identifier* *interface-base*$_{opt}$ *interface-body* ;$_{opt}$
interface-modifiers:
  interface-modifier
  interface-modifiers *interface-modifier*
interface-modifier:
  new
  public

protected
internal
private

interface-base:
: *interface-type-list*

interface-body:
{ *interface-member-declarations*$_{opt}$ }

interface-member-declarations:
interface-member-declaration
interface-member-declarations *interface-member-declaration*

interface-member-declaration:
interface-method-declaration
interface-property-declaration
interface-event-declaration
interface-indexer-declaration

interface-method-declaration:
attributes$_{opt}$ new$_{opt}$ *return-type* *identifier* (*formal-parameter-list*$_{opt}$) ;

interface-property-declaration:
attributes$_{opt}$ new$_{opt}$ *type* *identifier* { *interface-accessors* }

interface-accessors:
attributes$_{opt}$ get ;
attributes$_{opt}$ set ;
attributes$_{opt}$ get ; *attributes*$_{opt}$ set ;
attributes$_{opt}$ set ; *attributes*$_{opt}$ get ;

interface-event-declaration:
attributes$_{opt}$ new$_{opt}$ event *type* *identifier* ;

interface-indexer-declaration:
attributes$_{opt}$ new$_{opt}$ *type* this [*formal-parameter-list*] { *interface-accessors* }

C.2.11 Enums

enum-declaration:
attributes$_{opt}$ *enum-modifiers*$_{opt}$ enum *identifier* *enum-base*$_{opt}$ *enum-body* ;$_{opt}$

enum-base:
: *integral-type*

enum-body:
{ *enum-member-declarations*$_{opt}$ }
{ *enum-member-declarations* , }

enum-modifiers:
enum-modifier
enum-modifiers *enum-modifier*

enum-modifier:
new
public
protected
internal
private

enum-member-declarations:
enum-member-declaration
enum-member-declarations , *enum-member-declaration*

enum-member-declaration:
attributes$_{opt}$ *identifier*
attributes$_{opt}$ *identifier* = *constant-expression*

C.2.12 Delegates

delegate-declaration:
attributes$_{opt}$ *delegate-modifiers*$_{opt}$ delegate *return-type* *identifier* (*formal-parameter-list*$_{opt}$) ;

delegate-modifiers:
delegate-modifier
delegate-modifiers *delegate-modifier*

delegate-modifier:
  new
  public
  protected
  internal
  private

C.2.13 Attributes

global-attributes:
  global-attribute-sections
global-attribute-sections:
  global-attribute-section
  global-attribute-sections global-attribute-section
global-attribute-section:
  [global-attribute-target-specifier attribute-list]
  [global-attribute-target-specifier attribute-list ,]
global-attribute-target-specifier:
  global-attribute-target :
global-attribute-target:
  assembly
  module
attributes:
  attribute-sections
attribute-sections:
  attribute-section
  attribute-sections attribute-section
attribute-section:
  [*attribute-target-specifier*$_{\text{opt}}$ *attribute-list*]
  [*attribute-target-specifier*$_{\text{opt}}$ *attribute-list* ,]
attribute-target-specifier:
  attribute-target :
attribute-target:
  field
  event
  method
  param
  property
  return
  type
attribute-list:
  attribute
  attribute-list , *attribute*
attribute:
  attribute-name attribute-arguments$_{opt}$
attribute-name:
  type-name
attribute-arguments:
  (*positional-argument-list*$_{\text{opt}}$)
  (positional-argument-list , named-argument-list)
  (named-argument-list)
positional-argument-list:
  positional-argument
  positional-argument-list , positional-argument
positional-argument:
  attribute-argument-expression
named-argument-list:
  named-argument
  named-argument-list , named-argument
named-argument:
  identifier = attribute-argument-expression
attribute-argument-expression:

expression

C.3 Grammar extensions for unsafe code

class-modifier:
  ...
  unsafe
struct-modifier:
  ...
  unsafe
interface-modifier:
  ...
  unsafe
delegate-modifier:
  ...
  unsafe
field-modifier:
  ...
  unsafe
method-modifier:
  ...
  unsafe
property-modifier:
  ...
  unsafe
event-modifier:
  ...
  unsafe
indexer-modifier:
  ...
  unsafe
operator-modifier:
  ...
  unsafe
constructor-modifier:
  ...
  unsafe
destructor-declaration:
  attributes$_{opt}$ extern$_{opt}$ unsafe$_{opt}$ ~ *identifier* ( ) *destructor-body*
  attributes$_{opt}$ unsafe$_{opt}$ extern$_{opt}$ ~ *identifier* ( ) *destructor-body*
static-constructor-modifiers:
  extern$_{opt}$ unsafe$_{opt}$ *static*
  unsafe$_{opt}$ extern$_{opt}$ *static*
  extern$_{opt}$ *static* unsafe$_{opt}$
  unsafe$_{opt}$ *static* extern$_{opt}$
  static extern$_{opt}$ unsafe$_{op}$
  static unsafe$_{opt}$ extern$_{opt}$
embedded-statement:
  ...
  unsafe-statement
unsafe-statement:
  unsafe *block*
type:
  value-type
  reference-type
  pointer-type
pointer-type:
  unmanaged-type *
  void *
unmanaged-type:
  type

primary-no-array-creation-expression:
  ...
  pointer-member-access
  pointer-element-access
  sizeof-expression
unary-expression:
  ...
  pointer-indirection-expression
  addressof-expression
pointer-indirection-expression:
  * *unary-expression*
pointer-member-access:
  primary-expression -> *identifier*
pointer-element-access:
  primary-no-array-creation-expression [*expression*]
addressof-expression:
  & *unary-expression*
sizeof-expression:
  sizeof (*unmanaged-type*)
embedded-statement:
  ...
  fixed-statement
fixed-statement:
  fixed (*pointer-type* *fixed-pointer-declarators*) *embedded-statement*
fixed-pointer-declarators:
  fixed-pointer-declarator
  fixed-pointer-declarators , *fixed-pointer-declarator*
fixed-pointer-declarator:
  identifier = *fixed-pointer-initializer*
fixed-pointer-initializer:
  & *variable-reference*
  expression
variable-initializer:
  expression
  array-initializer
  stackalloc-initializer
stackalloc-initializer:
  stackalloc *unmanaged-type* [*expression*]

D. References

Unicode Consortium. *The Unicode Standard, Version 3.0*. Addison-Wesley, Reading, Massachusetts, 2000, ISBN 0-201-616335-5.

IEEE. *IEEE Standard for Binary Floating-Point Arithmetic*. ANSI/IEEE Standard 754-1985. Available from http://www.ieee.org.

ISO/IEC. *C++*. ANSI/ISO/IEC 14882:1998.

C# Programmer's Reference

The *C# Programmer's Reference* complements the *C# Language Specification* by providing tutorials, reference material, and topics on how to program in C#.

For information about	See
Learning the language	C# Tutorials
Keywords	C# Keywords
Operators	C# Operators
Attributes	C# Attributes
Conditional compilation	C# Preprocessor Directives
Types	Types Reference Tables
Arrays, Main method, properties, indexers, passing parameters, and other language features	C# Language Features
Documenting your code using XML	XML Documentation

See Also

C# Language Specification | Visual C# Samples | Visual C# Language

C# Tutorials

The C# tutorials provide an overview of the basics of the language and identify important language features. Each tutorial includes one or more sample programs. The tutorials discuss the sample code plus provide additional background information. They also link to the corresponding sample abstract topics, where you can download and run the sample programs.

> **Note** The C# tutorials do not teach how to create a graphical user interface for your program like a Windows Form or a Web Forms page. To get started in that area, see the Visual C# Walkthroughs.

Tutorial	Description	Level
Hello World	Shows a Hello World application.	Simple
Command Line Parameters	Shows simple command-line processing; also shows array indexing.	Simple
Arrays	Shows how to use arrays.	Simple
Properties	Shows how properties are declared and used; also demonstrates abstract properties.	Simple
Libraries	Shows how to use compiler options to create a DLL library from multiple source files; also, how to use them in other programs.	Simple
Versioning	Shows the use of **override** and **new** to support versioning.	Simple
Collection Classes	Shows how to make collection classes that can be used with **foreach**.	Intermediate
Structs	Shows how to use structs in C#.	Intermediate
Indexers	Shows how to use array notation to access an object.	Intermediate
Indexed Properties	Shows how to implement a class that uses indexed properties. Indexed properties allow you to use a class that represents an array-like collection of several different kinds of things.	Intermediate
User-Defined Conversions	Shows how to define conversions to and from user-defined types.	Intermediate
Operator Overloading	Shows how user-defined classes can overload operators.	Intermediate
Delegates	Shows how delegates are declared, mapped, and combined.	Intermediate
Events	Shows how to use events in C#.	Intermediate
Explicit Interface Implementation	Demonstrates how to explicitly implement interface members.	Intermediate
Conditional Methods	Demonstrates conditional methods, which provide a powerful mechanism by which calls to methods can be included or omitted depending on whether a symbol is defined.	Intermediate
XML Documentation	Shows how to document code by using XML.	Intermediate
Platform Invoke	Shows how to call platform invokes (exported DLL functions) from C#.	Advanced
COM Interop Part 1	Shows how to use C# to interoperate with COM objects. Covers a C# client.	Advanced
COM Interop Part 2	Shows how to use C# to interoperate with COM objects. Covers a C# server.	Advanced
Attributes	Shows how to create custom attribute classes, use them in code, and query them through reflection.	Advanced
Security	Discusses .NET Framework security and shows two ways to modify security permissions in C#: permission classes and permission attributes.	Advanced
Threading	Demonstrates various thread activities such as creating and executing a thread, synchronizing threads, interacting between threads, using a thread pool, and using a mutex object.	Advanced
Unsafe Code	Shows how to use pointers.	Advanced
OLE DB	Shows how to use OLE DB in C# by connecting to a Microsoft Access database.	Advanced

See Also

Tutorial Samples | C# Programmer's Reference

Hello World Tutorial

This tutorial shows several versions of a Hello World program in C#.

Sample Files

See Hello World Sample to download and build the sample files discussed in this tutorial.

Further Reading

Main

1.1 Getting started

3.3 Declarations

Tutorial

The following examples show different ways of writing the C# Hello World program.

Example 1

```
// Hello1.cs
public class Hello1
{
   public static void Main()
   {
      System.Console.WriteLine("Hello, World!");
   }
}
```

Output

```
Hello, World!
```

Code Discussion

- Every `Main` method must be contained inside a class (`Hello1` in this case).
- The **System.Console** class contains a **WriteLine** method that can be used to display a string to the console.

Example 2

To avoid fully qualifying classes throughout a program, you can use the `using` directive as shown below:

```
// Hello2.cs
using System;

public class Hello2
{
   public static void Main()
   {
      Console.WriteLine("Hello, World!");
   }
}
```

Output

```
Hello, World!
```

Example 3

If you need access to the command line parameters passed in to your application, simply change the signature of the `Main` method to include them as shown below. This example counts and displays the command line arguments.

```
// Hello3.cs
// arguments: A B C D
using System;

public class Hello3
{
   public static void Main(string[] args)
   {
      Console.WriteLine("Hello, World!");
      Console.WriteLine("You entered the following {0} command line arguments:",
         args.Length );
      for (int i=0; i < args.Length; i++)
      {
         Console.WriteLine("{0}", args[i]);
      }
   }
}
```

Output

```
Hello, World!
You entered the following 4 command line arguments:
A
B
C
D
```

Example 4

To return a return code, change the signature of the `Main` method as shown below:

```
// Hello4.cs
using System;

public class Hello4
{
   public static int Main(string[] args)
   {
      Console.WriteLine("Hello, World!");
      return 0;
   }
}
```

Output

```
Hello, World!
```

See Also

C# Tutorials

Command Line Parameters Tutorial

This tutorial shows how the command line can be accessed and two ways of accessing the array of command line parameters.

Sample Files

See Command Line Parameters Sample to download and build the sample files discussed in this tutorial.

Further Reading

- The C# Version of Hello World!
- foreach, in
- String.Length Property

Tutorial

The following examples show two different approaches to using the command line arguments passed to an application.

Example 1

This example demonstrates how to print out the command line arguments.

```
// cmdline1.cs
// arguments: A B C
using System;

public class CommandLine
{
   public static void Main(string[] args)
   {
       // The Length property is used to obtain the length of the array.
       // Notice that Length is a read-only property:
       Console.WriteLine("Number of command line parameters = {0}",
          args.Length);
       for(int i = 0; i < args.Length; i++)
       {
           Console.WriteLine("Arg[{0}] = [{1}]", i, args[i]);
       }
   }
}
```

Output

Run the program using some arguments like this: `cmdline1 A B C`.

The output will be:

```
Number of command line parameters = 3
Arg[0] = [A]
Arg[1] = [B]
Arg[2] = [C]
```

Example 2

Another approach to iterating over the array is to use the **foreach** statement as shown in this example. The **foreach** statement can be used to iterate over an array or over a .NET Framework collection class. It provides a simple way to iterate over collections.

```
// cmdline2.cs
// arguments: John Paul Mary
using System;

public class CommandLine2
```

```
{
   public static void Main(string[] args)
   {
       Console.WriteLine("Number of command line parameters = {0}",
          args.Length);
       foreach(string s in args)
       {
          Console.WriteLine(s);
       }
   }
}
```

Output

Run the program using some arguments like this: `cmdline2 John Paul Mary`.

The output will be:

```
Number of command line parameters = 3
John
Paul
Mary
```

See Also

C# Tutorials | Collection Classes Tutorial

Arrays Tutorial

This tutorial describes arrays and shows how they work in C#.

Sample Files

See Arrays Sample to download and build the sample files discussed in this tutorial.

Further Reading

- Arrays
- 12. Arrays
- foreach, in
- Collection Classes Tutorial

Tutorial

This tutorial is divided into the following sections:

- Arrays in General
- Declaring Arrays
- Initializing Arrays
- Accessing Array Members
- Arrays are Objects
- Using foreach with Arrays

Arrays in General

C# arrays are zero indexed; that is, the array indexes start at zero. Arrays in C# work similarly to how arrays work in most other popular languages There are, however, a few differences that you should be aware of.

When declaring an array, the square brackets ([]) must come after the type, not the identifier. Placing the brackets after the identifier is not legal syntax in C#.

```
int[] table; // not int table[];
```

Another detail is that the size of the array is not part of its type as it is in the C language. This allows you to declare an array and assign any array of **int** objects to it, regardless of the array's length.

```
int[] numbers; // declare numbers as an int array of any size
numbers = new int[10];  // numbers is a 10-element array
numbers = new int[20];  // now it's a 20-element array
```

Declaring Arrays

C# supports single-dimensional arrays, multidimensional arrays (rectangular arrays), and array-of-arrays (jagged arrays). The following examples show how to declare different kinds of arrays:

Single-dimensional arrays:

```
int[] numbers;
```

Multidimensional arrays:

```
string[,] names;
```

Array-of-arrays (jagged):

```
byte[][] scores;
```

Declaring them (as shown above) does not actually create the arrays. In C#, arrays are objects (discussed later in this tutorial) and must be instantiated. The following examples show how to create arrays:

Single-dimensional arrays:

```
int[] numbers = new int[5];
```

Multidimensional arrays:

```
string[,] names = new string[5,4];
```

Array-of-arrays (jagged):

```
byte[][] scores = new byte[5][];
for (int x = 0; x < scores.Length; x++)
{
   scores[x] = new byte[4];
}
```

You can also have larger arrays. For example, you can have a three-dimensional rectangular array:

```
int[,,] buttons = new int[4,5,3];
```

You can even mix rectangular and jagged arrays. For example, the following code declares a single-dimensional array of three-dimensional arrays of two-dimensional arrays of type **int**:

```
int[][,,][,] numbers;
```

Example

The following is a complete C# program that declares and instantiates arrays as discussed above.

```
// arrays.cs
using System;
class DeclareArraysSample
{
    public static void Main()
    {
        // Single-dimensional array
        int[] numbers = new int[5];

        // Multidimensional array
        string[,] names = new string[5,4];

        // Array-of-arrays (jagged array)
        byte[][] scores = new byte[5][];

        // Create the jagged array
        for (int i = 0; i < scores.Length; i++)
        {
            scores[i] = new byte[i+3];
        }

        // Print length of each row
        for (int i = 0; i < scores.Length; i++)
        {
            Console.WriteLine("Length of row {0} is {1}", i, scores[i].Length);
        }
    }
}
```

Output

```
Length of row 0 is 3
Length of row 1 is 4
Length of row 2 is 5
Length of row 3 is 6
Length of row 4 is 7
```

Initializing Arrays

C# provides simple and straightforward ways to initialize arrays at declaration time by enclosing the initial values in curly braces ({}). The following examples show different ways to initialize different kinds of arrays.

> **Note** If you do not initialize an array at the time of declaration, the array members are automatically initialized to the default initial value for the array type. Also, if you declare the array as a field of a type, it will be set to the default value null when you instantiate the type.

Single-Dimensional Array

```
int[] numbers = new int[5] {1, 2, 3, 4, 5};
string[] names = new string[3] {"Matt", "Joanne", "Robert"};
```

You can omit the size of the array, like this:

```
int[] numbers = new int[] {1, 2, 3, 4, 5};
string[] names = new string[] {"Matt", "Joanne", "Robert"};
```

You can also omit the **new** operator if an initializer is provided, like this:

```
int[] numbers = {1, 2, 3, 4, 5};
string[] names = {"Matt", "Joanne", "Robert"};
```

Multidimensional Array

```
int[,] numbers = new int[3, 2] { {1, 2}, {3, 4}, {5, 6} };
string[,] siblings = new string[2, 2] { {"Mike","Amy"}, {"Mary","Albert"} };
```

You can omit the size of the array, like this:

```
int[,] numbers = new int[,] { {1, 2}, {3, 4}, {5, 6} };
string[,] siblings = new string[,] { {"Mike","Amy"}, {"Mary","Albert"} };
```

You can also omit the **new** operator if an initializer is provided, like this:

```
int[,] numbers = { {1, 2}, {3, 4}, {5, 6} };
string[,] siblings = { {"Mike", "Amy"}, {"Mary", "Albert"} };
```

Jagged Array (Array-of-Arrays)

You can initialize jagged arrays like this example:

```
int[][] numbers = new int[2][] { new int[] {2,3,4}, new int[] {5,6,7,8,9} };
```

You can also omit the size of the first array, like this:

```
int[][] numbers = new int[][] { new int[] {2,3,4}, new int[] {5,6,7,8,9} };
```

-or-

```
int[][] numbers = { new int[] {2,3,4}, new int[] {5,6,7,8,9} };
```

Notice that there is no initialization syntax for the elements of a jagged array.

Accessing Array Members

Accessing array members is straightforward and similar to how you access array members in C/C++. For example, the following code creates an array called `numbers` and then assigns a `5` to the fifth element of the array:

```
int[] numbers = {10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0};
numbers[4] = 5;
```

The following code declares a multidimensional array and assigns `5` to the member located at `[1, 1]`:

```
int[,] numbers = { {1, 2}, {3, 4}, {5, 6}, {7, 8}, {9, 10} };
numbers[1, 1] = 5;
```

The following is a declaration of a single-dimension jagged array that contains two elements. The first element is an array of two integers, and the second is an array of three integers:

```
int[][] numbers = new int[][] { new int[] {1, 2}, new int[] {3, 4, 5}
};
```

The following statements assign 58 to the first element of the first array and 667 to the second element of the second array:

```
numbers[0][0] = 58;
numbers[1][1] = 667;
```

Arrays are Objects

In C#, arrays are actually objects. **System.Array** is the abstract base type of all array types. You can use the properties, and other class members, that **System.Array** has. An example of this would be using the Length property to get the length of an array. The following code assigns the length of the `numbers` array, which is `5`, to a variable called `LengthOfNumbers`:

```
int[] numbers = {1, 2, 3, 4, 5};
int LengthOfNumbers = numbers.Length;
```

The **System.Array** class provides many other useful methods/properties, such as methods for sorting, searching, and copying arrays.

Using foreach on Arrays

C# also provides the **foreach** statement. This statement provides a simple, clean way to iterate through the elements of an array. For example, the following code creates an array called `numbers` and iterates through it with the **foreach** statement:

```
int[] numbers = {4, 5, 6, 1, 2, 3, -2, -1, 0};
foreach (int i in numbers)
{
   System.Console.WriteLine(i);
}
```

With multidimensional arrays, you can use the same method to iterate through the elements, for example:

```
int[,] numbers = new int[3, 2] {{9, 99}, {3, 33}, {5, 55}};
foreach(int i in numbers)
{
```

```
    Console.Write("{0} ", i);
}
```

The output of this example is:

```
9 99 3 33 5 55
```

However, with multidimensional arrays, using a nested **for** loop gives you more control over the array elements.

See Also

C# Tutorials

Properties Tutorial

This tutorial shows how properties are an integral part of the C# programming language. It demonstrates how properties are declared and used.

Sample Files

See Properties Sample to download and build the sample files discussed in this tutorial.

Further Reading

- Properties
- Comparison Between Properties and Indexers
- 10.6 Properties

Tutorial

This tutorial includes two examples. The first example shows how to declare and use read/write properties. The second example demonstrates abstract properties and shows how to override these properties in subclasses.

Example 1

This sample shows a `Person` class that has two properties: `Name` (string) and `Age` (int). Both properties are read/write.

```
// person.cs
using System;
class Person
{
    private string myName ="N/A";
    private int myAge = 0;

    // Declare a Name property of type string:
    public string Name
    {
        get
        {
           return myName;
        }
        set
        {
           myName = value;
        }
    }

    // Declare an Age property of type int:
    public int Age
    {
        get
        {
           return myAge;
        }
        set
        {
           myAge = value;
        }
    }

    public override string ToString()
    {
        return "Name = " + Name + ", Age = " + Age;
    }

    public static void Main()
    {
        Console.WriteLine("Simple Properties");
```

```
        // Create a new Person object:
        Person person = new Person();

        // Print out the name and the age associated with the person:
        Console.WriteLine("Person details - {0}", person);

        // Set some values on the person object:
        person.Name = "Joe";
        person.Age = 99;
        Console.WriteLine("Person details - {0}", person);

        // Increment the Age property:
        person.Age += 1;
        Console.WriteLine("Person details - {0}", person);
    }
}
```

Output

```
Simple Properties
Person details - Name = N/A, Age = 0
Person details - Name = Joe, Age = 99
Person details - Name = Joe, Age = 100
```

Code Discussion

- Notice the way that the properties are declared, for example, consider the `Name` property:

  ```
  public string Name
  {
      get
      {
         return myName;
      }
      set
      {
         myName = value;
      }
  }
  ```

  The Set and Get methods of a property are contained inside the property declaration. You can control whether a property is read/write, read-only, or write-only by controlling whether a Get or Set method is included.

- Once the properties are declared, they can be used as if they were fields of the class. This allows for a very natural syntax when both getting and setting the value of a property, as in the following statements:

  ```
  person.Name = "Joe";
  person.Age = 99;
  ```

- Note that in a property Set method a special `value` variable is available. This variable contains the value that the user specified, for example:

  ```
  myName = value;
  ```

- Notice the clean syntax for incrementing the `Age` property on a `Person` object:

  ```
  person.Age += 1;
  ```

  If separate Set and Get methods were used to model properties, the equivalent code might look like this:

```
person.SetAge(person.GetAge() + 1);
```

- The ToString method is overridden in this example:

```
public override string ToString()
{
    return "Name = " + Name + ", Age = " + Age;
}
```

Notice that ToString is not explicitly used in the program. It is invoked by default by the WriteLine calls.

Example 2

The following example shows how to define abstract properties. An abstract property declaration does not provide an implementation of the property accessors. The example demonstrates how to override these properties in subclasses.

This sample consists of three files. In the Properties Sample, these files are compiled into a single compilation but in this tutorial, each file is compiled individually and its resulting assembly referenced by the next compilation:

- `abstractshape.cs`: The `Shape` class that contains an abstract `Area` property.
- `shapes.cs`: The subclasses of the `Shape` class.
- `shapetest.cs`: A test program to display the areas of some `Shape`-derived objects.

To compile the example, use the command line:

```
csc abstractshape.cs shapes.cs shapetest.cs
```

This will create the executable file shapetest.exe.

File 1 - abstractshape.cs

This file declares the `Shape` class that contains the `Area` property of the type double

```
// abstractshape.cs
// compile with: /target:library
// csc /target:library abstractshape.cs
using System;

public abstract class Shape
{
   private string myId;

   public Shape(string s)
   {
      Id = s;   // calling the set accessor of the Id property
   }

   public string Id
   {
      get
      {
         return myId;
      }

      set
      {
         myId = value;
      }
   }

   // Area is a read-only property - only a get accessor is needed:
   public abstract double Area
   {
```

```
      get;
   }

   public override string ToString()
   {
      return Id + " Area = " + string.Format("{0:F2}",Area);
   }
}
```

Code Discussion

- Modifiers on the property are placed on the property declaration itself, for example:

  ```
  public abstract double Area
  ```

- When declaring an abstract property (such as `Area` in this example), you simply indicate what property accessors are available, but do not implement them. In this example, only a Get accessor is available, so the property is read-only.

File 2 - shapes.cs

The following code shows three subclasses of `Shape` and how they override the `Area` property to provide their own implementation.

```
// shapes.cs
// compile with: /target:library /reference:abstractshape.dll
public class Square : Shape
{
   private int mySide;

   public Square(int side, string id) : base(id)
   {
      mySide = side;
   }

   public override double Area
   {
      get
      {
         // Given the side, return the area of a square:
         return mySide * mySide;
      }
   }
}

public class Circle : Shape
{
   private int myRadius;

   public Circle(int radius, string id) : base(id)
   {
      myRadius = radius;
   }

   public override double Area
   {
      get
      {
         // Given the radius, return the area of a circle:
         return myRadius * myRadius * System.Math.PI;
      }
   }
}

public class Rectangle : Shape
{
   private int myWidth;
```

```
   private int myHeight;

   public Rectangle(int width, int height, string id) : base(id)
   {
      myWidth  = width;
      myHeight = height;
   }

   public override double Area
   {
      get
      {
         // Given the width and height, return the area of a rectangle:
         return myWidth * myHeight;
      }
   }
}
```

File 3 - shapetest.cs

The following code shows a test program that creates a number of `Shape`-derived objects and prints out their areas.

```
// shapetest.cs
// compile with: /reference:abstractshape.dll;shapes.dll
public class TestClass
{
   public static void Main()
   {
      Shape[] shapes =
         {
            new Square(5, "Square #1"),
            new Circle(3, "Circle #1"),
            new Rectangle( 4, 5, "Rectangle #1")
         };

      System.Console.WriteLine("Shapes Collection");
      foreach(Shape s in shapes)
      {
         System.Console.WriteLine(s);
      }

   }
}
```

Output

```
Shapes Collection
Square #1 Area = 25.00
Circle #1 Area = 28.27
Rectangle #1 Area = 20.00
```

See Also

C# Tutorials

Libraries Tutorial

This tutorial shows how to create and use libraries in C#.

Sample Files

See Libraries Sample to download and build the sample files discussed in this tutorial.

Further Reading

- Creating and Using C# DLLs
- C# Compiler Options

Tutorial

This tutorial demonstrates how to create a managed DLL file by using the necessary compiler options, and how to use the library by a client program.

Example

This example uses the following modules:

- The DLL library (`Functions.dll`), which is built from the following source files:

  `Factorial.cs`: Calculates and returns the factorial of a number.

  `DigitCounter.cs`: Counts the number of digits in the passed string.

- The client program (`FunctionTest.exe`), which uses the DLL, is compiled from the source file `FunctionClient.cs`. The program displays the factorial of the input arguments.

Building the Library

To build the library, make Functions the current directory and type the following at the command prompt:

```
csc /target:library /out:Functions.dll Factorial.cs DigitCounter.cs
```

where:

`/target:library`	Specifies that the output is a DLL and not an executable file (this also stops the compiler fro m looking for a default entry point).
`/out:Functions.dll`	Specifies that the output file name is `Functions.dll`. Normally the output name is the same name as the first C# source file on the command line (in this example, `Factorial`).
`Factorial.cs DigitCounter.cs`	Specifies the files to compile and place in the DLL.

Compiling the Client

To compile the program, make FunctionTest the current directory and type the following at the command prompt:

```
copy ..\Functions\Functions.dll .
csc /out:FunctionTest.exe /R:Functions.DLL FunctionClient.cs
```

where:

`/out:FunctionTest.exe`	Specifies that the output file name is `FunctionTest.exe`.
`/R:Functions.DLL`	Specifies that `Functions.DLL` must be included when resolving references. This DLL must be located in the current directory or have a fully qualified path.
`FunctionClient.cs`	Specifies the client source code.

This creates the executable file `FunctionTest.exe`.

File 1 - Factorial.cs

The following code calculates the factorial of the integer passed to the method (unlike Libraries Sample, compile this to a library).

```
// Factorial.cs
// compile with: /target:library
using System;

// Declares a namespace. You need to package your libraries according
// to their namespace so the .NET runtime can correctly load the classes.
namespace Functions
{
    public class Factorial
    {
// The "Calc" static method calculates the factorial value for the
// specified integer passed in:
        public static int Calc(int i)
        {
            return((i <= 1) ? 1 : (i * Calc(i-1)));
        }
    }
}
```

File 2 - DigitCounter.cs

The following code is used to count the number of digit characters in the passed string:

```
// DigitCounter.cs
// compile with: /target:library /out:Functions.dll /reference:Factorial.dll
using System;

// Declare the same namespace as the one in Factorial.cs. This simply
// allows types to be added to the same namespace.
namespace Functions
{
    public class DigitCount
    {
        // The NumberOfDigits static method calculates the number of
        // digit characters in the passed string:
        public static int NumberOfDigits(string theString)
        {
            int count = 0;
            for ( int i = 0; i < theString.Length; i++ )
            {
                if ( Char.IsDigit(theString[i]) )
                {
                    count++;
                }
            }
            return count;
        }
    }
}
```

File 3 - FunctionClient.cs

Once you build a library, it can be used by other programs. The following client program uses the classes defined in the library. The basic function of the program is to take each command-line parameter and attempt to compute the factorial value for each argument.

```
// FunctionClient.cs
// compile with: /reference:Functions.dll,Factorial.dll /out:FunctionTest.exe
// arguments: 3 5 10
using System;
// The following using directive makes the types defined in the Functions
// namespace available in this compilation unit:
```

```
using Functions;
class FunctionClient
{
    public static void Main(string[] args)
    {
        Console.WriteLine("Function Client");

        if ( args.Length == 0 )
        {
            Console.WriteLine("Usage: FunctionTest ... ");
            return;
        }

        for ( int i = 0; i < args.Length; i++ )
        {
            int num = Int32.Parse(args[i]);
            Console.WriteLine(
               "The Digit Count for String [{0}] is [{1}]",
               args[i],
               // Invoke the NumberOfDigits static method in the
               // DigitCount class:
               DigitCount.NumberOfDigits(args[i]));
            Console.WriteLine(
               "The Factorial for [{0}] is [{1}]",
                num,
               // Invoke the Calc static method in the Factorial class:
                Factorial.Calc(num) );
        }
    }
}
```

Output

The command line `FunctionTest 3 5 10` uses the program `FunctionTest` to calculate the factorial of the three integers 3, 5, and 10. It also displays the number of digits for each argument.

This run gives the output:

```
Function Client
The Digit Count for String [3] is [1]
The Factorial for [3] is [6]
The Digit Count for String [5] is [1]
The Factorial for [5] is [120]
The Digit Count for String [10] is [2]
The Factorial for [10] is [3628800]
```

> **Note** To run the client executable (`FunctionTest.exe`), the file `Functions.DLL` must be in the current directory, a child directory, or in the Global Assembly Cache. For more information see Global Assembly Cache.

See Also

C# Tutorials

Versioning Tutorial

This tutorial demonstrates versioning in C# through the use of the **override** and **new** keywords. Versioning maintains compatibility between base and derived classes as they evolve.

Sample Files

See Versioning Sample to download and build the sample files discussed in this tutorial.

Further Reading

- virtual
- override
- new
- 10.5.3 Virtual methods

Tutorial

The C# language is designed such that versioning between base and derived classes in different libraries can evolve and maintain backwards compatibility. This means, for example, that the introduction of a new member in a base class with the same name as a member in a derived class is not an error. It also means that a class must explicitly state whether a method is intended to override an inherited method, or whether a method is a new method that simply hides a similarly named inherited method.

In C#, methods are by default, not virtual. To make a method virtual, the **virtual** modifier has to be used in the method declaration of the base class. The derived class can then override the base virtual method by using the **override** keyword or hide the virtual method in the base class by using the **new** keyword. If neither the **override** keyword nor the **new** keyword is specified, the compiler will issue a warning and the method in the derived class will hide the method in the base class. The following example shows these concepts in action.

Example

```
// versioning.cs
// CS0114 expected
public class MyBase
{
   public virtual string Meth1()
   {
      return "MyBase-Meth1";
   }
   public virtual string Meth2()
   {
      return "MyBase-Meth2";
   }
   public virtual string Meth3()
   {
      return "MyBase-Meth3";
   }
}

class MyDerived : MyBase
{
   // Overrides the virtual method Meth1 using the override keyword:
   public override string Meth1()
   {
      return "MyDerived-Meth1";
   }
   // Explicitly hide the virtual method Meth2 using the new
   // keyword:
   public new string Meth2()
   {
      return "MyDerived-Meth2";
   }
   // Because no keyword is specified in the following declaration
   // a warning will be issued to alert the programmer that
```

```
    // the method hides the inherited member MyBase.Meth3():
    public string Meth3()
    {
       return "MyDerived-Meth3";
    }

    public static void Main()
    {
       MyDerived mD = new MyDerived();
       MyBase mB = (MyBase) mD;

       System.Console.WriteLine(mB.Meth1());
       System.Console.WriteLine(mB.Meth2());
       System.Console.WriteLine(mB.Meth3());
    }
 }
```

Output

```
MyDerived-Meth1
MyBase-Meth2
MyBase-Meth3
```

Code Discussion

Hiding a base class member from a derived class isn't an error in C#. This feature enables you to make changes in the base class without breaking other libraries that inherit this base class. For example, at some point you could have the following classes:

```
class Base {}
class Derived: Base
{
   public void F() {}
}
```

At some later point, the base class could evolve to add a void method `F()` as follows:

```
class Base
{
   public void F() {}
}
class Derived: Base
{
   public void F() {}
}
```

Thus, in C#, both the base and derived classes can evolve freely and maintain binary compatibility.

See Also

C# Tutorials

Collection Classes Tutorial

This tutorial shows how to implement a collection class that can be used with the **foreach** statement.

Sample Files

See Collection Classes Sample to download and build the sample files discussed in this tutorial.

Further Reading

- foreach, in
- 8.8.4 The foreach statement
- Command Line Parameters Tutorial
- Arrays Tutorial

Tutorial

The **foreach** statement is a convenient way to iterate over the elements of an array. It can also enumerate the elements of a collection, provided that the collection class has implemented the **System.Collections.IEnumerator** and **System.Collections.IEnumerable** interfaces.

Example 1

The following code sample illustrates how to write a collection class that can be used with **foreach**. The class is a string tokenizer, similar to the C run-time function **strtok**.

```
// tokens.cs
using System;
// The System.Collections namespace is made available:
using System.Collections;

// Declare the Tokens class:
public class Tokens : IEnumerable
{
   private string[] elements;

   Tokens(string source, char[] delimiters)
   {
      // Parse the string into tokens:
      elements = source.Split(delimiters);
   }

   // IEnumerable Interface Implementation:
   //   Declaration of the GetEnumerator() method
   //   required by IEnumerable
   public IEnumerator GetEnumerator()
   {
      return new TokenEnumerator(this);
   }

   // Inner class implements IEnumerator interface:
   private class TokenEnumerator : IEnumerator
   {
      private int position = -1;
      private Tokens t;

      public TokenEnumerator(Tokens t)
      {
         this.t = t;
      }

      // Declare the MoveNext method required by IEnumerator:
      public bool MoveNext()
      {
         if (position < t.elements.Length - 1)
```

```
         {
            position++;
            return true;
         }
         else
         {
            return false;
         }
      }

      // Declare the Reset method required by IEnumerator:
      public void Reset()
      {
         position = -1;
      }

      // Declare the Current property required by IEnumerator:
      public object Current
      {
         get
         {
            return t.elements[position];
         }
      }
   }

   // Test Tokens, TokenEnumerator

   static void Main()
   {
      // Testing Tokens by breaking the string into tokens:
      Tokens f = new Tokens("This is a well-done program.",
         new char[] {' ','-'});
      foreach (string item in f)
      {
         Console.WriteLine(item);
      }
   }
}
```

Output

```
This
is
a
well
done
program.
```

Code Discussion

In the preceding example, the following code is used to `Tokens` by breaking "This is a well-done program." into tokens (using ' ' and '-' as separators) and enumerating those tokens with the **foreach** statement:

```
Tokens f = new Tokens("This is a well-done program.",
   new char[] {' ','-'});
foreach (string item in f)
{
   Console.WriteLine(item);
}
```

Notice that, internally, `Tokens` uses an array, which implements `IEnumerator` and `IEnumerable` itself. The code sample could have leveraged the array's enumeration methods as its own, but that would have defeated the purpose of this example.

In C#, it is not strictly necessary for a collection class to inherit from **IEnumerable** and **IEnumerator** in order to be compatible with **foreach**; as long as the class has the required `GetEnumerator`, `MoveNext`, `Reset`, and `Current` members, it will work with **foreach**. Omitting the interfaces has the advantage of allowing you to define the return type of `Current` to be more specific than **object**, thereby providing type-safety.

For example, starting with the sample code above, change the following lines:

```
public class Tokens  // no longer inherits from IEnumerable
public TokenEnumerator GetEnumerator()  // doesn't return an IEnumerator
public class TokenEnumerator  // no longer inherits from IEnumerator
public string Current  // type-safe: returns string, not object
```

Now, because `Current` returns a string, the compiler can detect when an incompatible type is used in a **foreach** statement:

```
foreach (int item in f)  // Error: cannot convert string to int
```

The disadvantage of omitting **IEnumerable** and **IEnumerator** is that the collection class is no longer interoperable with the **foreach** statements (or equivalents) of other common language runtime-compatible languages.

You can have the best of both worlds — type-safety within C# and interoperability with other common language runtime-compatible languages — by inheriting from **IEnumerable** and **IEnumerator** and using explicit interface implementation, as demonstrated in the following example.

Example 2

This sample is equivalent in function to Example 1, but it provides additional type-safety in C# while maintaining interoperability with other languages.

```
// tokens2.cs
using System;
using System.Collections;

public class Tokens: IEnumerable
{
   private string[] elements;

   Tokens(string source, char[] delimiters)
   {
      elements = source.Split(delimiters);
   }

   // IEnumerable Interface Implementation:

   public TokenEnumerator GetEnumerator() // non-IEnumerable version
   {
      return new TokenEnumerator(this);
   }

   IEnumerator IEnumerable.GetEnumerator() // IEnumerable version
   {
      return (IEnumerator) new TokenEnumerator(this);
   }

   // Inner class implements IEnumerator interface:

   public class TokenEnumerator: IEnumerator
   {
      private int position = -1;
      private Tokens t;

      public TokenEnumerator(Tokens t)
      {
         this.t = t;
      }
```

```
        public bool MoveNext()
        {
           if (position < t.elements.Length - 1)
           {
              position++;
              return true;
           }
           else
           {
              return false;
           }
        }

        public void Reset()
        {
           position = -1;
        }

        public string Current // non-IEnumerator version: type-safe
        {
           get
           {
              return t.elements[position];
           }
        }

        object IEnumerator.Current // IEnumerator version: returns object
        {
           get
           {
              return t.elements[position];
           }
        }
     }

     // Test Tokens, TokenEnumerator

     static void Main()
     {
        Tokens f = new Tokens("This is a well-done program.",
           new char [] {' ','-'});
        foreach (string item in f) // try changing string to int
        {
           Console.WriteLine(item);
        }
     }
  }
```

See Also

C# Tutorials

Structs Tutorial

This tutorial presents the syntax and usage of structs. It also covers the important differences between classes and structs.

Sample Files

See Structs Sample to download and build the sample files discussed in this tutorial.

Further Reading

- struct
- 1.8 Structs
- Structs Grammar in C. Grammar
- Value Types
- C# Attributes
- 17. Attributes
- StructLayoutAttribute Class

Tutorial

This tutorial includes two examples. The first example shows you how to declare and use structs, and the second example demonstrates the difference between structs and classes when instances are passed to methods. You are also introduced to the following topics:

- Structs vs. Classes
- Heap or Stack?
- Constructors and Inheritance
- Attributes on Structs

Example 1

This example declares a struct with three members: a property, a method, and a private field. It creates an instance of the struct and puts it to use:

```
// struct1.cs
using System;
struct SimpleStruct
{
    private int xval;
    public int X
    {
        get
        {
            return xval;
        }
        set
        {
            if (value < 100)
                xval = value;
        }
    }
    public void DisplayX()
    {
        Console.WriteLine("The stored value is: {0}", xval);
    }
}

class TestClass
{
    public static void Main()
    {
        SimpleStruct ss = new SimpleStruct();
        ss.X = 5;
```

```
        ss.DisplayX();
    }
}
```

Output

```
The stored value is: 5
```

Structs vs. Classes

Structs may seem similar to classes, but there are important differences that you should be aware of. First of all, classes are reference types and structs are value types. By using structs, you can create objects that behave like the built-in types and enjoy their benefits as well.

Heap or Stack?

When you call the New operator on a class, it will be allocated on the heap. However, when you instantiate a struct, it gets created on the stack. This will yield performance gains. Also, you will not be dealing with references to an instance of a struct as you would with classes. You will be working directly with the struct instance. Because of this, when passing a struct to a method, it's passed by value instead of as a reference.

Example 2

This example shows that when a struct is passed to a method, a copy of the struct is passed, but when a class instance is passed, a reference is passed.

```
// struct2.cs
using System;

class TheClass
{
    public int x;
}

struct TheStruct
{
    public int x;
}

class TestClass
{
    public static void structtaker(TheStruct s)
    {
        s.x = 5;
    }
    public static void classtaker(TheClass c)
    {
        c.x = 5;
    }
    public static void Main()
    {
        TheStruct a = new TheStruct();
        TheClass b = new TheClass();
        a.x = 1;
        b.x = 1;
        structtaker(a);
        classtaker(b);
        Console.WriteLine("a.x = {0}", a.x);
        Console.WriteLine("b.x = {0}", b.x);
    }
}
```

Output

```
a.x = 1
b.x = 5
```

Code Discussion

The output of the example shows that only the value of the class field was changed when the class instance was passed to the `classtaker` method. The struct field, however, did not change by passing its instance to the `structtaker` method. This is because a copy of the struct was passed to the `structtaker` method, while a reference to the class was passed to the `classtaker` method.

Constructors and Inheritance

Structs can declare constructors, but they must take parameters. It is an error to declare a default (parameterless) constructor for a struct. Struct members cannot have initializers. A default constructor is always provided to initialize the struct members to their default values.

When you create a struct object using the New operator, it gets created and the appropriate constructor is called. Unlike classes, structs can be instantiated without using the New operator. If you do not use New, the fields will remain unassigned and the object cannot be used until all the fields are initialized.

There is no inheritance for structs as there is for classes. A struct cannot inherit from another struct or class, and it cannot be the base of a class. Structs, however, inherit from the base class object. A struct can implement interfaces, and it does that exactly as classes do. Here's a code snippet of a struct implementing an interface:

```
interface IImage
{
    void Paint();
}

struct Picture : IImage
{
    public void Paint()
    {
         // painting code goes here
    }
    private int x, y, z;  // other struct members
}
```

Attributes on Structs

By using attributes you can customize how structs are laid out in memory. For example, you can create what's known as a union in C/C++ by using the **StructLayout(LayoutKind.Explicit)** and **FieldOffset** attributes.

```
using System.Runtime.InteropServices;
[StructLayout(LayoutKind.Explicit)]
struct TestUnion
{
    [FieldOffset(0)]
    public int i;
    [FieldOffset(0)]
    public double d;
    [FieldOffset(0)]
    public char c;
    [FieldOffset(0)]
    public byte b1;
}
```

In the preceding code segment, all of the fields of `TestUnion` start at the same location in memory.

The following is another example where fields start at different explicitly set locations:

```
using System.Runtime.InteropServices;
```

```
[StructLayout(LayoutKind.Explicit)]
struct TestExplicit
{
    [FieldOffset(0)]
    public long lg;
    [FieldOffset(0)]
    public int i1;
    [FieldOffset(4)]
    public int i2;
    [FieldOffset(8)]
    public double d;
    [FieldOffset(12)]
    public char c;
    [FieldOffset(14)]
    public byte b1;
}
```

The two **int** fields, `i1` and `i2`, share the same memory locations as `lg`. This sort of control over struct layout is useful when using platform invocation.

Conclusion

Structs are simple to use and can prove to be useful at times. Just keep in mind that they're created on the stack and that you're not dealing with references to them but dealing directly with them. Whenever you have a need for a type that will be used often and is mostly just a piece of data, structs might be a good option.

See Also

C# Tutorials

Indexers Tutorial

This tutorial shows how C# classes can declare indexers to provide array-like access to the classes.

Sample Files

See Indexers Sample to download and build the sample files discussed in this tutorial.

Further Reading

- Indexers
- Accessors
- Comparison Between Properties and Indexers
- 10.8 Indexers
- 10.8.1 Indexer overloading

Tutorial

Defining an *indexer* allows you to create classes that act like "virtual arrays." Instances of that class can be accessed using the **[]** array access operator. Defining an indexer in C# is similar to defining operator **[]** in C++, but is considerably more flexible. For classes that encapsulate array- or collection-like functionality, using an indexer allows the users of that class to use the array syntax to access the class.

For example, suppose you want to define a class that makes a file appear as an array of bytes. If the file were very large, it would be impractical to read the entire file into memory, especially if you only wanted to read or change a few bytes. By defining a `FileByteArray` class, you could make the file appear similar to an array of bytes, but actually do file input and output when a byte was read or written.

In addition to the example below, an advanced topic on Creating an Indexed Property is discussed in this tutorial.

Example

In this example, the class `FileByteArray` makes it possible to access a file as if it were a byte array. The `Reverse` class reverses the bytes of the file. You can run this program to reverse the bytes of any text file including the program source file itself. To change the reversed file back to normal, run the program on the same file again.

```
// indexer.cs
// arguments: indexer.txt
using System;
using System.IO;

// Class to provide access to a large file
// as if it were a byte array.
public class FileByteArray
{
    Stream stream;      // Holds the underlying stream
                        // used to access the file.
// Create a new FileByteArray encapsulating a particular file.
    public FileByteArray(string fileName)
    {
        stream = new FileStream(fileName, FileMode.Open);
    }

    // Close the stream. This should be the last thing done
    // when you are finished.
    public void Close()
    {
        stream.Close();
        stream = null;
    }

    // Indexer to provide read/write access to the file.
    public byte this[long index]   // long is a 64-bit integer
    {
```

```
        // Read one byte at offset index and return it.
        get
        {
            byte[] buffer = new byte[1];
            stream.Seek(index, SeekOrigin.Begin);
            stream.Read(buffer, 0, 1);
            return buffer[0];
        }
        // Write one byte at offset index and return it.
        set
        {
            byte[] buffer = new byte[1] {value};
            stream.Seek(index, SeekOrigin.Begin);
            stream.Write(buffer, 0, 1);
        }
    }

    // Get the total length of the file.
    public long Length
    {
        get
        {
            return stream.Seek(0, SeekOrigin.End);
        }
    }
}

// Demonstrate the FileByteArray class.
// Reverses the bytes in a file.
public class Reverse
{
    public static void Main(String[] args)
    {
        // Check for arguments.
        if (args.Length == 0)
        {
            Console.WriteLine("indexer <filename>");
            return;
        }

        FileByteArray file = new FileByteArray(args[0]);
        long len = file.Length;

        // Swap bytes in the file to reverse it.
        for (long i = 0; i < len / 2; ++i)
        {
            byte t;

            // Note that indexing the "file" variable invokes the
            // indexer on the FileByteStream class, which reads
            // and writes the bytes in the file.
            t = file[i];
            file[i] = file[len - i - 1];
            file[len - i - 1] = t;
        }

        file.Close();
    }
}
```

Input: indexer.txt

To test the program you can use a text file with the following contents (this file is called `Test.txt` in the Indexers Sample).

```
public class Hello1
{
   public static void Main()
```

```
    {
       System.Console.WriteLine("Hello, World!");
    }
}
```

To reverse the bytes of this file, compile the program and then use the command line:

```
indexer indexer.txt
```

To display the reversed file, enter the command:

```
Type indexer.txt
```

Sample Output

```
}
}
;)"!dlroW ,olleH"(eniLetirW.elosnoC.metsyS
{
)(niaM diov citats cilbup
{
1olleH ssalc cilbup
```

Code Discussion

- Since an indexer is accessed using the **[]** operator, it does not have a name. For indexer declaration syntax, see Indexers.
- In the example above, the indexer is of type byte and takes a single index of type long (64-bit integer). The Get accessor defines the code to read a byte from the file, while the Set accessor defines the code to write a byte to the file. Inside the Set accessor, the predefined parameter value has the value that is being assigned to the virtual array element.
- An indexer must have at least one parameter. Although it is comparatively rare, an indexer can have more than one parameter in order to simulate a multidimensional "virtual array." Although integral parameters are the most common, the indexer parameter can be of any type. For example, the standard Dictionary class provides an indexer with a parameter of type Object.
- Although indexers are a powerful feature, it is important to use them only when the array-like abstraction makes sense. Always carefully consider whether using regular method(s) would be just as clear. For example, the following is a bad use of an indexer:

  ```
  class Employee
  {
      // VERY BAD STYLE: using an indexer to access
      // the salary of an employee.
      public double this[int year]
     {
          get
          {
              // return employee's salary for a given year.
          }
     }
  }
  ```

  Although legal, an indexer with only a Get accessor is rarely good style. Strongly consider using a method in this case.

- Indexers can be overloaded (for more information, see 10.8.1 Indexer overloading).

See Also

C# Tutorials

Indexed Properties Tutorial

This tutorial shows how to implement a class that uses indexed properties. Indexed properties allow you to use a class that represents an array-like collection of several different kinds of things. You should complete the Indexers Tutorial before working through this tutorial.

Sample Files

See Indexed Properties Sample to download and build the sample files discussed in this tutorial.

Further Reading

- Indexers
- Accessors
- Indexers Tutorial
- 10.8 Indexers
- 10.8.1 Indexer overloading

Tutorial

Suppose you want to write a class, `Document`, which encapsulates a lengthy section of text. To allow easy implementation of various operations such as checking spelling, you might want to view the document as a virtual array of words, as well as of characters.

The following example shows a technique for implementing such a class. For each "indexed property," you define a nested class, which contains a reference back to the main class instance. A **readonly** field on the main class provides access to an instance of the nested class that defines each virtual array. Each of the nested classes defines an indexer, as well as other collection-like methods (a `Count` property, for example). The following example shows this for "`Words`" and "`Characters`."

> **Note** Use this technique sparingly! Only use this pattern if the abstraction provided by using array indexing operations significantly clarifies code that uses your class, and if the indexers have both Get and Set accessors.

Example

In this example the `Document` class is defined. Two indexed properties, `Words` and `Characters`, are used to perform some text operations on the `Document` object.

```
// indexedproperty.cs
using System;

public class Document
{
    // Type allowing the document to be viewed like an array of words:
    public class WordCollection
    {
        readonly Document document;  // The containing document

        internal WordCollection(Document d)
        {
           document = d;
        }

        // Helper function -- search character array "text", starting at
        // character "begin", for word number "wordCount." Returns false
        // if there are less than wordCount words. Sets "start" and
        // length" to the position and length of the word within text:
        private bool GetWord(char[] text, int begin, int wordCount,
                                        out int start, out int length)
        {
            int end = text.Length;
            int count = 0;
            int inWord = -1;
            start = length = 0;
```

```
        for (int i = begin; i <= end; ++i)
        {
            bool isLetter = i < end && Char.IsLetterOrDigit(text[i]);

            if (inWord >= 0)
            {
                if (!isLetter)
                {
                    if (count++ == wordCount)
                    {
                        start = inWord;
                        length = i - inWord;
                        return true;
                    }
                    inWord = -1;
                }
            }
            else
            {
                if (isLetter)
                    inWord = i;
            }
        }
        return false;
    }

    // Indexer to get and set words of the containing document:
    public string this[int index]
    {
        get
        {
            int start, length;
            if (GetWord(document.TextArray, 0, index, out start,
                                                      out length))
                return new string(document.TextArray, start, length);
            else
                throw new IndexOutOfRangeException();
        }
        set
        {
            int start, length;
            if (GetWord(document.TextArray, 0, index, out start,
                                                     out length))
            {
                // Replace the word at start/length with the
                // string "value":
                if (length == value.Length)
                {
                    Array.Copy(value.ToCharArray(), 0,
                               document.TextArray, start, length);
                }
                else
                {
                    char[] newText =
                        new char[document.TextArray.Length +
                                 value.Length - length];
                    Array.Copy(document.TextArray, 0, newText,
                                                   0, start);
                    Array.Copy(value.ToCharArray(), 0, newText,
                                         start, value.Length);
                    Array.Copy(document.TextArray, start + length,
                               newText, start + value.Length,
                              document.TextArray.Length - start
                                                        - length);
                    document.TextArray = newText;
                }
            }
            else
```

```
                throw new IndexOutOfRangeException();
            }
        }

        // Get the count of words in the containing document:
        public int Count
        {
            get
            {
                int count = 0, start = 0, length = 0;
                while (GetWord(document.TextArray, start + length, 0,
                                                out start, out length))
                    ++count;
                return count;
            }
        }
    }

    // Type allowing the document to be viewed like an "array"
    // of characters:
    public class CharacterCollection
    {
        readonly Document document;  // The containing document

        internal CharacterCollection(Document d)
        {
          document = d;
        }

        // Indexer to get and set characters in the containing document:
        public char this[int index]
        {
            get
            {
                return document.TextArray[index];
            }
            set
            {
                document.TextArray[index] = value;
            }
        }

        // Get the count of characters in the containing document:
        public int Count
        {
            get
            {
                return document.TextArray.Length;
            }
        }
    }

    // Because the types of the fields have indexers,
    // these fields appear as "indexed properties":
    public readonly WordCollection Words;
    public readonly CharacterCollection Characters;

    private char[] TextArray;  // The text of the document.

    public Document(string initialText)
    {
        TextArray = initialText.ToCharArray();
        Words = new WordCollection(this);
        Characters = new CharacterCollection(this);
    }

    public string Text
    {
```

```
        get
        {
           return new string(TextArray);
        }
    }
}

class Test
{
    static void Main()
    {
        Document d = new Document(
           "peter piper picked a peck of pickled peppers. How many pickled peppers did peter
piper pick?"
        );

        // Change word "peter" to "penelope":
        for (int i = 0; i < d.Words.Count; ++i)
        {
            if (d.Words[i] == "peter")
                d.Words[i] = "penelope";
        }

        // Change character "p" to "P"
        for (int i = 0; i < d.Characters.Count; ++i)
        {
            if (d.Characters[i] == 'p')
                d.Characters[i] = 'P';
        }

        Console.WriteLine(d.Text);
    }
}
```

Output

```
PeneloPe PiPer Picked a Peck of Pickled PePPers. How many Pickled PePPers did PeneloPe PiPer
Pick?
```

See Also

C# Tutorials

User-Defined Conversions Tutorial

This tutorial shows how to define and use conversions to or from classes or structs.

Sample Files

See User-Defined Conversions Sample to download and build the sample files discussed in this tutorial.

Further Reading

- explicit
- implicit
- () Operator
- Implicit Numeric Conversions Table
- operator
- 6.4 User-defined conversions
- 6.4.4 User-defined explicit conversions

Tutorial

C# allows programmers to declare conversions on classes or structs so that classes or structs can be converted to and/or from other classes or structs, or basic types. Conversions are defined like operators and are named for the type to which they convert.

In C#, conversions can be declared either as **implicit**, which occur automatically when required, or **explicit**, which require a cast to be called. All conversions must be **static**, and must either take the type the conversion is defined on, or return that type.

This tutorial introduces two examples. The first example shows how to declare and use conversions, and the second example demonstrates conversions between structs.

Example 1

In this example, a `RomanNumeral` type is declared, and several conversions to and from it are defined.

```
// conversion.cs
using System;

struct RomanNumeral
{
    public RomanNumeral(int value)
    {
       this.value = value;
    }
    // Declare a conversion from an int to a RomanNumeral. Note the
    // the use of the operator keyword. This is a conversion
    // operator named RomanNumeral:
    static public implicit operator RomanNumeral(int value)
    {
       // Note that because RomanNumeral is declared as a struct,
       // calling new on the struct merely calls the constructor
       // rather than allocating an object on the heap:
       return new RomanNumeral(value);
    }
    // Declare an explicit conversion from a RomanNumeral to an int:
    static public explicit operator int(RomanNumeral roman)
    {
       return roman.value;
    }
    // Declare an implicit conversion from a RomanNumeral to
    // a string:
    static public implicit operator string(RomanNumeral roman)
    {
       return("Conversion not yet implemented");
    }
    private int value;
```

```
}

class Test
{
    static public void Main()
    {
        RomanNumeral numeral;

        numeral = 10;

// Call the explicit conversion from numeral to int. Because it is
// an explicit conversion, a cast must be used:
        Console.WriteLine((int)numeral);

// Call the implicit conversion to string. Because there is no
// cast, the implicit conversion to string is the only
// conversion that is considered:
        Console.WriteLine(numeral);

// Call the explicit conversion from numeral to int and
// then the explicit conversion from int to short:
        short s = (short)numeral;

        Console.WriteLine(s);
    }
}
```

Output

```
10
Conversion not yet implemented
10
```

Example 2

This example defines two structs, `RomanNumeral` and `BinaryNumeral`, and demonstrates conversions between them.

```
// structconversion.cs
using System;

struct RomanNumeral
{
    public RomanNumeral(int value)
    {
        this.value = value;
    }
    static public implicit operator RomanNumeral(int value)
    {
        return new RomanNumeral(value);
    }
    static public implicit operator RomanNumeral(BinaryNumeral binary)
    {
        return new RomanNumeral((int)binary);
    }
    static public explicit operator int(RomanNumeral roman)
    {
         return roman.value;
    }
    static public implicit operator string(RomanNumeral roman)
    {
        return("Conversion not yet implemented");
    }
    private int value;
}
```

```
struct BinaryNumeral
{
    public BinaryNumeral(int value)
    {
        this.value = value;
    }
    static public implicit operator BinaryNumeral(int value)
    {
        return new BinaryNumeral(value);
    }
    static public implicit operator string(BinaryNumeral binary)
    {
        return("Conversion not yet implemented");
    }
    static public explicit operator int(BinaryNumeral binary)
    {
        return(binary.value);
    }

    private int value;
}

class Test
{
    static public void Main()
    {
        RomanNumeral roman;
        roman = 10;
        BinaryNumeral binary;
        // Perform a conversion from a RomanNumeral to a
        // BinaryNumeral:
        binary = (BinaryNumeral)(int)roman;
        // Performs a conversion from a BinaryNumeral to a RomanNumeral.
        // No cast is required:
        roman = binary;
        Console.WriteLine((int)binary);
        Console.WriteLine(binary);
    }
}
```

Output

```
10
Conversion not yet implemented
```

Code Discussion

- In the preceding example, the statement:

  ```
  binary = (BinaryNumeral)(int)roman;
  ```

  performs a conversion from a `RomanNumeral` to a `BinaryNumeral`. Because there is no direct conversion from `RomanNumeral` to `BinaryNumeral`, a cast is used to convert from a `RomanNumeral` to an **int**, and another cast to convert from an **int** to a `BinaryNumeral`.

- Also the statement:

  ```
  roman = binary;
  ```

  performs a conversion from a `BinaryNumeral` to a `RomanNumeral`. Because `RomanNumeral` defines an implicit conversion from `BinaryNumeral`, no cast is required.

See Also

Operator Overloading Tutorial

This tutorial demonstrates how user-defined classes can overload operators.

Sample Files

See Operator Overloading Sample to download and build the sample files discussed in this tutorial.

Further Reading

- 7.2.2 Operator overloading
- C# Operators
- Overloadable Operators
- true
- false
- bool

Tutorial

Operator overloading permits user-defined operator implementations to be specified for operations where one or both of the operands are of a user-defined class or struct type. The tutorial contains two examples. The first example shows how to use operator overloading to create a complex number class that defines complex addition. The second example shows how to use operator overloading to implement a three-valued logical type.

Example 1

This example shows how you can use operator overloading to create a complex number class `Complex` that defines complex addition. The program displays the imaginary and the real parts of the numbers and the addition result using an override of the **ToString** method.

```
// complex.cs
using System;

public struct Complex
{
   public int real;
   public int imaginary;

   public Complex(int real, int imaginary)
   {
      this.real = real;
      this.imaginary = imaginary;
   }

   // Declare which operator to overload (+), the types
   // that can be added (two Complex objects), and the
   // return type (Complex):
   public static Complex operator +(Complex c1, Complex c2)
   {
      return new Complex(c1.real + c2.real, c1.imaginary + c2.imaginary);
   }
   // Override the ToString method to display an complex number in the suitable format:
   public override string ToString()
   {
      return(String.Format("{0} + {1}i", real, imaginary));
   }

   public static void Main()
   {
      Complex num1 = new Complex(2,3);
      Complex num2 = new Complex(3,4);

      // Add two Complex objects (num1 and num2) through the
      // overloaded plus operator:
```

```
        Complex sum = num1 + num2;

       // Print the numbers and the sum using the overriden ToString method:
        Console.WriteLine("First complex number:  {0}",num1);
        Console.WriteLine("Second complex number: {0}",num2);
        Console.WriteLine("The sum of the two numbers: {0}",sum);

    }
}
```

Output

```
First complex number:  2 + 3i
Second complex number: 3 + 4i
The sum of the two numbers: 5 + 7i
```

Example 2

This example shows how operator overloading can be used to implement a three-valued logical type. The possible values of this type are `DBBool.dbTrue`, `DBBool.dbFalse`, and `DBBool.dbNull`, where the `dbNull` member indicates an unknown value.

> **Note** Defining the True and False operators is only useful for types that represent True, False, and Null (neither True nor False), as used in databases.

```
// dbbool.cs
using System;

public struct DBBool
{
   // The three possible DBBool values:
   public static readonly DBBool dbNull = new DBBool(0);
   public static readonly DBBool dbFalse = new DBBool(-1);
   public static readonly DBBool dbTrue = new DBBool(1);
   // Private field that stores -1, 0, 1 for dbFalse, dbNull, dbTrue:
   int value;

   // Private constructor. The value parameter must be -1, 0, or 1:
   DBBool(int value)
   {
      this.value = value;
   }

   // Implicit conversion from bool to DBBool. Maps true to
   // DBBool.dbTrue and false to DBBool.dbFalse:
   public static implicit operator DBBool(bool x)
   {
      return x? dbTrue: dbFalse;
   }

   // Explicit conversion from DBBool to bool. Throws an
   // exception if the given DBBool is dbNull, otherwise returns
   // true or false:
   public static explicit operator bool(DBBool x)
   {
      if (x.value == 0) throw new InvalidOperationException();
      return x.value > 0;
   }

   // Equality operator. Returns dbNull if either operand is dbNull,
   // otherwise returns dbTrue or dbFalse:
   public static DBBool operator ==(DBBool x, DBBool y)
   {
      if (x.value == 0 || y.value == 0) return dbNull;
      return x.value == y.value? dbTrue: dbFalse;
```

```
    }

    // Inequality operator. Returns dbNull if either operand is
    // dbNull, otherwise returns dbTrue or dbFalse:
    public static DBBool operator !=(DBBool x, DBBool y)
    {
       if (x.value == 0 || y.value == 0) return dbNull;
       return x.value != y.value? dbTrue: dbFalse;
    }

    // Logical negation operator. Returns dbTrue if the operand is
    // dbFalse, dbNull if the operand is dbNull, or dbFalse if the
    // operand is dbTrue:
    public static DBBool operator !(DBBool x)
    {
       return new DBBool(-x.value);
    }

    // Logical AND operator. Returns dbFalse if either operand is
    // dbFalse, dbNull if either operand is dbNull, otherwise dbTrue:
    public static DBBool operator &(DBBool x, DBBool y)
    {
       return new DBBool(x.value < y.value? x.value: y.value);
    }

    // Logical OR operator. Returns dbTrue if either operand is
    // dbTrue, dbNull if either operand is dbNull, otherwise dbFalse:
    public static DBBool operator |(DBBool x, DBBool y)
    {
       return new DBBool(x.value > y.value? x.value: y.value);
    }

    // Definitely true operator. Returns true if the operand is
    // dbTrue, false otherwise:
    public static bool operator true(DBBool x)
    {
       return x.value > 0;
    }

    // Definitely false operator. Returns true if the operand is
    // dbFalse, false otherwise:
    public static bool operator false(DBBool x)
    {
       return x.value < 0;
    }

    // Overload the conversion from DBBool to string:
    public static implicit operator string(DBBool x)
    {
       return x.value > 0 ? "dbTrue"
            : x.value < 0 ? "dbFalse"
            : "dbNull";
    }

    // Override the Object.Equals(object o) method:
    public override bool Equals(object o)
    {
       try
       {
          return (bool) (this == (DBBool) o);
       }
       catch
       {
          return false;
       }
    }

    // Override the Object.GetHashCode() method:
```

```
   public override int GetHashCode()
   {
      return value;
   }

   // Override the ToString method to convert DBBool to a string:
   public override string ToString()
   {
      switch (value)
      {
         case -1:
            return "DBBool.False";
         case 0:
            return "DBBool.Null";
         case 1:
            return "DBBool.True";
         default:
            throw new InvalidOperationException();
      }
   }
}

class Test
{
   static void Main()
   {
      DBBool a, b;
      a = DBBool.dbTrue;
      b = DBBool.dbNull;

      Console.WriteLine( "!{0} = {1}", a, !a);
      Console.WriteLine( "!{0} = {1}", b, !b);
      Console.WriteLine( "{0} & {1} = {2}", a, b, a & b);
      Console.WriteLine( "{0} | {1} = {2}", a, b, a | b);
      // Invoke the true operator to determine the Boolean
      // value of the DBBool variable:
      if (b)
         Console.WriteLine("b is definitely true");
      else
         Console.WriteLine("b is not definitely true");
   }
}
```

Output

```
!DBBool.True = DBBool.False
!DBBool.Null = DBBool.Null
DBBool.True & DBBool.Null = DBBool.Null
DBBool.True | DBBool.Null = DBBool.True
b is not definitely true
```

See Also

C# Tutorials

Delegates Tutorial

This tutorial demonstrates the delegate types. It shows how to map delegates to static and instance methods, and how to combine them (multicast).

Sample Files

See Delegates Sample to download and build the sample files discussed in this tutorial.

Further Reading

- delegate
- 15. Delegates
- Events Tutorial
- Asynchronous Delegates

Tutorial

A delegate in C# is similar to a function pointer in C or C++. Using a delegate allows the programmer to encapsulate a reference to a method inside a delegate object. The delegate object can then be passed to code which can call the referenced method, without having to know at compile time which method will be invoked. Unlike function pointers in C or C++, delegates are object-oriented, type-safe, and secure.

A delegate declaration defines a type that encapsulates a method with a particular set of arguments and return type. For static methods, a delegate object encapsulates the method to be called. For instance methods, a delegate object encapsulates both an instance and a method on the instance. If you have a delegate object and an appropriate set of arguments, you can invoke the delegate with the arguments.

An interesting and useful property of a delegate is that it does not know or care about the class of the object that it references. Any object will do; all that matters is that the method's argument types and return type match the delegate's. This makes delegates perfectly suited for "anonymous" invocation.

> **Note** Delegates run under the caller's security permissions, not the declarer's permissions.

This tutorial includes two examples:

- Example 1 shows how to declare, instantiate, and call a delegate.
- Example 2 shows how to combine two delegates.

In addition, it discusses the following topics:

- Delegates and Events
- Delegates vs. Interfaces

Example 1

The following example illustrates declaring, instantiating, and using a delegate. The `BookDB` class encapsulates a bookstore database that maintains a database of books. It exposes a method `ProcessPaperbackBooks`, which finds all paperback books in the database and calls a delegate for each one. The **delegate** type used is called `ProcessBookDelegate`. The `Test` class uses this class to print out the titles and average price of the paperback books.

The use of delegates promotes good separation of functionality between the bookstore database and the client code. The client code has no knowledge of how the books are stored or how the bookstore code finds paperback books. The bookstore code has no knowledge of what processing is done on the paperback books after it finds them.

```
// bookstore.cs
using System;

// A set of classes for handling a bookstore:
namespace Bookstore
{
   using System.Collections;

   // Describes a book in the book list:
```

```
   public struct Book
   {
      public string Title;        // Title of the book.
      public string Author;       // Author of the book.
      public decimal Price;       // Price of the book.
      public bool Paperback;      // Is it paperback?

      public Book(string title, string author, decimal price, bool paperBack)
      {
         Title = title;
         Author = author;
         Price = price;
         Paperback = paperBack;
      }
   }

   // Declare a delegate type for processing a book:
   public delegate void ProcessBookDelegate(Book book);

   // Maintains a book database.
   public class BookDB
   {
      // List of all books in the database:
      ArrayList list = new ArrayList();

      // Add a book to the database:
      public void AddBook(string title, string author, decimal price, bool paperBack)
      {
         list.Add(new Book(title, author, price, paperBack));
      }

      // Call a passed-in delegate on each paperback book to process it:
      public void ProcessPaperbackBooks(ProcessBookDelegate processBook)
      {
         foreach (Book b in list)
         {
            if (b.Paperback)
            // Calling the delegate:
               processBook(b);
         }
      }
   }
}

// Using the Bookstore classes:
namespace BookTestClient
{
   using Bookstore;

   // Class to total and average prices of books:
   class PriceTotaller
   {
      int countBooks = 0;
      decimal priceBooks = 0.0m;

      internal void AddBookToTotal(Book book)
      {
         countBooks += 1;
         priceBooks += book.Price;
      }

      internal decimal AveragePrice()
      {
         return priceBooks / countBooks;
      }
   }

   // Class to test the book database:
```

```
    class Test
    {
       // Print the title of the book.
       static void PrintTitle(Book b)
       {
          Console.WriteLine("   {0}", b.Title);
       }

       // Execution starts here.
       static void Main()
       {
          BookDB bookDB = new BookDB();

          // Initialize the database with some books:
          AddBooks(bookDB);

          // Print all the titles of paperbacks:
          Console.WriteLine("Paperback Book Titles:");
          // Create a new delegate object associated with the static
          // method Test.PrintTitle:
          bookDB.ProcessPaperbackBooks(new ProcessBookDelegate(PrintTitle));

          // Get the average price of a paperback by using
          // a PriceTotaller object:
          PriceTotaller totaller = new PriceTotaller();
          // Create a new delegate object associated with the nonstatic
          // method AddBookToTotal on the object totaller:
          bookDB.ProcessPaperbackBooks(new ProcessBookDelegate(totaller.AddBookToTotal));
          Console.WriteLine("Average Paperback Book Price: ${0:#.##}",
             totaller.AveragePrice());
       }

       // Initialize the book database with some test books:
       static void AddBooks(BookDB bookDB)
       {
          bookDB.AddBook("The C Programming Language",
             "Brian W. Kernighan and Dennis M. Ritchie", 19.95m, true);
          bookDB.AddBook("The Unicode Standard 2.0",
             "The Unicode Consortium", 39.95m, true);
          bookDB.AddBook("The MS-DOS Encyclopedia",
             "Ray Duncan", 129.95m, false);
          bookDB.AddBook("Dogbert's Clues for the Clueless",
             "Scott Adams", 12.00m, true);
       }
    }
 }
```

Output

```
 Paperback Book Titles:
    The C Programming Language
    The Unicode Standard 2.0
    Dogbert's Clues for the Clueless
 Average Paperback Book Price: $23.97
```

Code Discussion

- **Declaring a delegate** The following statement:

  ```
  public delegate void ProcessBookDelegate(Book book);
  ```

  declares a new delegate type. Each delegate type describes the number and types of the arguments, and the type of the return value of methods that it can encapsulate. Whenever a new set of argument types or return value type is needed, a new delegate type must be declared.

- **Instantiating a delegate** Once a delegate type has been declared, a delegate object must be created and associated with a particular method. Like all other objects, a new delegate object is created with a **new** expression. When creating a delegate, however, the argument passed to the **new** expression is special — it is written like a method call, but without the arguments to the method.

  The following statement:

  ```
  bookDB.ProcessPaperbackBooks(new ProcessBookDelegate(PrintTitle));
  ```

  creates a new delegate object associated with the static method `Test.PrintTitle`. The following statement:

  ```
  bookDB.ProcessPaperbackBooks(new
     ProcessBookDelegate(totaller.AddBookToTotal));
  ```

  creates a new delegate object associated with the nonstatic method `AddBookToTotal` on the object `totaller`. In both cases, this new delegate object is immediately passed to the `ProcessPaperbackBooks` method.

  Note that once a delegate is created, the method it is associated with never changes — delegate objects are immutable.

- **Calling a delegate** Once a delegate object is created, the delegate object is typically passed to other code that will call the delegate. A delegate object is called by using the name of the delegate object, followed by the parenthesized arguments to be passed to the delegate. An example of a delegate call is:

  ```
  processBook(b);
  ```

  A delegate can either be called synchronously, as in this example, or asynchronously by using BeginInvoke and EndInvoke methods.

Example 2

This example demonstrates composing delegates. A useful property of delegate objects is that they can be composed using the "**+**" operator. A composed delegate calls the two delegates it was composed from. Only delegates of the same type can be composed.

The "**-**" operator can be used to remove a component delegate from a composed delegate.

```
// compose.cs
using System;

delegate void MyDelegate(string s);

class MyClass
{
    public static void Hello(string s)
    {
        Console.WriteLine("  Hello, {0}!", s);
    }

    public static void Goodbye(string s)
    {
        Console.WriteLine("  Goodbye, {0}!", s);
    }

    public static void Main()
    {
        MyDelegate a, b, c, d;

        // Create the delegate object a that references
        // the method Hello:
        a = new MyDelegate(Hello);
        // Create the delegate object b that references
        // the method Goodbye:
        b = new MyDelegate(Goodbye);
```

```
        // The two delegates, a and b, are composed to form c:
        c = a + b;
        // Remove a from the composed delegate, leaving d,
        // which calls only the method Goodbye:
        d = c - a;

        Console.WriteLine("Invoking delegate a:");
        a("A");
        Console.WriteLine("Invoking delegate b:");
        b("B");
        Console.WriteLine("Invoking delegate c:");
        c("C");
        Console.WriteLine("Invoking delegate d:");
        d("D");
    }
}
```

Output

```
Invoking delegate a:
  Hello, A!
Invoking delegate b:
  Goodbye, B!
Invoking delegate c:
  Hello, C!
  Goodbye, C!
Invoking delegate d:
  Goodbye, D!
```

Delegates and Events

Delegates are ideally suited for use as events — notifications from one component to "listeners" about changes in that component. For more information on the use of delegates for events, see the Events Tutorial.

Delegates vs. Interfaces

Delegates and interfaces are similar in that they enable the separation of specification and implementation. Multiple independent authors can produce implementations that are compatible with an interface specification. Similarly, a delegate specifies the signature of a method, and authors can write methods that are compatible with the delegate specification. When should you use interfaces, and when should you use delegates?

Delegates are useful when:

- A single method is being called.
- A class may want to have multiple implementations of the method specification.
- It is desirable to allow using a static method to implement the specification.
- An event-like design pattern is desired (for more information, see the Events Tutorial).
- The caller has no need to know or obtain the object that the method is defined on.
- The provider of the implementation wants to "hand out" the implementation of the specification to only a few select components.
- Easy composition is desired.

Interfaces are useful when:

- The specification defines a set of related methods that will be called.
- A class typically implements the specification only once.
- The caller of the interface wants to cast to or from the interface type to obtain other interfaces or classes.

See Also

C# Tutorials

Events Tutorial

This tutorial shows how to declare, invoke, and hook up to events in C#.

Sample Files

See Events Sample to download and build the sample files discussed in this tutorial.

Further Reading

- event
- delegate
- Handling and Raising Events
- Delegates Tutorial
- Introduction to Event Handlers in Windows Forms

Tutorial

An *event* in C# is a way for a class to provide notifications to clients of that class when some interesting thing happens to an object. The most familiar use for events is in graphical user interfaces; typically, the classes that represent controls in the interface have events that are notified when the user does something to the control (for example, click a button).

Events, however, need not be used only for graphical interfaces. Events provide a generally useful way for objects to signal state changes that may be useful to clients of that object. Events are an important building block for creating classes that can be reused in a large number of different programs.

Events are declared using delegates. If you have not yet studied the Delegates Tutorial, you should do so before continuing. Recall that a delegate object encapsulates a method so that it can be called anonymously. An event is a way for a class to allow clients to give it delegates to methods that should be called when the event occurs. When the event occurs, the delegate(s) given to it by its clients are invoked.

In addition to the examples on declaring, invoking, and hooking up to events, this tutorial also introduces the following topics:

- Events and Inheritance
- Events in Interfaces
- .NET Framework Guidelines

Example 1

The following simple example shows a class, `ListWithChangedEvent`, which is similar to the standard `ArrayList` class, but also invokes a `Changed` event whenever the contents of the list change. Such a general-purpose class could be used in numerous ways in a large program.

For example, a word processor might maintain a list of the open documents. Whenever this list changes, many different objects in the word processor might need to be notified so that the user interface could be updated. By using events, the code that maintains the list of documents doesn't need to know who needs to be notified — once the list of documents is changed, the event is automatically invoked and every object that needs to be notified is correctly notified. By using events, the modularity of the program is increased.

```
// events1.cs
using System;
namespace MyCollections
{
   using System.Collections;

   // A delegate type for hooking up change notifications.
   public delegate void ChangedEventHandler(object sender, EventArgs e);

   // A class that works just like ArrayList, but sends event
   // notifications whenever the list changes.
   public class ListWithChangedEvent: ArrayList
   {
      // An event that clients can use to be notified whenever the
      // elements of the list change.
```

```
      public event ChangedEventHandler Changed;

      // Invoke the Changed event; called whenever list changes
      protected virtual void OnChanged(EventArgs e)
      {
         if (Changed != null)
            Changed(this, e);
      }

      // Override some of the methods that can change the list;
      // invoke event after each
      public override int Add(object value)
      {
         int i = base.Add(value);
         OnChanged(EventArgs.Empty);
         return i;
      }

      public override void Clear()
      {
         base.Clear();
         OnChanged(EventArgs.Empty);
      }

      public override object this[int index]
      {
         set
         {
            base[index] = value;
            OnChanged(EventArgs.Empty);
         }
      }
   }
}

namespace TestEvents
{
   using MyCollections;

   class EventListener
   {
      private ListWithChangedEvent List;

      public EventListener(ListWithChangedEvent list)
      {
         List = list;
         // Add "ListChanged" to the Changed event on "List".
         List.Changed += new ChangedEventHandler(ListChanged);
      }

      // This will be called whenever the list changes.
      private void ListChanged(object sender, EventArgs e)
      {
         Console.WriteLine("This is called when the event fires.");
      }

      public void Detach()
      {
         // Detach the event and delete the list
         List.Changed -= new ChangedEventHandler(ListChanged);
         List = null;
      }
   }

   class Test
   {
      // Test the ListWithChangedEvent class.
      public static void Main()
```

```
        {
        // Create a new list.
        ListWithChangedEvent list = new ListWithChangedEvent();

        // Create a class that listens to the list's change event.
        EventListener listener = new EventListener(list);

        // Add and remove items from the list.
        list.Add("item 1");
        list.Clear();
        listener.Detach();
        }
    }
}
```

Output

```
This is called when the event fires.
This is called when the event fires.
```

Code Discussion

- **Declaring an event** To declare an event inside a class, first a delegate type for the event must be declared, if none is already declared.

  ```
  public delegate void ChangedEventHandler(object sender, EventArgs e);
  ```

  The delegate type defines the set of arguments that are passed to the method that handles the event. Multiple events can share the same delegate type, so this step is only necessary if no suitable delegate type has already been declared.

  Next, the event itself is declared.

  ```
  public event ChangedEventHandler Changed;
  ```

  An event is declared like a field of delegate type, except that the keyword **event** precedes the event declaration, following the modifiers. Events usually are declared public, but any accessibility modifier is allowed.

- **Invoking an event** Once a class has declared an event, it can treat that event just like a field of the indicated delegate type. The field will either be null, if no client has hooked up a delegate to the event, or else it refers to a delegate that should be called when the event is invoked. Thus, invoking an event is generally done by first checking for null and then calling the event.

  ```
  if (Changed != null)
  Changed(this, e);
  ```

  Invoking an event can only be done from within the class that declared the event.

- **Hooking up to an event** From outside the class that declared it, an event looks like a field, but access to that field is very restricted. The only things that can be done are:
  - Compose a new delegate onto that field.
  - Remove a delegate from a (possibly composite) field.

  This is done with the **+=** and **-=** operators. To begin receiving event invocations, client code first creates a delegate of the event type that refers to the method that should be invoked from the event. Then it composes that delegate onto any other delegates that the event might be connected to using **+=**.

  ```
  // Add "ListChanged" to the Changed event on "List":
  List.Changed += new ChangedEventHandler(ListChanged);
  ```

  When the client code is done receiving event invocations, it removes its delegate from the event by using operator **-=**.

```
// Detach the event and delete the list:
List.Changed -= new ChangedEventHandler(ListChanged);
```

Events and Inheritance

When creating a general component that can be derived from, what seems to be a problem sometimes arises with events. Since events can only be invoked from within the class that declared them, derived classes cannot directly invoke events declared within the base class. Although this is sometimes what is desired, often it is appropriate to give the derived class the freedom to invoke the event. This is typically done by creating a protected invoking method for the event. By calling this invoking method, derived classes can invoke the event. For even more flexibility, the invoking method is often declared as virtual, which allows the derived class to override it. This allows the derived class to intercept the events that the base class is invoking, possibly doing its own processing of them.

In the preceding example, this has been done with the `OnChanged` method. A derived class could call or override this method if it needed to.

Events in Interfaces

One other difference between events and fields is that an event can be placed in an interface while a field cannot. When implementing the interface, the implementing class must supply a corresponding event in the class that implements the interface.

.NET Framework Guidelines

Although the C# language allows events to use any delegate type, the .NET Framework has some stricter guidelines on the delegate types that should be used for events. If you intend for your component to be used with the .NET Framework, you probably will want to follow these guidelines.

The .NET Framework guidelines indicate that the delegate type used for an event should take two parameters, an "object source" parameter indicating the source of the event, and an "e" parameter that encapsulates any additional information about the event. The type of the "e" parameter should derive from the **EventArgs** class. For events that do not use any additional information, the .NET Framework has already defined an appropriate delegate type: **EventHandler**.

Example 2

The following example is a modified version of Example 1 that follows the .NET Framework guidelines. The example uses the **EventHandler** delegate type.

```
// events2.cs
using System;
namespace MyCollections
{
   using System.Collections;

   // A class that works just like ArrayList, but sends event
   // notifications whenever the list changes:
   public class ListWithChangedEvent: ArrayList
   {
      // An event that clients can use to be notified whenever the
      // elements of the list change:
      public event EventHandler Changed;

      // Invoke the Changed event; called whenever list changes:
      protected virtual void OnChanged(EventArgs e)
      {
         if (Changed != null)
            Changed(this,e);
      }

      // Override some of the methods that can change the list;
      // invoke event after each:
      public override int Add(object value)
      {
         int i = base.Add(value);
         OnChanged(EventArgs.Empty);
```

```
            return i;
        }

        public override void Clear()
        {
            base.Clear();
            OnChanged(EventArgs.Empty);
        }

        public override object this[int index]
        {
            set
            {
                base[index] = value;
                OnChanged(EventArgs.Empty);
            }
        }
    }
}

namespace TestEvents
{
    using MyCollections;

    class EventListener
    {
        private ListWithChangedEvent List;

        public EventListener(ListWithChangedEvent list)
        {
            List = list;
            // Add "ListChanged" to the Changed event on "List":
            List.Changed += new EventHandler(ListChanged);
        }

        // This will be called whenever the list changes:
        private void ListChanged(object sender, EventArgs e)
        {
            Console.WriteLine("This is called when the event fires.");
        }

        public void Detach()
        {
            // Detach the event and delete the list:
            List.Changed -= new EventHandler(ListChanged);
            List = null;
        }
    }

    class Test
    {
        // Test the ListWithChangedEvent class:
        public static void Main()
        {
        // Create a new list:
        ListWithChangedEvent list = new ListWithChangedEvent();

        // Create a class that listens to the list's change event:
        EventListener listener = new EventListener(list);

        // Add and remove items from the list:
        list.Add("item 1");
        list.Clear();
        listener.Detach();
        }
    }
}
```

Output

```
This is called when the event fires.
This is called when the event fires.
```

See Also

C# Tutorials

Explicit Interface Implementation Tutorial

This tutorial demonstrates how to explicitly implement interface members and how to access those members from the interface instances.

Sample Files

See Explicit Interface Implementation Sample to download and build the sample files discussed in this tutorial.

Further Reading

- interface
- 13.4 Interface implementations
- 13.4.1 Explicit interface member implementations

Tutorial

A class that implements an interface can explicitly implement a member of that interface. When a member is explicitly implemented, it cannot be accessed through a class instance, but only through an instance of the interface. This tutorial contains two examples. The first example illustrates how to explicitly implement and access interface members. The second example shows how to implement two interfaces that have the same member names.

Example 1

This example declares an interface `IDimensions`, and a class `Box`, which explicitly implements the interface members `Length` and `Width`. The members are accessed through the interface instance `myDimensions`.

```
// explicit1.cs
interface IDimensions
{
   float Length();
   float Width();
}

class Box : IDimensions
{
   float lengthInches;
   float widthInches;

   public Box(float length, float width)
   {
      lengthInches = length;
      widthInches = width;
   }
   // Explicit interface member implementation:
   float IDimensions.Length()
   {
      return lengthInches;
   }
   // Explicit interface member implementation:
   float IDimensions.Width()
   {
      return widthInches;
   }

   public static void Main()
   {
      // Declare a class instance "myBox":
      Box myBox = new Box(30.0f, 20.0f);
      // Declare an interface instance "myDimensions":
      IDimensions myDimensions = (IDimensions) myBox;
      // Print out the dimensions of the box:
      /* The following commented lines would produce compilation
         errors because they try to access an explicitly implemented
         interface member from a class instance:                    */
```

```
      //System.Console.WriteLine("Length: {0}", myBox.Length());
      //System.Console.WriteLine("Width: {0}", myBox.Width());
      /* Print out the dimensions of the box by calling the methods
         from an instance of the interface:                        */
      System.Console.WriteLine("Length: {0}", myDimensions.Length());
      System.Console.WriteLine("Width: {0}", myDimensions.Width());
   }
}
```

Output

```
Length: 30
Width: 20
```

Code Discussion

- Notice that the following lines, in the Main method, are commented out because they would produce compilation errors. An interface member that is explicitly implemented cannot be accessed from a class instance:

  ```
  //System.Console.WriteLine("Length: {0}", myBox.Length());
  //System.Console.WriteLine("Width: {0}", myBox.Width());
  ```

- Notice also that the following lines, in the Main method, successfully print out the dimensions of the box because the methods are being called from an instance of the interface:

  ```
  System.Console.WriteLine("Length: {0}", myDimensions.Length());
  System.Console.WriteLine("Width: {0}", myDimensions.Width());
  ```

Example 2

Explicit interface implementation also allows the programmer to inherit two interfaces that share the same member names and give each interface member a separate implementation. This example displays the dimensions of a box in both metric and English units. The `Box` class inherits two interfaces `IEnglishDimensions` and `IMetricDimensions`, which represent the different measurement systems. Both interfaces have identical member names, `Length` and `Width`.

```
// explicit2.cs
// Declare the English units interface:
interface IEnglishDimensions
{
   float Length();
   float Width();
}
// Declare the metric units interface:
interface IMetricDimensions
{
   float Length();
   float Width();
}
// Declare the "Box" class that implements the two interfaces:
// IEnglishDimensions and IMetricDimensions:
class Box : IEnglishDimensions, IMetricDimensions
{
   float lengthInches;
   float widthInches;
   public Box(float length, float width)
   {
      lengthInches = length;
      widthInches = width;
   }
// Explicitly implement the members of IEnglishDimensions:
   float IEnglishDimensions.Length()
   {
```

```
      return lengthInches;
   }
   float IEnglishDimensions.Width()
   {
      return widthInches;
   }
// Explicitly implement the members of IMetricDimensions:
   float IMetricDimensions.Length()
   {
      return lengthInches * 2.54f;
   }
   float IMetricDimensions.Width()
   {
      return widthInches * 2.54f;
   }
   public static void Main()
   {
      // Declare a class instance "myBox":
      Box myBox = new Box(30.0f, 20.0f);
      // Declare an instance of the English units interface:
      IEnglishDimensions eDimensions = (IEnglishDimensions) myBox;
      // Declare an instance of the metric units interface:
      IMetricDimensions mDimensions = (IMetricDimensions) myBox;
      // Print dimensions in English units:
      System.Console.WriteLine("Length(in): {0}", eDimensions.Length());
      System.Console.WriteLine("Width (in): {0}", eDimensions.Width());
      // Print dimensions in metric units:
      System.Console.WriteLine("Length(cm): {0}", mDimensions.Length());
      System.Console.WriteLine("Width (cm): {0}", mDimensions.Width());
   }
}
```

Output

```
Length(in): 30
Width (in): 20
Length(cm): 76.2
Width (cm): 50.8
```

Code Discussion

If you want to make the default measurements in English units, implement the methods `Length` and `Width` normally, and explicitly implement the `Length` and `Width` methods from the `IMetricDimensions` interface:

```
// Normal implementation:
public float Length()
{
   return lengthInches;
}
public float Width()
{
   return widthInches;
}
// Explicit implementation:
float IMetricDimensions.Length()
{
   return lengthInches * 2.54f;
}
float IMetricDimensions.Width()
{
   return widthInches * 2.54f;
}
```

In this case, you can access the English units from the class instance and access the metric units from the interface instance:

```
System.Console.WriteLine("Length(in): {0}", myBox.Length());
System.Console.WriteLine("Width (in): {0}", myBox.Width());
System.Console.WriteLine("Length(cm): {0}", mDimensions.Length());
System.Console.WriteLine("Width (cm): {0}", mDimensions.Width());
```

See Also

C# Tutorials

Conditional Methods Tutorial

This tutorial demonstrates conditional methods, which provide a powerful mechanism by which calls to methods can be included or omitted depending on whether a preprocessor symbol is defined.

Sample Files

See Conditional Methods Sample to download and build the sample files discussed in this tutorial.

Further Reading

- Command-Line Arguments
- 17.4.2 The Conditional attribute
- Conditional
- #define
- /define (Preprocessor Definition)
- System.Diagnostics Namespace
- Debug.Assert Method
- Assertions in Managed Code

Tutorial

Conditional methods allow developers to create methods whose calls can be placed in the code and then either included or omitted during compilation based on a preprocessing symbol.

Suppose that you want to enable some assertion code in the debug builds and disable it in the retail builds. In C++, there is more than one way to have this functionality in your code, for example:

- Using **#ifdef**, define both debug and release versions of a macro. The debug version calls the tracing code, and the release version does nothing. Because C# doesn't support macros, this method doesn't work.
- Have two implementations of the code being called. That is, in the debug version, have full functionality, and in the retail version, have empty stubs for the methods. Users then choose which one to include when linking the project. The problem with this approach is that the retail builds contain calls to empty methods, and the configuration is more complex.

C# conditional methods provide a simple solution to this problem that is similar to the first approach listed above. There are two basic mechanisms to do this:

- **#define** the preprocessing identifier in the source code directly. (See an example of this approach in Conditional.)
- Define the preprocessing identifier on the C# command line via the /define option (/d). This approach is used in the following example.

Conditional methods are used in the .NET Framework. The **System.Diagnostics** namespace contains a number of classes that support tracing and debugging in applications. Use the **System.Diagnostics.Trace** and **System.Diagnostics.Debug** classes to add sophisticated tracing and debugging to your application (functionality that can be compiled out of your retail builds through the use of conditional methods).

The example below shows how to implement a very simple tracing mechanism using conditional methods. **System.Diagnostics.Trace** provides much more sophisticated tracing mechanisms, but it uses the fundamental mechanism below to provide this functionality.

Example

This example consists of two source files: the first file is the library that provides a tracing mechanism, and the second file is the client program that uses this library.

File #1: Creating Conditional Methods

The code below shows a simple library that provides a tracing mechanism that displays trace messages to the system console. Clients can embed trace calls in the code and then be able to control whether the tracing gets called by defining symbols in their own compilation phase.

```
// CondMethod.cs
// compile with: /target:library /d:DEBUG
using System;
using System.Diagnostics;
namespace TraceFunctions
{
   public class Trace
   {
       [Conditional("DEBUG")]
       public static void Message(string traceMessage)
       {
           Console.WriteLine("[TRACE] - " + traceMessage);
       }
   }
}
```

Code Discussion

The following line:

```
[Conditional("DEBUG")]
```

marks the `Message` method as being conditional (via the **Conditional** attribute). The **Conditional** attribute takes one parameter — the preprocessing identifier that controls whether the method call is included when clients are compiled. If the preprocessing identifier is defined, the method is called; otherwise, the call is never inserted in the client's code.

There are restrictions on which methods can be marked as conditional; see 17.4.2 The Conditional attribute in the *C# Language Specification* for more information.

File #2: Using the Conditional Method

The following client program uses the `Trace` class defined in file #1 to do some simple tracing.

```
// TraceTest.cs
// compile with: /reference:CondMethod.dll
// arguments: A B C
using System;
using TraceFunctions;

public class TraceClient
{
   public static void Main(string[] args)
   {
      Trace.Message("Main Starting");

      if (args.Length == 0)
      {
          Console.WriteLine("No arguments have been passed");
      }
      else
      {
          for( int i=0; i < args.Length; i++)
          {
              Console.WriteLine("Arg[{0}] is [{1}]",i,args[i]);
          }
      }

       Trace.Message("Main Ending");
   }
}
```

Code Discussion

Conditional code is included in client code depending on whether the preprocessing identifier is defined when the client code gets compiled.

Compiling with client code with the /d:DEBUG flag means that the compiler inserts the call to the `Trace` method. If the symbol is not defined, the call is never made.

Sample Run

The command:

```
tracetest A B C
```

gives the following output:

```
[TRACE] - Main Starting
Arg[0] is [A]
Arg[1] is [B]
Arg[2] is [C]
[TRACE] - Main Ending
```

The command:

```
tracetest
```

gives the following output:

```
[TRACE] - Main Starting
No arguments have been passed
[TRACE] - Main Ending
```

See Also

C# Tutorials

XML Documentation Tutorial

This tutorial shows how to document code using XML.

Sample Files

See XML Documentation Sample to download and build the sample files discussed in this tutorial.

Further Reading

- /doc (Process Documentation Comments)
- Recommended Tags for Documentation Comments
- Delimiters for Documentation Tags
- Processing the XML File
- XML Glossary

Tutorial

C# provides a mechanism for developers to document their code using XML. In source code files, lines that begin with /// and that precede a user-defined type such as a class, delegate, or interface; a member such as a field, event, property, or method; or a namespace declaration can be processed as comments and placed in a file.

Example

The following sample provides a basic overview of a type that has been documented. To compile the example, type the following command line:

```
csc XMLsample.cs /doc:XMLsample.xml
```

This will create the XML file XMLsample.xml, which you can view in your browser or by using the TYPE command.

```
// XMLsample.cs
// compile with: /doc:XMLsample.xml
using System;

/// <summary>
/// Class level summary documentation goes here.</summary>
/// <remarks>
/// Longer comments can be associated with a type or member
/// through the remarks tag</remarks>
public class SomeClass
{
   /// <summary>
   /// Store for the name property</summary>
   private string myName = null;

   /// <summary>
   /// The class constructor. </summary>
   public SomeClass()
   {
       // TODO: Add Constructor Logic here
   }

   /// <summary>
   /// Name property </summary>
   /// <value>
   /// A value tag is used to describe the property value</value>
   public string Name
   {
      get
      {
         if ( myName == null )
         {
```

```
            throw new Exception("Name is null");
         }

         return myName;
      }
   }

   /// <summary>
   /// Description for SomeMethod.</summary>
   /// <param name="s"> Parameter description for s goes here</param>
   /// <seealso cref="String">
   /// You can use the cref attribute on any tag to reference a type or member
   /// and the compiler will check that the reference exists. </seealso>
   public void SomeMethod(string s)
   {
   }

   /// <summary>
   /// Some other method. </summary>
   /// <returns>
   /// Return results are described through the returns tag.</returns>
   /// <seealso cref="SomeMethod(string)">
   /// Notice the use of the cref attribute to reference a specific method </seealso>
   public int SomeOtherMethod()
   {
      return 0;
   }

   /// <summary>
   /// The entry point for the application.
   /// </summary>
   /// <param name="args"> A list of command line arguments</param>
   public static int Main(String[] args)
   {
      // TODO: Add code to start application here

       return 0;
   }
}
```

Code Discussion

XML documentation starts with ///. When you create a new project, the wizards put some starter /// lines in for you. The processing of these comments has some restrictions:

- The documentation must be well-formed XML. If the XML is not well-formed, a warning is generated and the documentation file will contain a comment saying that an error was encountered. For more information on well-formed XML, see XML Glossary.
- Developers are free to create their own set of tags. There is a recommended set of tags (see the Further Reading section). Some of the recommended tags have special meanings:
    - The <param> tag is used to describe parameters. If used, the compiler will verify that the parameter exists and that all parameters are described in the documentation. If the verification failed, the compiler issues a warning.
    - The **cref** attribute can be attached to any tag to provide a reference to a code element. The compiler will verify that this code element exists. If the verification failed, the compiler issues a warning. The compiler also respects any **using** statements when looking for a type described in the **cref** attribute.
    - The <summary> tag is used by IntelliSense inside Visual Studio to display additional information about a type or member.

Sample Output

Here is the resulting XML file from the class above:

```
<?xml version="1.0"?>
<doc>
```

```
    <assembly>
        <name>xmlsample</name>
    </assembly>
    <members>
        <member name="T:SomeClass">
            <summary>
            Class level summary documentation goes here.</summary>
            <remarks>
            Longer comments can be associated with a type or member
            through the remarks tag</remarks>
        </member>
        <member name="F:SomeClass.myName">
            <summary>
            Store for the name property</summary>
        </member>
        <member name="M:SomeClass.#ctor">
            <summary>The class constructor.</summary>
        </member>
        <member name="M:SomeClass.SomeMethod(System.String)">
            <summary>
            Description for SomeMethod.</summary>
            <param name="s"> Parameter description for s goes here</param>
            <seealso cref="T:System.String">
            You can use the cref attribute on any tag to reference a type or member
            and the compiler will check that the reference exists. </seealso>
        </member>
        <member name="M:SomeClass.SomeOtherMethod">
            <summary>
            Some other method. </summary>
            <returns>
            Return results are described through the returns tag.</returns>
            <seealso cref="M:SomeClass.SomeMethod(System.String)">
            Notice the use of the cref attribute to reference a specific method </seealso>
        </member>
        <member name="M:SomeClass.Main(System.String[])">
            <summary>
            The entry point for the application.
            </summary>
            <param name="args"> A list of command line arguments</param>
        </member>
        <member name="P:SomeClass.Name">
            <summary>
            Name property </summary>
            <value>
            A value tag is used to describe the property value</value>
        </member>
    </members>
</doc>
```

Note The XML file does not provide full information about the type and members (for example, it does not contain any type information). To get full information about a type or member, the documentation file must be used in conjunction with reflection on the actual type or member.

See Also

C# Tutorials

Platform Invoke Tutorial

Platform Invocation Services (PInvoke) allows managed code to call unmanaged functions that are implemented in a DLL.

This tutorial shows you what you need to do to be able to call unmanaged DLL functions from C#. The attributes discussed in the tutorial allow you to call these functions and have data types be marshaled correctly.

Sample Files

See Platform Invoke Sample to download and build the sample files discussed in this tutorial.

Platform Invoke

Further Reading

- A Closer Look at Platform Invoke
- Using Attributes
- DllImportAttribute Class
- MarshalAsAttribute Class
- StructLayoutAttribute Class
- InAttribute Class
- OutAttribute Class

Tutorial

There are two ways that C# code can directly call unmanaged code:

- Directly call a function exported from a DLL.
- Call an interface method on a COM object (for more information, see COM Interop Part 1: C# Client Tutorial).

For both techniques, you must provide the C# compiler with a declaration of the unmanaged function, and you may also need to provide the C# compiler with a description of how to marshal the parameters and return value to and from the unmanaged code.

The tutorial consists of the following topics:

- Calling a DLL Export Directly from C#
- Default Marshaling and Specifying Custom Marshaling for Parameters to Unmanaged Methods
- Specifying Custom Marshaling for User-Defined Structs
- Registering Callback Methods

The tutorial includes the following examples:

- Example 1 Using DllImport
- Example 2 Overriding Default Marshaling
- Example 3 Specifying Custom Marshaling

Calling a DLL Export Directly from C#

To declare a method as having an implementation from a DLL export, do the following:

- Declare the method with the **static** and **extern** C# keywords.
- Attach the **DllImport** attribute to the method. The **DllImport** attribute allows you to specify the name of the DLL that contains the method. The common practice is to name the C# method the same as the exported method, but you can also use a different name for the C# method.
- Optionally, specify custom marshaling information for the method's parameters and return value, which will override the .NET Framework default marshaling.

Example 1

This example shows you how to use the **DllImport** attribute to output a message by calling `puts` from `msvcrt.dll`.

```
// PInvokeTest.cs
```

```
using System;
using System.Runtime.InteropServices;

class PlatformInvokeTest
{
    [DllImport("msvcrt.dll")]
    public static extern int puts(string c);
    [DllImport("msvcrt.dll")]
    internal static extern int _flushall();

    public static void Main()
    {
        puts("Test");
        _flushall();
    }
}
```

Output

```
Test
```

Code Discussion

The preceding example shows the minimum requirements for declaring a C# method that is implemented in an unmanaged DLL. The method `PlatformInvokeTest.puts` is declared with the **static** and **extern** modifiers and has the **DllImport** attribute which tells the compiler that the implementation comes from `msvcrt.dll`, using the default name of `puts`. To use a different name for the C# method such as `putstring`, you must use the **EntryPoint** option in the **DllImport** attribute, that is:

```
[DllImport("msvcrt.dll", EntryPoint="puts")]
```

For more information on the syntax of the **DllImport** attribute, see DllImportAttribute Class.

Default Marshaling and Specifying Custom Marshaling for Parameters to Unmanaged Methods

When calling an unmanaged function from C# code, the common language runtime must marshal the parameters and return values.

For every .NET Framework type there is a default unmanaged type, which the common language runtime will use to marshal data across a managed to unmanaged function call. For example, the default marshaling for C# string values is to the type LPTSTR (pointer to TCHAR char buffer). You can override the default marshaling using the **MarshalAs** attribute in the C# declaration of the unmanaged function.

Example 2

This example uses the **DllImport** attribute to output a string. It also shows you how to override the default marshaling of the function parameters by using the **MarshalAs** attribute.

```
// Marshal.cs
using System;
using System.Runtime.InteropServices;

class PlatformInvokeTest
{
    [DllImport("msvcrt.dll")]
    public static extern int puts(
        [MarshalAs(UnmanagedType.LPStr)]
        string m);
    [DllImport("msvcrt.dll")]
    internal static extern int _flushall();


    public static void Main()
```

```
    {
        puts("Hello World!");
        _flushall();
    }
}
```

Output

When you run this example, the string,

```
Hello World!
```

will display at the console.

Code Discussion

In the preceding example, the default marshaling for the parameter to the `puts` function has been overridden from the default of LPTSTR to LPSTR.

The **MarshalAs** attribute can be placed on method parameters, method return values, and fields of structs and classes. To set the marshaling of a method return value, place the **MarshalAs** attribute in an attribute block on the method with the return attribute location override. For example, to explicitly set the marshaling for the return value of the `puts` method:

```
...
[DllImport("msvcrt.dll")]
[return : MarshalAs(UnmanagedType.I4)]
public static extern int puts(
...
```

For more information on the syntax of the **MarshalAs** attribute, see MarshalAsAttribute Class.

> **Note** The **In** and **Out** attributes can be used to annotate parameters to unmanaged methods. They behave in a similar manner to the **in** and **out** modifiers in MIDL source files. Note that the **Out** attribute is different from the C# parameter modifier, out. For more information on the **In** and **Out** attributes, see InAttribute Class and OutAttribute Class.

Specifying Custom Marshaling for User-Defined Structs

You can specify custom marshaling attributes for fields of structs and classes passed to or from unmanaged functions. You do this by adding **MarshalAs** attributes to the fields of the struct or class. You must also use the **StructLayout** attribute to set the layout of the struct, optionally to control the default marshaling of string members, and to set the default packing size.

Example 3

This example demonstrates how to specify custom marshaling attributes for a struct.

Consider the following C structure:

```
typedef struct tagLOGFONT
{
   LONG lfHeight;
   LONG lfWidth;
   LONG lfEscapement;
   LONG lfOrientation;
   LONG lfWeight;
   BYTE lfItalic;
   BYTE lfUnderline;
   BYTE lfStrikeOut;
   BYTE lfCharSet;
   BYTE lfOutPrecision;
   BYTE lfClipPrecision;
   BYTE lfQuality;
   BYTE lfPitchAndFamily;
```

```
    TCHAR lfFaceName[LF_FACESIZE];
} LOGFONT;
```

In C#, you can describe the preceding struct by using the **StructLayout** and **MarshalAs** attributes as follows:

```
// logfont.cs
// compile with: /target:module
using System;
using System.Runtime.InteropServices;

[StructLayout(LayoutKind.Sequential)]
public class LOGFONT
{
    public const int LF_FACESIZE = 32;
    public int lfHeight;
    public int lfWidth;
    public int lfEscapement;
    public int lfOrientation;
    public int lfWeight;
    public byte lfItalic;
    public byte lfUnderline;
    public byte lfStrikeOut;
    public byte lfCharSet;
    public byte lfOutPrecision;
    public byte lfClipPrecision;
    public byte lfQuality;
    public byte lfPitchAndFamily;
    [MarshalAs(UnmanagedType.ByValTStr, SizeConst=LF_FACESIZE)]
    public string lfFaceName;
}
```

For more information on the syntax of the **StructLayout** attribute, see StructLayoutAttribute Class.

The structure can then be used in C# code as shown below:

```
// pinvoke.cs
// compile with: /addmodule:logfont.netmodule
using System;
using System.Runtime.InteropServices;

class PlatformInvokeTest
{
      [DllImport("gdi32.dll", CharSet=CharSet.Auto)]
      public static extern IntPtr CreateFontIndirect(
            [In, MarshalAs(UnmanagedType.LPStruct)]
            LOGFONT lplf   // characteristics
            );

      [DllImport("gdi32.dll")]
      public static extern bool DeleteObject(
            IntPtr handle
            );

      public static void Main()
      {
            LOGFONT lf = new LOGFONT();
            lf.lfHeight = 9;
            lf.lfFaceName = "Arial";
            IntPtr handle = CreateFontIndirect(lf);

            if (IntPtr.Zero == handle)
            {
                  Console.WriteLine("Can't creates a logical font.");
            }
            else
            {
```

```
                    if (IntPtr.Size == 4)
                          Console.WriteLine("{0:X}", handle.ToInt32());
                    else
                          Console.WriteLine("{0:X}", handle.ToInt64());

                    // Delete the logical font created.
                    if (!DeleteObject(handle))
                         Console.WriteLine("Can't delete the logical font");
              }
        }
  }
```

Sample Run

```
C30A0AE5
```

Code Discussion

In the preceding example, the `CreateFontIndirect` method is using a parameter of the type LOGFONT. The **MarshalAs** and **In** attributes are used to qualify the parameter. The program displays the numeric value returned by the method as a hexadecimal uppercase string.

Registering Callback Methods

To register a managed callback that calls an unmanaged function, declare a delegate with the same argument list and pass an instance of it via PInvoke. On the unmanaged side it will appear as a function pointer. For more information about PInvoke and callback, see A Closer Look at Platform Invoke.

For example, consider the following unmanaged function, `MyFunction`, which requires callback as one of the arguments:

```
typedef void (__stdcall *PFN_MYCALLBACK)();
int __stdcall MyFunction(PFN_ MYCALLBACK callback);
```

To call `MyFunction` from managed code, declare the delegate, attach DllImport to the function declaration, and optionally marshal any parameters or the return value:

```
public delegate void MyCallback();
[DllImport("MYDLL.DLL")]
public static extern void MyFunction(MyCallback callback);
```

Also, make sure the lifetime of the delegate instance covers the lifetime of the unmanaged code; otherwise, the delegate will not be available after it is garbage-collected.

See Also

C# Tutorials

COM Interop Tutorials

This section introduces two tutorials on interoperability between managed code and COM components.

COM Interop provides backward compatibility to let you access existing COM components without requiring that the original component be modified. The following tutorial demonstrates backward compatibility:

- COM Interop Part 1: C# Client Tutorial

COM Interop also allows COM developers to access managed code as easily as they access other COM objects. The following tutorial demonstrates accessing a C# server from a COM client:

- COM Interop Part 2: C# Server Tutorial

See Also

C# Tutorials

COM Interop Part 1: C# Client Tutorial

COM Interop provides access to existing COM components without requiring that the original component be modified. When you want to incorporate COM code into a managed application, import the relevant COM types by using a COM Interop utility (TlbImp.exe) for that purpose. Once imported, the COM types are ready to use.

In addition, COM Interop allows COM developers to access managed objects as easily as they access other COM objects. Again, COM Interop provides a specialized utility (RegAsm.exe) that exports the managed types into a type library and registers the managed component as a traditional COM component.

At run time, the common language runtime marshals data between COM objects and managed objects as needed.

This tutorial shows how to use C# to interoperate with COM objects.

COM Interop Part 2: C# Server Tutorial covers using a C# server with a C++ COM client. For an overview of both tutorials, see COM Interop Tutorials.

Sample Files

See COM Interop Part 1 Sample to download and build the sample files discussed in this tutorial.

Further Reading

- C# Attributes
- 17. Attributes
- Advanced COM Interop
- System.Runtime.InteropServices Namespace
- Type Library to Assembly Conversion Summary
- Type Library Importer (Tlbimp.exe)
- MSIL Disassembler (Ildasm.exe)
- Assembly Registration Tool (Regasm.exe)
- Interop Marshaling
- COM Interop Part 2: C# Server Tutorial

Tutorial

C# uses .NET Framework facilities to perform COM Interop. C# has support for:

- Creating COM objects.
- Determining if a COM interface is implemented by an object.
- Calling methods on COM interfaces.
- Implementing objects and interfaces that can be called by COM clients.

The .NET Framework handles reference-counting issues with COM Interop so there is no need to call or implement **AddRef** and **Release**.

This tutorial covers the following topics:

- Creating a COM Class Wrapper
- Declaring a COM coclass
- Creating a COM Object
- Declaring a COM Interface
- Using Casts Instead of QueryInterface
- Putting It All Together

Creating a COM Class Wrapper

For C# code to reference COM objects and interfaces, you need to include a .NET Framework definition for the COM interfaces in your C# build. The easiest way to do this is to use TlbImp.exe (Type Library Importer), a command-line tool included in the .NET Framework SDK. TlbImp converts a COM type library into .NET Framework metadata — effectively creating a managed wrapper that can be called from any managed language. .NET Framework metadata created with TlbImp can be included in a C# build via

the /R compiler option. If you are using the Visual Studio development environment, you only need to add a reference to the COM type library and the conversion is done for you automatically.

TlbImp performs the following conversions:

- COM coclasses are converted to C# classes with a parameterless constructor.
- COM structs are converted to C# structs with public fields.

A great way to check the output of TlbImp is to run the .NET Framework SDK command-line tool Ildasm.exe (Microsoft Intermediate Language Disassembler) to view the result of the conversion.

Although TlbImp is the preferred method for converting COM definitions to C#, it is not always possible to use it (for example, if there is no typelib for the COM definitions, or if TlbImp cannot handle the definitions in the typelib). In these cases, the alternative is to manually define the COM definitions in C# source code using C# attributes. Once you have created the C# source mapping, you simply compile the C# source code to produce the managed wrapper.

The main attributes you need to understand to perform COM mapping are:

- **ComImport** - Marks a class as an externally implemented COM class.
- **Guid** – Used to specify a universally unique identifier (UUID) for a class or an interface.
- **InterfaceType** – specifies whether an interface derives from **IUnknown** or **IDispatch**.
- **PreserveSig** – specifies whether the native return value should be converted from an HRESULT to a .NET Framework exception.

Each of these attributes is shown in the context of an actual example in this tutorial.

Declaring a COM coclass

COM coclasses are represented in C# as classes. These classes must have the **ComImport** attribute associated with them. The following restrictions apply to these classes:

- The class must not inherit from any other class.
- The class must implement no interfaces.
- The class must also have a **Guid** attribute that sets the globally unique identifier (GUID) for the class.

The following example declares a coclass in C#:

```
//
// declare FilgraphManager as a COM coclass
//
[ComImport, Guid("E436EBB3-524F-11CE-9F53-0020AF0BA770")]
class FilgraphManager
{
}
```

The C# compiler will add a parameterless constructor that you can call to create an instance of the COM coclass.

Creating a COM Object

COM coclasses are represented in C# as classes with a parameterless constructor. Creating an instance of this class using the **new** operator is the C# equivalent of calling **CoCreateInstance**. Using the class defined above, it is simple to instantiate the class:

```
class MainClass
{
    public static void Main()
    {
        //
        // Create an instance of a COM coclass - calls
        //
        // CoCreateInstance(E436EBB3-524F-11CE-9F53-0020AF0BA770,
        //                  NULL, CLSCTX_ALL,
        //                  IID_IUnknown, &f)
        //
        // returns null on failure.
        //
```

```
        FilgraphManager f = new FilgraphManager();
    }
}
```

Declaring a COM Interface

COM interfaces are represented in C# as interfaces with **ComImport** and **Guid** attributes. They cannot include any interfaces in their base interface list, and they must declare the interface member functions in the order that the methods appear in the COM interface.

COM interfaces declared in C# must include declarations for all members of their base interfaces with the exception of members of **IUnknown** and **IDispatch** — the .NET Framework automatically adds these. COM interfaces which derive from **IDispatch** must be marked with the **InterfaceType** attribute.

When calling a COM interface method from C# code, the common language runtime must marshal the parameters and return values to/from the COM object. For every .NET Framework type, there is a default type that the common language runtime will use to marshal when marshaling across a COM call. For example, the default marshaling for C# string values is to the native type LPTSTR (pointer to TCHAR char buffer). You can override the default marshaling using the **MarshalAs** attribute in the C# declaration of the COM interface.

In COM, a common way to return success or failure is to return an HRESULT and have an out parameter marked as "retval" in MIDL for the real return value of the method. In C# (and the .NET Framework), the standard way to indicate an error has occurred is to throw an exception.

By default, the .NET Framework provides an automatic mapping between the two styles of exception handling for COM interface methods called by the .NET Framework.

- The return value changes to the signature of the parameter marked retval (void if the method has no parameter marked as retval).
- The parameter marked as retval is left off of the argument list of the method.

Any non-success return value will cause a **System.COMException** exception to be thrown.

This example shows a COM interface declared in MIDL and the same interface declared in C# (note that the methods use the COM error-handling approach).

Here is the original MIDL version of the interface:

```
[
  odl,
  uuid(56A868B1-0AD4-11CE-B03A-0020AF0BA770),
  helpstring("IMediaControl interface"),
  dual,
  oleautomation
]
interface IMediaControl : IDispatch
{
  [id(0x60020000)]
  HRESULT Run();

  [id(0x60020001)]
  HRESULT Pause();

  [id(0x60020002)]
  HRESULT Stop();

  [id(0x60020003)]
  HRESULT GetState( [in] long msTimeout, [out] long* pfs);

  [id(0x60020004)]
  HRESULT RenderFile([in] BSTR strFilename);

  [id(0x60020005)]
  HRESULT AddSourceFilter( [in] BSTR strFilename, [out] IDispatch** ppUnk);

  [id(0x60020006), propget]
  HRESULT FilterCollection([out, retval] IDispatch** ppUnk);
```

```
  [id(0x60020007), propget]
  HRESULT RegFilterCollection([out, retval] IDispatch** ppUnk);

  [id(0x60020008)]
  HRESULT StopWhenReady();
};
```

Here is the C# equivalent of this interface:

```
using System.Runtime.InteropServices;

// Declare IMediaControl as a COM interface which
// derives from the IDispatch interface.
[Guid("56A868B1-0AD4-11CE-B03A-0020AF0BA770"),
    InterfaceType(ComInterfaceType.InterfaceIsDual)]
interface IMediaControl // cannot list any base interfaces here
{
    // Note that the members of IUnknown and Interface are NOT
    // listed here
    //
    void Run();

    void Pause();

    void Stop();

    void GetState( [In] int msTimeout, [Out] out int pfs);

    void RenderFile(
    [In, MarshalAs(UnmanagedType.BStr)] string strFilename);

    void AddSourceFilter(
    [In, MarshalAs(UnmanagedType.BStr)] string strFilename,
    [Out, MarshalAs(UnmanagedType.Interface)] out object ppUnk);

    [return : MarshalAs(UnmanagedType.Interface)]
    object FilterCollection();

    [return : MarshalAs(UnmanagedType.Interface)]
    object RegFilterCollection();

    void StopWhenReady();
}
```

Note how the C# interface has mapped the error-handling cases. If the COM method returns an error, an exception will be raised on the C# side.

To prevent the translation of HRESULTs to COMExceptions, attach the **PreserveSig(true)** attribute to the method in the C# declaration. For details, see PreserveSigAttribute Class.

Using Casts Instead of QueryInterface

A C# coclass is not very useful until you can access an interface that it implements. In C++ you would navigate an object's interfaces using the **QueryInterface** method on the **IUnknown** interface. In C# you can do the same thing by explicitly casting the COM object to the desired COM interface. If the cast fails, then an invalid cast exception is thrown:

```
// Create an instance of a COM coclass:
FilgraphManager graphManager = new FilgraphManager();

// See if it supports the IMediaControl COM interface.
// Note that this will throw a System.InvalidCastException if
// the cast fails. This is equivalent to QueryInterface for
// COM objects:
IMediaControl mc = (IMediaControl) graphManager;
```

```
// Now you call a method on a COM interface:
mc.Run();
```

Putting It All Together

Here is a complete example that creates an AVI file viewer using C#. The program creates an instance of a COM coclass, casts it to a COM interface, and then calls methods on the COM interface.

The examples in this section represent two approaches:

- Example 1 Using TlbImp to create the .NET Framework class.
- Example 2 Writing C# code that manually does the COM mapping.

Example 1: Using TlbImp

This example shows you how to create an AVI viewer using TlbImp. The program reads an AVI filename from the command line, creates an instance of the Quartz COM object, then uses the methods **RenderFile** and **Run** to display the AVI file.

These are the steps to build the program:

- Run TlbImp over the TLB. The Media Player used in this example is contained in Quartz.dll, which should be in your Windows system directory. Use the following command to create the .NET Framework DLL:

  ```
  tlbimp c:\winnt\system32\quartz.dll /out:QuartzTypeLib.dll
  ```

  Note that the resulting DLL needs to be named `QuartzTypeLib`, so the .NET Framework can load the containing types correctly when running.

- You can use the Ildasm tool to examine the resulting DLL. For example, to display the contents of the file `QuartzTypeLib.dll`, use the following command:

  ```
  Ildasm QuartzTypeLib.dll
  ```

- Build the program using the C# compiler option /R to include the `QuartzTypeLib.dll` file.

You can then use the program to display a movie (an example movie to try is Clock.avi, which resides in your Windows directory).

```
// interop1.cs
// compile with: /R:QuartzTypeLib.dll
using System;
class MainClass
{
      /*************************************************************
      Abstract: This method collects the file name of an AVI to
      show then creates an instance of the Quartz COM object.
      To show the AVI, the program calls RenderFile and Run on
      IMediaControl. Quartz uses its own thread and window to
      display the AVI.The main thread blocks on a ReadLine until
      the user presses ENTER.
            Input Parameters: the location of the AVI file it is
            going to display
            Returns: void
      *************************************************************/
      public static void Main(string[] args)
      {
            // Check to see if the user passed in a filename
            if (args.Length != 1)
            {
                  DisplayUsage();
                  return;
            }

            if (args[0] == "/?")
            {
                  DisplayUsage();
```

```
                return;
            }

            string filename = args[0];

            // Check to see if the file exists
            if (!System.IO.File.Exists(filename))
            {
                Console.WriteLine("File " + filename + " not found.");
                DisplayUsage();
                return;
            }

            // Create instance of Quartz
            // (Calls CoCreateInstance(E436EBB3-524F-11CE-9F53-0020AF0BA770,
            // NULL, CLSCTX_ALL, IID_IUnknown, &graphManager).):

            try
            {
                QuartzTypeLib.FilgraphManager graphManager =
                    new QuartzTypeLib.FilgraphManager();

                // QueryInterface for the IMediaControl interface:
                QuartzTypeLib.IMediaControl mc =
                    (QuartzTypeLib.IMediaControl)graphManager;

                // Call some methods on a COM interface
                // Pass in file to RenderFile method on COM object.
                mc.RenderFile(filename);

                // Show file.
                mc.Run();
            }
            catch(Exception ex)
            {
                Console.WriteLine("Unexpected COM exception: " + ex.Message);
            }

            // Wait for completion.
            Console.WriteLine("Press Enter to continue.");
            Console.ReadLine();
        }

        private static void DisplayUsage()
        {
            // User did not provide enough parameters.
            // Display usage:
            Console.WriteLine("VideoPlayer: Plays AVI files.");
            Console.WriteLine("Usage: VIDEOPLAYER.EXE filename");
            Console.WriteLine("where filename is the full path and");
            Console.WriteLine("file name of the AVI to display.");
        }
    }
```

Sample Run

To display the example movie, Clock.avi, use the following command:

```
interop1 %windir%\clock.avi
```

This will display the movie on your screen after you press ENTER.

Example 2: The C# Code Approach

This example uses the same **Main** method as Example 1, but instead of running TlbImp, it simply maps the Media Player COM

object using C#.

```
// interop2.cs
using System;
using System.Runtime.InteropServices;

namespace QuartzTypeLib
{
      // Declare IMediaControl as a COM interface which
      // derives from IDispatch interface:
      [Guid("56A868B1-0AD4-11CE-B03A-0020AF0BA770"),
      InterfaceType(ComInterfaceType.InterfaceIsDual)]
      interface IMediaControl   // Cannot list any base interfaces here
      {
            // Note that IUnknown Interface members are NOT listed here:


            void Run();
            void Pause();

            void Stop();

            void GetState( [In] int msTimeout, [Out] out int pfs);

            void RenderFile(
                  [In, MarshalAs(UnmanagedType.BStr)] string strFilename);

            void AddSourceFilter(
                  [In, MarshalAs(UnmanagedType.BStr)] string strFilename,
                  [Out, MarshalAs(UnmanagedType.Interface)]
                  out object ppUnk);

            [return: MarshalAs(UnmanagedType.Interface)]
            object FilterCollection();

            [return: MarshalAs(UnmanagedType.Interface)]
            object RegFilterCollection();

            void StopWhenReady();
      }
      // Declare FilgraphManager as a COM coclass:
      [ComImport, Guid("E436EBB3-524F-11CE-9F53-0020AF0BA770")]
      class FilgraphManager   // Cannot have a base class or
            // interface list here.
      {
            // Cannot have any members here
            // NOTE that the C# compiler will add a default constructor
            // for you (no parameters).
      }
}

class MainClass
{
      /*************************************************************
      Abstract: This method collects the file name of an AVI to
      show then creates an instance of the Quartz COM object.
      To show the AVI, the program calls RenderFile and Run on
      IMediaControl. Quartz uses its own thread and window to
      display the AVI.The main thread blocks on a ReadLine until
      the user presses ENTER.
            Input Parameters: the location of the AVI file it is
            going to display
            Returns: void
      *************************************************************/

      public static void Main(string[] args)
      {
            // Check to see if the user passed in a filename:
```

```
            if (args.Length != 1)
            {
                  DisplayUsage();
                  return;
            }

            if (args[0] == "/?")
            {
                  DisplayUsage();
                  return;
            }

            String filename = args[0];

            // Check to see if the file exists
            if (!System.IO.File.Exists(filename))
            {
                  Console.WriteLine("File " + filename + " not found.");
                  DisplayUsage();
                  return;
            }

            // Create instance of Quartz
            // (Calls CoCreateInstance(E436EBB3-524F-11CE-9F53-0020AF0BA770,
            //  NULL, CLSCTX_ALL, IID_IUnknown,
            //  &graphManager).):
            try
            {
                  QuartzTypeLib.FilgraphManager graphManager =
                        new QuartzTypeLib.FilgraphManager();

                  // QueryInterface for the IMediaControl interface:
                  QuartzTypeLib.IMediaControl mc =
                        (QuartzTypeLib.IMediaControl)graphManager;

                  // Call some methods on a COM interface.
                  // Pass in file to RenderFile method on COM object.
                  mc.RenderFile(filename);

                  // Show file.
                  mc.Run();
            }
            catch(Exception ex)
            {
                  Console.WriteLine("Unexpected COM exception: " + ex.Message);
            }
            // Wait for completion.
            Console.WriteLine("Press Enter to continue.");
            Console.ReadLine();
      }

      private static void DisplayUsage()
      {
            // User did not provide enough parameters.
            // Display usage.
            Console.WriteLine("VideoPlayer: Plays AVI files.");
            Console.WriteLine("Usage: VIDEOPLAYER.EXE filename");
            Console.WriteLine("where filename is the full path and");
            Console.WriteLine("file name of the AVI to display.");
      }
}
```

Sample Run

To display the example movie, Clock.avi, use the following command:

```
interop2 %windir%\clock.avi
```

This will display the movie on your screen after you press ENTER.

See Also

C# Tutorials | COM Interop Tutorials

COM Interop Part 2: C# Server Tutorial

COM Interop allows COM developers to access managed code as easily as they access other COM objects. This tutorial demonstrates using a C# server with a C++ COM client. It also explains the following activities:

- How to create the C# server
- How to create the COM client

The tutorial also briefly demonstrates the marshaling that is automatically applied between managed and unmanaged components.

COM Interop Part 1: C# Client Tutorial shows the fundamentals of using C# to interoperate with COM objects and is a prerequisite for this tutorial. For an overview of both tutorials, see COM Interop Tutorials.

Sample Files

See COM Interop Part 2 Sample to download and build the sample files discussed in this tutorial.

Further Reading

- C# Attributes
- 17. Attributes
- Advanced COM Interop
- System.Runtime.InteropServices Namespace
- Type Library to Assembly Conversion Summary
- Assembly Registration Tool (Regasm.exe)
- Type Library Importer (Tlbimp.exe)
- Interop Marshaling
- COM Interop Part 1: C# Client Tutorial

Tutorial

This tutorial demonstrates the following activities to create the C# server:

- How to use the **Guid** attribute on interfaces and classes to expose them as COM objects and how to generate a globally unique identifier (GUID) for the **Guid** attribute.
- How to use RegAsm to register a .NET Framework program for use by COM clients and create a type library (.tlb file) from a .NET Framework program.

The tutorial also demonstrates the following activities to create the COM client:

- How to export managed servers and how to use them to create COM objects.
- How to import the .tlb file, generated by RegAsm, into the COM client and how to use **CoCreateInstance** to create an instance of a .NET Framework coclass.

> **Note** To create a GUID for interfaces and coclasses that you export to COM clients, use the tool Guidgen.exe, shipped as part of Visual Studio. Guidgen allows you to choose the format in which the GUID is expressed so you don't have to retype it. For more information on Guidgen, see the Knowledge Base article Q168318 "XADM: Guidgen.exe Available Only for Intel Platforms." KB articles are available in the MSDN Library and on the Web at http://support.microsoft.com.

Example

This example consists of two files:

- A C# file, CSharpServer.cs, that creates the CSharpServer.dll file. The .dll is used to create the file CSharpServer.tlb.
- A C++ file, COMClient.cpp, that creates the executable client, COMClient.exe.

File 1: CSharpServer.cs

```
// CSharpServer.cs
// compile with: /target:library
// post-build command: regasm CSharpServer.dll /tlb:CSharpServer.tlb

using System;
using System.Runtime.InteropServices;
namespace CSharpServer
{
   // Since the .NET Framework interface and coclass have to behave as
   // COM objects, we have to give them guids.
   [Guid("DBE0E8C4-1C61-41f3-B6A4-4E2F353D3D05")]
   public interface IManagedInterface
   {
      int PrintHi(string name);
   }

   [Guid("C6659361-1625-4746-931C-36014B146679")]
   public class InterfaceImplementation : IManagedInterface
   {
      public int PrintHi(string name)
      {
         Console.WriteLine("Hello, {0}!", name);
         return 33;
      }
   }
}
```

File 2: COMClient.cpp

```
// COMClient.cpp
// Build with "cl COMClient.cpp"
// arguments: friend

#include <windows.h>
#include <stdio.h>

#pragma warning (disable: 4278)

// To use managed-code servers like the C# server,
// we have to import the common language runtime:
#import <mscorlib.tlb> raw_interfaces_only

// For simplicity, we ignore the server namespace and use named guids:
#if defined (USINGPROJECTSYSTEM)
#import "..\RegisterCSharpServerAndExportTLB\CSharpServer.tlb" no_namespace named_guids
#else  // Compiling from the command line, all files in the same directory
#import "CSharpServer.tlb" no_namespace named_guids
#endif
int main(int argc, char* argv[])
{
   IManagedInterface *cpi = NULL;
   int retval = 1;

   // Initialize COM and create an instance of the InterfaceImplementation class:
   CoInitialize(NULL);
   HRESULT hr = CoCreateInstance(CLSID_InterfaceImplementation,
               NULL, CLSCTX_INPROC_SERVER,
               IID_IManagedInterface, reinterpret_cast<void**>(&cpi));

   if (FAILED(hr))
   {
      printf("Couldn't create the instance!... 0x%x\n", hr);
   }
   else
   {
      if (argc > 1)
```

```
        {
           printf("Calling function.\n");
           fflush(stdout);
           // The variable cpi now holds an interface pointer
           // to the managed interface.
           // If you are on an OS that uses ASCII characters at the
           // command prompt, notice that the ASCII characters are
           // automatically marshaled to Unicode for the C# code.
           if (cpi->PrintHi(argv[1]) == 33)
              retval = 0;
           printf("Returned from function.\n");
        }
        else
           printf ("Usage:  COMClient <name>\n");
        cpi->Release();
        cpi = NULL;
     }

     // Be a good citizen and clean up COM:
     CoUninitialize();
     return retval;
  }
```

Output

The executable client can be invoked with the command line: `COMClient <name>`, where `<name>` is any string you want to use, for example, `COMClient friend`.

```
Calling function.
Hello, friend!
Returned from function.
```

In the sample IDE project, set the **Command Line Arguments** property in the project's Property Pages to the desired string (for example, "friend").

See Also

C# Tutorials | COM Interop Tutorials

Attributes Tutorial

This tutorial shows how to create custom attribute classes, use them in code, and query them through reflection.

Sample Files

See Attributes Sample to download and build the sample files discussed in this tutorial.

Further Reading

- C# Attributes
- 17. Attributes
- AttributeUsage
- Discovering Type Information at Runtime

Tutorial

Attributes provide a powerful method of associating declarative information with C# code (types, methods, properties, and so forth). Once associated with a program entity, the attribute can be queried at run time and used in any number of ways.

Example usage of attributes includes:

- Associating help documentation with program entities (through a `Help` attribute).
- Associating value editors to a specific type in a GUI framework (through a `ValueEditor` attribute).

In addition to a complete example, this tutorial includes the following topics:

- Declaring an Attribute Class

  The first thing you need to be able to do is declare an attribute.

- Using an Attribute Class

  Once the attribute has been created, you then associate the attribute with a particular program element.

- Accessing Attributes Through Reflection

  Once the attribute has been associated with a program element, you use reflection to query its existence and its value.

Declaring an Attribute Class

Declaring an attribute in C# is simple — it takes the form of a class declaration that inherits from **System.Attribute** and has been marked with the **AttributeUsage** attribute as shown below:

```
using System;
[AttributeUsage(AttributeTargets.All)]
public class HelpAttribute : System.Attribute
{
   public readonly string Url;

   public string Topic               // Topic is a named parameter
   {
      get
      {
         return topic;
      }
      set
      {

        topic = value;
      }
   }

   public HelpAttribute(string url)  // url is a positional parameter
   {
      this.Url = url;
```

```
    }

    private string topic;
  }
```

Code Discussion

- The attribute **AttributeUsage** specifies the language elements to which the attribute can be applied.
- Attributes classes are public classes derived from **System.Attribute** that have at least one public constructor.
- Attribute classes have two types of parameters:
  - *Positional parameters* must be specified every time the attribute is used. Positional parameters are specified as constructor arguments to the attribute class. In the example above, `url` is a positional parameter.
  - *Named parameters* are optional. If they are specified when the attribute is used, the name of the parameter must be used. Named parameters are defined by having a nonstatic field or property. In the example above, `Topic` is a named parameter.
- Attribute parameters are restricted to constant values of the following types:
  - Simple types (bool, byte, char, short, int, long, float, and double)
  - string
  - System.Type
  - enums
  - object (The argument to an attribute parameter of type object must be a constant value of one of the above types.)
  - One-dimensional arrays of any of the above types

Parameters for the AttributeUsage Attribute

The attribute **AttributeUsage** provides the underlying mechanism by which attributes are declared.

AttributeUsage has one positional parameter:

- **AllowOn**, which specifies the program elements that the attribute can be assigned to (class, method, property, parameter, and so on). Valid values for this parameter can be found in the **System.Attributes.AttributeTargets** enumeration in the .NET Framework. The default value for this parameter is all program elements (**AttributeElements.All**).

AttributeUsage has one named parameter:

- **AllowMultiple**, a Boolean value that indicates whether multiple attributes can be specified for one program element. The default value for this parameter is **False**.

Using an Attribute Class

Here's a simple example of using the attribute declared in the previous section:

```
[HelpAttribute("http://localhost/MyClassInfo")]
class MyClass
{
}
```

In this example, the `HelpAttribute` attribute is associated with `MyClass`.

> **Note** By convention, all attribute names end with the word "Attribute" to distinguish them from other items in the .NET Framework. However, you do not need to specify the attribute suffix when using attributes in code. For example, you can specify `HelpAttribute` as follows:
>
> ```
> [Help("http://localhost/MyClassInfo")] // [Help] == [HelpAttribute]
>
> class MyClass
>
> {
>
> }
> ```

Accessing Attributes Through Reflection

Once attributes have been associated with program elements, reflection can be used to query their existence and values. The main reflection methods to query attributes are contained in the **System.Reflection.MemberInfo** class (**GetCustomAttributes** family of methods). The following example demonstrates the basic way of using reflection to get access to attributes:

```
class MainClass
{
   public static void Main()
   {
      System.Reflection.MemberInfo info = typeof(MyClass);
      object[] attributes = info.GetCustomAttributes(true);
      for (int i = 0; i < attributes.Length; i ++)
      {
         System.Console.WriteLine(attributes[i]);
      }
   }
}
```

Example

The following is a complete example where all pieces are brought together.

```
// AttributesTutorial.cs
// This example shows the use of class and method attributes.

using System;
using System.Reflection;
using System.Collections;

// The IsTested class is a user-defined custom attribute class.
// It can be applied to any declaration including
//  - types (struct, class, enum, delegate)
//  - members (methods, fields, events, properties, indexers)
// It is used with no arguments.
public class IsTestedAttribute : Attribute
{
    public override string ToString()
    {
        return "Is Tested";
    }
}

// The AuthorAttribute class is a user-defined attribute class.
// It can be applied to classes and struct declarations only.
// It takes one unnamed string argument (the author's name).
// It has one optional named argument Version, which is of type int.
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Struct)]
public class AuthorAttribute : Attribute
{
    // This constructor specifies the unnamed arguments to the attribute class.
    public AuthorAttribute(string name)
    {
        this.name = name;
        this.version = 0;
    }

    // This property is readonly (it has no set accessor)
    // so it cannot be used as a named argument to this attribute.
    public string Name
    {
        get
        {
            return name;
        }
    }

    // This property is read-write (it has a set accessor)
```

```
    // so it can be used as a named argument when using this
    // class as an attribute class.
    public int Version
    {
        get
        {
            return version;
        }
        set
        {
            version = value;
        }
    }

    public override string ToString()
    {
        string value = "Author : " + Name;
        if (version != 0)
        {
            value += " Version : " + Version.ToString();
        }
        return value;
    }

    private string name;
    private int version;
}

// Here you attach the AuthorAttribute user-defined custom attribute to
// the Account class. The unnamed string argument is passed to the
// AuthorAttribute class's constructor when creating the attributes.
[Author("Joe Programmer")]
class Account
{
    // Attach the IsTestedAttribute custom attribute to this method.
    [IsTested]
    public void AddOrder(Order orderToAdd)
    {
        orders.Add(orderToAdd);
    }

    private ArrayList orders = new ArrayList();
}

// Attach the AuthorAttribute and IsTestedAttribute custom attributes
// to this class.
// Note the use of the 'Version' named argument to the AuthorAttribute.
[Author("Jane Programmer", Version = 2), IsTested()]
class Order
{
    // add stuff here ...
}

class MainClass
{
   private static bool IsMemberTested(MemberInfo member)
   {
        foreach (object attribute in member.GetCustomAttributes(true))
        {
            if (attribute is IsTestedAttribute)
            {
               return true;
            }
        }
      return false;
   }

    private static void DumpAttributes(MemberInfo member)
```

```
    {
        Console.WriteLine("Attributes for : " + member.Name);
        foreach (object attribute in member.GetCustomAttributes(true))
        {
            Console.WriteLine(attribute);
        }
    }

    public static void Main()
    {
        // display attributes for Account class
        DumpAttributes(typeof(Account));

        // display list of tested members
        foreach (MethodInfo method in (typeof(Account)).GetMethods())
        {
            if (IsMemberTested(method))
            {
               Console.WriteLine("Member {0} is tested!", method.Name);
            }
            else
            {
               Console.WriteLine("Member {0} is NOT tested!", method.Name);
            }
        }
        Console.WriteLine();

        // display attributes for Order class
        DumpAttributes(typeof(Order));

        // display attributes for methods on the Order class
        foreach (MethodInfo method in (typeof(Order)).GetMethods())
        {
           if (IsMemberTested(method))
           {
               Console.WriteLine("Member {0} is tested!", method.Name);
           }
           else
           {
               Console.WriteLine("Member {0} is NOT tested!", method.Name);
           }
        }
        Console.WriteLine();
    }
}
```

Output

```
Attributes for : Account
Author : Joe Programmer
Member GetHashCode is NOT tested!
Member Equals is NOT tested!
Member ToString is NOT tested!
Member AddOrder is tested!
Member GetType is NOT tested!

Attributes for : Order
Author : Jane Programmer Version : 2
Is Tested
Member GetHashCode is NOT tested!
Member Equals is NOT tested!
Member ToString is NOT tested!
Member GetType is NOT tested!
```

See Also

Security Tutorial

This tutorial discusses .NET Framework security and shows two ways to modify security permissions in C#: imperative and declarative security.

Sample Files

See Security Sample to download and build the sample files discussed in this tutorial.

Further Reading

- Securing Applications
- Permissions
- Security Policy
- Role-Based Security Checks
- Security Demands
- Demands

Tutorial

Most application and component developers should not need to do anything special in order to work with the .NET Framework security system and benefit from the safety protection it provides.

One exception that requires more in-depth knowledge and special consideration of the security system is *secure libraries*. This code represents the boundary between secure managed code and unrestricted code, such as native code (that is outside the ability of the .NET Framework security infrastructure to enforce). These libraries typically must be highly trusted to work, and are the one place in managed code where a programming error can potentially expose a security vulnerability. Code access security can't eliminate the potential for human error, but compared to the much larger volume of application code that uses a few secure libraries, the amount of code that requires close scrutiny is dramatically reduced.

Examples

The tutorial includes the following examples:

- Example 1: Imperative Security
- Example 2: Declarative Security
- Example 3: Suppressing Security

Security

The .NET Framework security protects your code and data from being misused or damaged by other code by enforcing security restrictions on managed code. When a .NET Framework application requests permission, the security policy established by the administrator grants the permission or refuses to run the code. Trust is based on evidence about the code such as a digital signature, where the code comes from, and so forth. Once granted, security enforces permissions that control what code is (and by not being granted, what code is not) allowed to do.

Permissions

The .NET Framework security allows code to use protected resources only if it has "permission" to do so. To express this, the .NET Framework uses the concept of *permissions*, which represent the right for code to access protected resources. Code requests the permissions it needs, and the security policy applied by the .NET Framework determines which permissions the code is actually granted.

The .NET Framework provides code access permission classes, each of which encapsulates the ability to access a particular resource. You use these permissions to indicate to the .NET Framework what your code needs to be allowed to do and to indicate what your code's callers must be authorized to do. Policy also uses these objects to determine what permissions to grant to code.

Policy

Enforcement of security policy is what makes .NET Framework managed code safe. Every assembly that loads is subject to security policy that grants code permissions based on trust, with trust based on evidence about the code. See the .NET Framework documentation in the Reading List for information on administering security policy.

Demanding Security Permissions in C#

There are two ways to demand security permissions in C#:

- Imperatively: Using calls to permission classes in the .NET Framework
- Declaratively: Using security permission attributes

The following two examples demonstrate these two approaches. For more information on demanding security permissions, see Demands.

Example 1: Imperative Security

The following is an example of using .NET Framework calls to deny the **UnmanagedCode** permission.

```
// ImperativeSecurity.cs
using System;
using System.Security;
using System.Security.Permissions;
using System.Runtime.InteropServices;

class NativeMethods
{
    // This is a call to unmanaged code. Executing this method requires
    // the UnmanagedCode security permission. Without this permission
    // an attempt to call this method will throw a SecurityException:
    [DllImport("msvcrt.dll")]
    public static extern int puts(string str);
    [DllImport("msvcrt.dll")]
    internal static extern int _flushall();
}

class MainClass
{
    private static void CallUnmanagedCodeWithoutPermission()
    {
        // Create a security permission object to describe the
        // UnmanagedCode permission:
        SecurityPermission perm =
           new SecurityPermission(SecurityPermissionFlag.UnmanagedCode);

        // Deny the UnmanagedCode from our current set of permissions.
        // Any method that is called on this thread until this method
        // returns will be denied access to unmanaged code.
        // Even though the CallUnmanagedCodeWithPermission method
        // is called from a stack frame that already
        // calls Assert for unmanaged code, you still cannot call native
        // code. Because you use Deny here, the permission gets
        // overwritten.
        perm.Deny();

        try
        {
            Console.WriteLine("Attempting to call unmanaged code without permission.");
            NativeMethods.puts("Hello World!");
            NativeMethods._flushall();
            Console.WriteLine("Called unmanaged code without permission. Whoops!");
        }
        catch (SecurityException)
        {
            Console.WriteLine("Caught Security Exception attempting to call unmanaged code.")
;
        }
    }

    private static void CallUnmanagedCodeWithPermission()
    {
        // Create a security permission object to describe the
```

```
        // UnmanagedCode permission:
        SecurityPermission perm =
           new SecurityPermission(SecurityPermissionFlag.UnmanagedCode);

        // Check that you have permission to access unmanaged code.
        // If you don't have permission to access unmanaged code, then
        // this call will throw a SecurityException.
        // Even though the CallUnmanagedCodeWithPermission method
        // is called from a stack frame that already
        // calls Assert for unmanaged code, you still cannot call native
        // code. Because you use Deny here, the permission gets
        // overwritten.
        perm.Assert();

        try
        {
            Console.WriteLine("Attempting to call unmanaged code with permission.");
            NativeMethods.puts("Hello World!");
            NativeMethods._flushall();
            Console.WriteLine("Called unmanaged code with permission.");
        }
        catch (SecurityException)
        {
            Console.WriteLine("Caught Security Exception attempting to call unmanaged code. W
hoops!");
        }
    }

    public static void Main()
    {
        // The method itself will call the security permission Deny
        // for unmanaged code, which will override the Assert permission
        // in this stack frame.
        SecurityPermission perm = new
            SecurityPermission(SecurityPermissionFlag.UnmanagedCode);
        perm.Assert();
        CallUnmanagedCodeWithoutPermission();

        // The method itself will call the security permission Assert
        // for unmanaged code, which will override the Deny permission in
        // this stack frame.
        perm.Deny();
        CallUnmanagedCodeWithPermission();
    }
}
```

Output

```
Attempting to call unmanaged code without permission.
Caught Security Exception attempting to call unmanaged code.
Attempting to call unmanaged code with permission.
Hello World!
Called unmanaged code with permission.
```

Example 2: Declarative Security

This is the same example using attributes for the security permissions.

```
// DeclarativeSecurity.cs
using System;
using System.Security;
using System.Security.Permissions;
using System.Runtime.InteropServices;

class NativeMethods
```

```
{
    // This is a call to unmanaged code. Executing this method requires
    // the UnmanagedCode security permission. Without this permission,
    // an attempt to call this method will throw a SecurityException:
    [DllImport("msvcrt.dll")]
    public static extern int puts(string str);
    [DllImport("msvcrt.dll")]
    internal static extern int _flushall();
}

class MainClass
{
    // The security permission attached to this method will deny the
    // UnmanagedCode permission from the current set of permissions for
    // the duration of the call to this method:
    // Even though the CallUnmanagedCodeWithoutPermission method is
    // called from a stack frame that already calls
    // Assert for unmanaged code, you still cannot call native code.
    // Because this function is attached with the Deny permission for
    // unmanaged code, the permission gets overwritten.
    [SecurityPermission(SecurityAction.Deny, Flags =
       SecurityPermissionFlag.UnmanagedCode)]
    private static void CallUnmanagedCodeWithoutPermission()
    {
        try
        {
            Console.WriteLine("Attempting to call unmanaged code without permission.");
            NativeMethods.puts("Hello World!");
            NativeMethods._flushall();
            Console.WriteLine("Called unmanaged code without permission. Whoops!");
        }
        catch (SecurityException)
        {
            Console.WriteLine("Caught Security Exception attempting to call unmanaged code.")
;
        }
    }

    // The security permission attached to this method will force a
    // runtime check for the unmanaged code permission whenever
    // this method is called. If the caller does not have unmanaged code
    // permission, then the call will generate a Security Exception.
    // Even though the CallUnmanagedCodeWithPermission method is called
    // from a stack frame that already calls
    // Deny for unmanaged code, it will not prevent you from calling
    // native code. Because this method is attached with the Assert
    // permission for unmanaged code, the permission gets overwritten.
    [SecurityPermission(SecurityAction.Assert, Flags =
       SecurityPermissionFlag.UnmanagedCode)]
    private static void CallUnmanagedCodeWithPermission()
    {
        try
        {
            Console.WriteLine("Attempting to call unmanaged code with permission.");
            NativeMethods.puts("Hello World!");
            NativeMethods._flushall();
            Console.WriteLine("Called unmanaged code with permission.");
        }
        catch (SecurityException)
        {
            Console.WriteLine("Caught Security Exception attempting to call unmanaged code. W
hoops!");
        }
    }

    public static void Main()
    {
        SecurityPermission perm = new
```

```
            SecurityPermission(SecurityPermissionFlag.UnmanagedCode);

        // The method itself is attached with the security permission
        // Deny for unmanaged code, which will override
        // the Assert permission in this stack frame.
        perm.Assert();
        CallUnmanagedCodeWithoutPermission();

        // The method itself is attached with the security permission
        // Assert for unmanaged code, which will override the Deny
        // permission in this stack frame.
        perm.Deny();
        CallUnmanagedCodeWithPermission();
    }
}
```

Output

```
Attempting to call unmanaged code without permission.
Caught Security Exception attempting to call unmanaged code.
Attempting to call unmanaged code with permission.
Hello World!
Called unmanaged code with permission.
```

Security and Performance

The .NET Framework security system prevents malicious code downloaded from the network from damaging your computer system. However, these security checks are not without cost, even if your code never throws a **SecurityException**.

Normally the common language runtime verifies that the caller of an unmanaged method has unmanaged code access permission at run time for every call to the unmanaged method. This can be very expensive for applications that make many calls to unmanaged code. The **SuppressUnmanagedCodeSecurityAttribute** changes this default behavior. When an unmanaged method is declared with this attribute, the security demand is checked when code that calls the method is loaded into the common language runtime.

> **Security Note** When using the SuppressUnmanagedCodeSecurityAttribute, you should take extra care to ensure that you are not introducing a security hole. For example, the developer needs to verify that he/she is using the unmanaged API safely and that callers cannot influence or abuse the call. Alternatively, the developer can add an appropriate demand to ensure that all callers have the appropriate permissions. For example, if a call was made into native code to access a file (to take advantage of structured storage such as extended file properties, and so forth) and the unmanaged code demand was suppressed, then a file IO demand should be made explicitly to ensure that the code cannot be misused.

Example 3: Optimizing Unmanaged Calls

In this example, the check for the unmanaged code permission is executed once at load time, rather than upon every call to the unmanaged method. If the unmanaged method is called multiple times, this could yield a significant performance gain.

```
// SuppressSecurity.cs
using System;
using System.Security;
using System.Security.Permissions;
using System.Runtime.InteropServices;

class NativeMethods
{
    // This is a call to unmanaged code. Executing this method requires
    // the UnmanagedCode security permission. Without this permission,
    // an attempt to call this method will throw a SecurityException:
    /* NOTE: The SuppressUnmanagedCodeSecurityAttribute disables the
       check for the UnmanagedCode permission at runtime. Be Careful! */
    [SuppressUnmanagedCodeSecurityAttribute()]
    [DllImport("msvcrt.dll")]
```

```
    internal static extern int puts(string str);
    [SuppressUnmanagedCodeSecurityAttribute()]
    [DllImport("msvcrt.dll")]
    internal static extern int _flushall();
}

class MainClass
{
    // The security permission attached to this method will remove the
    // UnmanagedCode permission from the current set of permissions for
    // the duration of the call to this method.
    // Even though the CallUnmanagedCodeWithoutPermission method is
    // called from a stack frame that already calls
    // Assert for unmanaged code, you still cannot call native code.
    // Because this method is attached with the Deny permission for
    // unmanaged code, the permission gets overwritten. However, because
    // you are using SuppressUnmanagedCodeSecurityAttribute here, you can
    // still call the unmanaged methods successfully.
    // The code should use other security checks to ensure that you don't
    // incur a security hole.
    [SecurityPermission(SecurityAction.Deny, Flags =
       SecurityPermissionFlag.UnmanagedCode)]
    private static void CallUnmanagedCodeWithoutPermission()
    {
        try
        {
            // The UnmanagedCode security check is disbled on the call
            // below. However, the unmanaged call only displays UI. The
            // security will be ensured by only allowing the unmanaged
            // call if there is a UI permission.
            UIPermission uiPermission =
               new UIPermission(PermissionState.Unrestricted);
            uiPermission.Demand();

            Console.WriteLine("Attempting to call unmanaged code without UnmanagedCode permis
sion.");
            NativeMethods.puts("Hello World!");
            NativeMethods._flushall();
            Console.WriteLine("Called unmanaged code without UnmanagedCode permission.");
        }
        catch (SecurityException)
        {
            Console.WriteLine("Caught Security Exception attempting to call unmanaged code.")
;
        }
    }

    // The security permission attached to this method will add the
    // UnmanagedCode permission to the current set of permissions for the
    // duration of the call to this method.
    // Even though the CallUnmanagedCodeWithPermission method is called
    // from a stack frame that already calls
    // Deny for unmanaged code, it will not prevent you from calling
    // native code. Because this method is attached with the Assert
    // permission for unmanaged code, the permission gets overwritten.
    // Because you are using SuppressUnmanagedCodeSecurityAttribute here,
    // you can call the unmanaged methods successfully.
    // The SuppressUnmanagedCodeSecurityAttribute will let you succeed,
    // even if you don't have a permission.
    [SecurityPermission(SecurityAction.Assert, Flags =
       SecurityPermissionFlag.UnmanagedCode)]
    private static void CallUnmanagedCodeWithPermission()
    {
        try
        {
            Console.WriteLine("Attempting to call unmanaged code with permission.");
            NativeMethods.puts("Hello World!");
            NativeMethods._flushall();
```

```
            Console.WriteLine("Called unmanaged code with permission.");
        }
        catch (SecurityException)
        {
            Console.WriteLine("Caught Security Exception attempting to call unmanaged code. W
hoops!");
        }
    }

    public static void Main()
    {
        SecurityPermission perm = new
            SecurityPermission(SecurityPermissionFlag.UnmanagedCode);

       // The method itself is attached with the security permission Deny
       // for unmanaged code, which will override the Assert permission in
       // this stack frame. However, because you are using
       // SuppressUnmanagedCodeSecurityAttribute, you can still call the
       // unmanaged methods successfully.
       // The code should use other security checks to ensure that you
       // don't incur a security hole.
       perm.Assert();
       CallUnmanagedCodeWithoutPermission();

       // The method itself is attached with the security permission
       // Assert for unmanaged code, which will override the Deny
       // permission in this stack frame. Because you are using
       // SuppressUnmanagedCodeSecurityAttribute, you can call the
       // unmanaged methods successfully.
       // The SuppressUnmanagedCodeSecurityAttribute will let you succeed,
       // even if you don't have a permission.
       perm.Deny();
       CallUnmanagedCodeWithPermission();
    }
}
```

Output

```
Attempting to call unmanaged code without UnmanagedCode permission.
Hello World!
Called unmanaged code without UnmanagedCode permission.
Attempting to call unmanaged code with permission.
Hello World!
Called unmanaged code with permission.
```

Code Discussion

Note that the above example allows both unmanaged calls to succeed even though the first call doesn't have **UnmanagedCode** permission. When using the **SuppressUnmanagedCodeSecurityAttribute**, you should use other security checks to ensure that you don't incur a security hole. In the above example, this is done by adding the **Demand** for the **UIPermission**:

```
uiPermission.Demand();
```

before the unmanaged call, which ensures that the caller has permission to display UI.

See Also

C# Tutorials

Threading Tutorial

The advantage of threading is the ability to create applications that use more than one thread of execution. For example, a process can have a user interface thread that manages interactions with the user and worker threads that perform other tasks while the user interface thread waits for user input.

This tutorial demonstrates various thread activities:

- Creating and executing a thread
- Synchronization of threads
- Interaction between threads
- Using a thread pool
- Using a mutex object to protect a shared resource

Sample Files

See Threading Sample to download and build the sample files discussed in this tutorial.

Further Reading

- Threads and Threading
- Threading Objects and Features
- System.Threading Namespace
- lock Statement
- Wait Synchronization Technology Sample

Tutorial

This tutorial contains the following examples:

- Example 1: Creating, starting, and interacting between threads
- Example 2: Synchronizing two threads: a producer and a consumer
- Example 3: Using a thread pool
- Example 4: Using the Mutex object

Example 1: Creating, starting, and interacting between threads

This example demonstrates how to create and start a thread, and shows the interaction between two threads running simultaneously within the same process. Note that you don't have to stop or free the thread. This is done automatically by the .NET Framework common language runtime.

The program begins by creating an object of type `Alpha` (`oAlpha`) and a thread (`oThread`) that references the `Beta` method of the `Alpha` class. The thread is then started. The `IsAlive` property of the thread allows the program to wait until the thread is initialized (created, allocated, and so on). The main thread is accessed through `Thread`, and the `Sleep` method tells the thread to give up its time slice and stop executing for a certain amount of milliseconds. The `oThread` is then stopped and joined. Joining a thread makes the main thread wait for it to die or for a specified time to expire (for more details, see Thread.Join Method). Finally, the program attempts to restart `oThread`, but fails because a thread cannot be restarted after it is stopped (aborted). For information on temporary cessation of execution, see Suspending Thread Execution.

```
// StopJoin.cs
using System;
using System.Threading;

public class Alpha
{

   // This method that will be called when the thread is started
   public void Beta()
   {
      while (true)
      {
         Console.WriteLine("Alpha.Beta is running in its own thread.");
```

```
        }
    }
};

public class Simple
{
    public static int Main()
    {
        Console.WriteLine("Thread Start/Stop/Join Sample");

        Alpha oAlpha = new Alpha();

        // Create the thread object, passing in the Alpha.Beta method
        // via a ThreadStart delegate. This does not start the thread.
        Thread oThread = new Thread(new ThreadStart(oAlpha.Beta));

        // Start the thread
        oThread.Start();

        // Spin for a while waiting for the started thread to become
        // alive:
        while (!oThread.IsAlive);

        // Put the Main thread to sleep for 1 millisecond to allow oThread
        // to do some work:
        Thread.Sleep(1);

        // Request that oThread be stopped
        oThread.Abort();

        // Wait until oThread finishes. Join also has overloads
        // that take a millisecond interval or a TimeSpan object.
        oThread.Join();

        Console.WriteLine();
        Console.WriteLine("Alpha.Beta has finished");

        try
        {
            Console.WriteLine("Try to restart the Alpha.Beta thread");
            oThread.Start();
        }
        catch (ThreadStateException)
        {
            Console.Write("ThreadStateException trying to restart Alpha.Beta. ");
            Console.WriteLine("Expected since aborted threads cannot be restarted.");
        }
        return 0;
    }
}
```

Example Output

```
Thread Start/Stop/Join Sample
Alpha.Beta is running in its own thread.
Alpha.Beta is running in its own thread.
Alpha.Beta is running in its own thread.
...
...
Alpha.Beta has finished
Try to restart the Alpha.Beta thread
ThreadStateException trying to restart Alpha.Beta. Expected since aborted threads cannot be r
estarted.
```

Example 2: Synchronizing two threads: a producer and a consumer

The following example shows how synchronization can be accomplished using the C# **lock** keyword and the **Pulse** method of the **Monitor** object. The **Pulse** method notifies a thread in the waiting queue of a change in the object's state (for more details on pulses, see the Monitor.Pulse Method).

The example creates a `Cell` object that has two methods: `ReadFromCell` and `WriteToCell`. Two other objects are created from classes `CellProd` and `CellCons`; these objects both have a method `ThreadRun` whose job is to call `ReadFromCell` and `WriteToCell`. Synchronization is accomplished by waiting for "pulses" from the **Monitor** object, which come in order. That is, first an item is produced (the consumer at this point is waiting for a pulse), then a pulse occurs, then the consumer consumes the production (while the producer is waiting for a pulse), and so on.

```
// MonitorSample.cs
// This example shows use of the following methods of the C# lock keyword
// and the Monitor class
// in threads:
//      Monitor.Pulse(Object)
//      Monitor.Wait(Object)
using System;
using System.Threading;

public class MonitorSample
{
   public static void Main(String[] args)
   {
      int result = 0;   // Result initialized to say there is no error
      Cell cell = new Cell( );

      CellProd prod = new CellProd(cell, 20);  // Use cell for storage,
                                               // produce 20 items
      CellCons cons = new CellCons(cell, 20);  // Use cell for storage,
                                               // consume 20 items

      Thread producer = new Thread(new ThreadStart(prod.ThreadRun));
      Thread consumer = new Thread(new ThreadStart(cons.ThreadRun));
      // Threads producer and consumer have been created,
      // but not started at this point.

      try
      {
         producer.Start( );
         consumer.Start( );

         producer.Join( );   // Join both threads with no timeout
                             // Run both until done.
         consumer.Join( );
      // threads producer and consumer have finished at this point.
      }
      catch (ThreadStateException e)
      {
         Console.WriteLine(e);  // Display text of exception
         result = 1;            // Result says there was an error
      }
      catch (ThreadInterruptedException e)
      {
         Console.WriteLine(e);  // This exception means that the thread
                                // was interrupted during a Wait
         result = 1;            // Result says there was an error
      }
      // Even though Main returns void, this provides a return code to
      // the parent process.
      Environment.ExitCode = result;
   }
}

public class CellProd
{
   Cell cell;         // Field to hold cell object to be used
   int quantity = 1;  // Field for how many items to produce in cell
```

```
   public CellProd(Cell box, int request)
   {
      cell = box;          // Pass in what cell object to be used
      quantity = request;  // Pass in how many items to produce in cell
   }
   public void ThreadRun( )
   {
      for(int looper=1; looper<=quantity; looper++)
         cell.WriteToCell(looper);  // "producing"
   }
}

public class CellCons
{
   Cell cell;         // Field to hold cell object to be used
   int quantity = 1;  // Field for how many items to consume from cell

   public CellCons(Cell box, int request)
   {
      cell = box;          // Pass in what cell object to be used
      quantity = request;  // Pass in how many items to consume from cell
   }
   public void ThreadRun( )
   {
      int valReturned;
      for(int looper=1; looper<=quantity; looper++)
      // Consume the result by placing it in valReturned.
         valReturned=cell.ReadFromCell( );
   }
}

public class Cell
{
   int cellContents;         // Cell contents
   bool readerFlag = false;  // State flag
   public int ReadFromCell( )
   {
      lock(this)   // Enter synchronization block
      {
         if (!readerFlag)
         {            // Wait until Cell.WriteToCell is done producing
            try
            {
               // Waits for the Monitor.Pulse in WriteToCell
               Monitor.Wait(this);
            }
            catch (SynchronizationLockException e)
            {
               Console.WriteLine(e);
            }
            catch (ThreadInterruptedException e)
            {
               Console.WriteLine(e);
            }
         }
         Console.WriteLine("Consume: {0}",cellContents);
         readerFlag = false;    // Reset the state flag to say consuming
                                // is done.
         Monitor.Pulse(this);   // Pulse tells Cell.WriteToCell that
                                // Cell.ReadFromCell is done.
      }   // Exit synchronization block
      return cellContents;
   }

   public void WriteToCell(int n)
   {
      lock(this)  // Enter synchronization block
```

```
      {
         if (readerFlag)
         {      // Wait until Cell.ReadFromCell is done consuming.
            try
            {
               Monitor.Wait(this);   // Wait for the Monitor.Pulse in
                                     // ReadFromCell
            }
            catch (SynchronizationLockException e)
            {
               Console.WriteLine(e);
            }
            catch (ThreadInterruptedException e)
            {
               Console.WriteLine(e);
            }
         }
         cellContents = n;
         Console.WriteLine("Produce: {0}",cellContents);
         readerFlag = true;    // Reset the state flag to say producing
                               // is done
         Monitor.Pulse(this);  // Pulse tells Cell.ReadFromCell that
                               // Cell.WriteToCell is done.
      }   // Exit synchronization block
   }
}
```

Example Output

```
Produce: 1
Consume: 1
Produce: 2
Consume: 2
Produce: 3
Consume: 3
...
...
Produce: 20
Consume: 20
```

Example 3: Using a thread pool

The following example shows how to use a thread pool. It first creates a **ManualResetEvent** object, which enables the program to know when the thread pool has finished running all of the work items. Next, it attempts to add one thread to the thread pool. If this succeeds, it adds the rest (four in this example). The thread pool will then put work items into available threads. The **WaitOne** method on `eventX` is called, which causes the rest of the program to wait until the event is triggered (with the `eventX.Set` method). Finally, the program prints out the load (the thread that actually executes a particular work item) on the threads.

```
// SimplePool.cs
// Simple thread pool example
using System;
using System.Collections;
using System.Threading;

// Useful way to store info that can be passed as a state on a work item
public class SomeState
{
   public int Cookie;
   public SomeState(int iCookie)
   {
      Cookie = iCookie;
   }
}
```

```
public class Alpha
{
   public Hashtable HashCount;
   public ManualResetEvent eventX;
   public static int iCount = 0;
   public static int iMaxCount = 0;
   public Alpha(int MaxCount)
   {
      HashCount = new Hashtable(MaxCount);
      iMaxCount = MaxCount;
   }

   // Beta is the method that will be called when the work item is
   // serviced on the thread pool.
   // That means this method will be called when the thread pool has
   // an available thread for the work item.
   public void Beta(Object state)
   {
      // Write out the hashcode and cookie for the current thread
      Console.WriteLine(" {0} {1} :", Thread.CurrentThread.GetHashCode(),
         ((SomeState)state).Cookie);
      // The lock keyword allows thread-safe modification
      // of variables accessible across multiple threads.
      Console.WriteLine(
         "HashCount.Count={{0}, Thread.CurrentThread.GetHashCode()={{1}",
         HashCount.Count,
         Thread.CurrentThread.GetHashCode());
      lock (HashCount)
      {
         if (!HashCount.ContainsKey(Thread.CurrentThread.GetHashCode()))
            HashCount.Add (Thread.CurrentThread.GetHashCode(), 0);
         HashCount[Thread.CurrentThread.GetHashCode()] =
            ((int)HashCount[Thread.CurrentThread.GetHashCode()])+1;
      }

      // Do some busy work.
      // Note: Depending on the speed of your machine, if you
      // increase this number, the dispersement of the thread
      // loads should be wider.
      int iX  = 2000;
      Thread.Sleep(iX);
      // The Interlocked.Increment method allows thread-safe modification
      // of variables accessible across multiple threads.
      Interlocked.Increment(ref iCount);
      if (iCount == iMaxCount)
      {
         Console.WriteLine();
         Console.WriteLine("Setting eventX ");
         eventX.Set();
      }
   }
}

public class SimplePool
{
   public static int Main(string[] args)
   {
      Console.WriteLine("Thread Pool Sample:");
      bool W2K = false;
      int MaxCount = 10;  // Allow a total of 10 threads in the pool
      // Mark the event as unsignaled.
      ManualResetEvent eventX = new ManualResetEvent(false);
      Console.WriteLine("Queuing {0} items to Thread Pool", MaxCount);
      Alpha oAlpha = new Alpha(MaxCount);  // Create the work items.
      // Make sure the work items have a reference to the signaling event.
      oAlpha.eventX = eventX;
      Console.WriteLine("Queue to Thread Pool 0");
      try
```

```
        {
           // Queue the work items, which has the added effect of checking
           // which OS is running.
           ThreadPool.QueueUserWorkItem(new WaitCallback(oAlpha.Beta),
              new SomeState(0));
           W2K = true;
        }
        catch (NotSupportedException)
        {
           Console.WriteLine("These API's may fail when called on a non-Windows 2000 system.");
           W2K = false;
        }
        if (W2K)  // If running on an OS which supports the ThreadPool methods.
        {
           for (int iItem=1;iItem < MaxCount;iItem++)
           {
              // Queue the work items:
              Console.WriteLine("Queue to Thread Pool {0}", iItem);
              ThreadPool.QueueUserWorkItem(new WaitCallback(oAlpha.Beta),new SomeState(iItem));
           }
           Console.WriteLine("Waiting for Thread Pool to drain");
           // The call to exventX.WaitOne sets the event to wait until
           // eventX.Set() occurs.
           // (See oAlpha.Beta).
           // Wait until event is fired, meaning eventX.Set() was called:
           eventX.WaitOne(Timeout.Infinite,true);
           // The WaitOne won't return until the event has been signaled.
           Console.WriteLine("Thread Pool has been drained (Event fired)");
           Console.WriteLine();
           Console.WriteLine("Load across threads");
           foreach(object o in oAlpha.HashCount.Keys)
              Console.WriteLine("{0} {1}", o, oAlpha.HashCount[o]);
        }
        return 0;
     }
  }
```

Example Output

Note The following output will vary from one computer to another.

```
Thread Pool Sample:
Queuing 10 items to Thread Pool
Queue to Thread Pool 0
Queue to Thread Pool 1
...
...
Queue to Thread Pool 9
Waiting for Thread Pool to drain
 98 0 :
HashCount.Count==0, Thread.CurrentThread.GetHashCode()==98
 100 1 :
HashCount.Count==1, Thread.CurrentThread.GetHashCode()==100
 98 2 :
...
...
Setting eventX
Thread Pool has been drained (Event fired)

Load across threads
101 2
100 3
98 4
102 1
```

Example 4: Using the Mutex object

You can use a mutex object to protect a shared resource from simultaneous access by multiple threads or processes. The state of a mutex object is either set to signaled, when it is not owned by any thread, or nonsignaled, when it is owned. Only one thread at a time can own a mutex object. For example, to prevent two threads from writing to shared memory at the same time, each thread waits for ownership of a mutex object before executing the code that accesses the memory. After writing to the shared memory, the thread releases the mutex object.

This example demonstrates how to use the classes **Mutex**, **AutoResetEvent**, and **WaitHandle** in processing threads. It also demonstrates the methods used in processing the mutex object.

```
// Mutex.cs
// Mutex object example
using System;
using System.Threading;

public class MutexSample
{
   static Mutex gM1;
   static Mutex gM2;
   const int ITERS = 100;
   static AutoResetEvent Event1 = new AutoResetEvent(false);
   static AutoResetEvent Event2 = new AutoResetEvent(false);
   static AutoResetEvent Event3 = new AutoResetEvent(false);
   static AutoResetEvent Event4 = new AutoResetEvent(false);

   public static void Main(String[] args)
   {
      Console.WriteLine("Mutex Sample ...");
      // Create Mutex initialOwned, with name of "MyMutex".
      gM1 = new Mutex(true,"MyMutex");
      // Create Mutex initialOwned, with no name.
      gM2 = new Mutex(true);
      Console.WriteLine(" - Main Owns gM1 and gM2");

      AutoResetEvent[] evs = new AutoResetEvent[4];
      evs[0] = Event1;    // Event for t1
      evs[1] = Event2;    // Event for t2
      evs[2] = Event3;    // Event for t3
      evs[3] = Event4;    // Event for t4

      MutexSample tm = new MutexSample( );
      Thread t1 = new Thread(new ThreadStart(tm.t1Start));
      Thread t2 = new Thread(new ThreadStart(tm.t2Start));
      Thread t3 = new Thread(new ThreadStart(tm.t3Start));
      Thread t4 = new Thread(new ThreadStart(tm.t4Start));
      t1.Start( );   // Does Mutex.WaitAll(Mutex[] of gM1 and gM2)
      t2.Start( );   // Does Mutex.WaitOne(Mutex gM1)
      t3.Start( );   // Does Mutex.WaitAny(Mutex[] of gM1 and gM2)
      t4.Start( );   // Does Mutex.WaitOne(Mutex gM2)

      Thread.Sleep(2000);
      Console.WriteLine(" - Main releases gM1");
      gM1.ReleaseMutex( );  // t2 and t3 will end and signal

      Thread.Sleep(1000);
      Console.WriteLine(" - Main releases gM2");
      gM2.ReleaseMutex( );  // t1 and t4 will end and signal

      // Waiting until all four threads signal that they are done.
      WaitHandle.WaitAll(evs);
      Console.WriteLine("... Mutex Sample");
   }

   public void t1Start( )
   {
      Console.WriteLine("t1Start started,  Mutex.WaitAll(Mutex[])");
```

```
        Mutex[] gMs = new Mutex[2];
        gMs[0] = gM1;  // Create and load an array of Mutex for WaitAll call
        gMs[1] = gM2;
        Mutex.WaitAll(gMs);  // Waits until both gM1 and gM2 are released
        Thread.Sleep(2000);
        Console.WriteLine("t1Start finished, Mutex.WaitAll(Mutex[]) satisfied");
        Event1.Set( );      // AutoResetEvent.Set() flagging method is done
    }

    public void t2Start( )
    {
        Console.WriteLine("t2Start started,  gM1.WaitOne( )");
        gM1.WaitOne( );    // Waits until Mutex gM1 is released
        Console.WriteLine("t2Start finished, gM1.WaitOne( ) satisfied");
        Event2.Set( );     // AutoResetEvent.Set() flagging method is done
    }

    public void t3Start( )
    {
        Console.WriteLine("t3Start started,  Mutex.WaitAny(Mutex[])");
        Mutex[] gMs = new Mutex[2];
        gMs[0] = gM1;  // Create and load an array of Mutex for WaitAny call
        gMs[1] = gM2;
        Mutex.WaitAny(gMs);  // Waits until either Mutex is released
        Console.WriteLine("t3Start finished, Mutex.WaitAny(Mutex[])");
        Event3.Set( );       // AutoResetEvent.Set() flagging method is done
    }

    public void t4Start( )
    {
        Console.WriteLine("t4Start started,  gM2.WaitOne( )");
        gM2.WaitOne( );   // Waits until Mutex gM2 is released
        Console.WriteLine("t4Start finished, gM2.WaitOne( )");
        Event4.Set( );    // AutoResetEvent.Set() flagging method is done
    }
}
```

Sample Output

```
Mutex Sample ...
 - Main Owns gM1 and gM2
t1Start started,  Mutex.WaitAll(Mutex[])
t2Start started,  gM1.WaitOne( )
t3Start started,  Mutex.WaitAny(Mutex[])
t4Start started,  gM2.WaitOne( )
 - Main releases gM1
t2Start finished, gM1.WaitOne( ) satisfied
t3Start finished, Mutex.WaitAny(Mutex[])
 - Main releases gM2
t1Start finished, Mutex.WaitAll(Mutex[]) satisfied
t4Start finished, gM2.WaitOne( )
... Mutex Sample
```

Note The output of the example may vary from one machine to another and from one run to another. The speed and operating system of the machine running the sample can affect the output order. In multithread environments, events may not occur in the order that you expect.

See Also

C# Tutorials

Unsafe Code Tutorial

This tutorial demonstrates how to use unsafe code (code using pointers) in C#.

Sample Files

See Unsafe Code Sample to download and build the sample files discussed in this tutorial.

Further Reading

- A. Unsafe code
- unsafe
- fixed Statement

Tutorial

The use of pointers is rarely required in C#, but there are some situations that require them. As examples, using an unsafe context to allow pointers is warranted by the following cases:

- Dealing with existing structures on disk
- Advanced COM or Platform Invoke scenarios that involve structures with pointers in them
- Performance-critical code

The use of unsafe context in other situations is discouraged. Specifically, an unsafe context should not be used to attempt to write C code in C#.

> **Caution** Code written using an unsafe context cannot be verified to be safe, so it will be executed only when the code is fully trusted. In other words, unsafe code cannot be executed in an untrusted environment. For example, you cannot run unsafe code directly from the Internet.

This tutorial includes the following examples:

- Example 1 Uses pointers to copy an array of bytes.
- Example 2 Shows how to call the Windows **ReadFile** function.
- Example 3 Shows how to print the Win32 version of the executable file.

Example 1

The following example uses pointers to copy an array of bytes from `src` to `dst`. Compile the example with the /unsafe option.

```
// fastcopy.cs
// compile with: /unsafe
using System;

class Test
{
    // The unsafe keyword allows pointers to be used within
    // the following method:
    static unsafe void Copy(byte[] src, int srcIndex,
        byte[] dst, int dstIndex, int count)
    {
        if (src == null || srcIndex < 0 ||
            dst == null || dstIndex < 0 || count < 0)
        {
            throw new ArgumentException();
        }
        int srcLen = src.Length;
        int dstLen = dst.Length;
        if (srcLen - srcIndex < count ||
            dstLen - dstIndex < count)
        {
            throw new ArgumentException();
        }
```

```
            // The following fixed statement pins the location of
            // the src and dst objects in memory so that they will
            // not be moved by garbage collection.
            fixed (byte* pSrc = src, pDst = dst)
            {
                  byte* ps = pSrc;
                  byte* pd = pDst;

            // Loop over the count in blocks of 4 bytes, copying an
            // integer (4 bytes) at a time:
            for (int n =0 ; n < count/4 ; n++)
            {
                *((int*)pd) = *((int*)ps);
                pd += 4;
                ps += 4;
            }

            // Complete the copy by moving any bytes that weren't
            // moved in blocks of 4:
            for (int n =0; n < count%4; n++)
            {
                *pd = *ps;
                pd++;
                ps++;
            }
            }
    }


    static void Main(string[] args)
    {
        byte[] a = new byte[100];
        byte[] b = new byte[100];
        for(int i=0; i<100; ++i)
           a[i] = (byte)i;
        Copy(a, 0, b, 0, 100);
        Console.WriteLine("The first 10 elements are:");
        for(int i=0; i<10; ++i)
           Console.Write(b[i] + " ");
        Console.WriteLine("\n");
    }
}
```

Example Output

```
The first 10 elements are:
0 1 2 3 4 5 6 7 8 9
```

Code Discussion

- Notice the use of the **unsafe** keyword, which allows pointers to be used within the `Copy` method.
- The **fixed** statement is used to declare pointers to the source and destination arrays. It pins the location of the `src` and `dst` objects in memory so that they will not be moved by garbage collection. The objects will be unpinned when the **fixed** block completes
- Unsafe code increases the performance by getting rid of array bounds checks.

Example 2

This example shows how to call the Windows **ReadFile** function from the *Platform SDK*, which requires the use of an unsafe context because the read buffer requires a pointer as a parameter.

```
// readfile.cs
```

```
// compile with: /unsafe
// arguments: readfile.cs

// Use the program to read and display a text file.
using System;
using System.Runtime.InteropServices;
using System.Text;

class FileReader
{
      const uint GENERIC_READ = 0x80000000;
      const uint OPEN_EXISTING = 3;
      IntPtr handle;

      [DllImport("kernel32", SetLastError=true)]
      static extern unsafe IntPtr CreateFile(
            string FileName,                    // file name
            uint DesiredAccess,                 // access mode
            uint ShareMode,                     // share mode
            uint SecurityAttributes,            // Security Attributes
            uint CreationDisposition,           // how to create
            uint FlagsAndAttributes,            // file attributes
            int hTemplateFile                   // handle to template file
            );

       [DllImport("kernel32", SetLastError=true)]
      static extern unsafe bool ReadFile(
            IntPtr hFile,                       // handle to file
            void* pBuffer,                      // data buffer
            int NumberOfBytesToRead,            // number of bytes to read
            int* pNumberOfBytesRead,            // number of bytes read
            int Overlapped                      // overlapped buffer
            );

      [DllImport("kernel32", SetLastError=true)]
      static extern unsafe bool CloseHandle(
            IntPtr hObject   // handle to object
            );

      public bool Open(string FileName)
      {
            // open the existing file for reading
            handle = CreateFile(
                  FileName,
                  GENERIC_READ,
                  0,
                  0,
                  OPEN_EXISTING,
                  0,
                  0);

            if (handle != IntPtr.Zero)
                  return true;
            else
                  return false;
      }

      public unsafe int Read(byte[] buffer, int index, int count)
      {
            int n = 0;
            fixed (byte* p = buffer)
            {
                  if (!ReadFile(handle, p + index, count, &n, 0))
                        return 0;
            }
            return n;
      }
```

```
        public bool Close()
        {
                // close file handle
                return CloseHandle(handle);
        }
}

class Test
{
        public static int Main(string[] args)
        {
                if (args.Length != 1)
                {
                        Console.WriteLine("Usage : ReadFile <FileName>");
                        return 1;
                }

                if (! System.IO.File.Exists(args[0]))
                {
                        Console.WriteLine("File " + args[0] + " not found.");
                        return 1;
                }

                byte[] buffer = new byte[128];
                FileReader fr = new FileReader();

                if (fr.Open(args[0]))
                {

                        // Assume that an ASCII file is being read
                        ASCIIEncoding Encoding = new ASCIIEncoding();

                        int bytesRead;
                        do
                        {
                                bytesRead = fr.Read(buffer, 0, buffer.Length);
                                string content = Encoding.GetString(buffer,0,bytesRead);
                                Console.Write("{0}", content);
                        }
                        while ( bytesRead > 0);

                        fr.Close();
                        return 0;
                }
                else
                {
                        Console.WriteLine("Failed to open requested file");
                        return 1;
                }
        }
}
```

Example Input

The following input from readfile.txt produces the output shown in Example Output when you compile and run this sample.

```
line 1
line 2
```

Example Output

```
line 1
line 2
```

Code Discussion

The byte array passed into the **Read** function is a managed type. This means that the common language runtime garbage collector could relocate the memory used by the array at will. The **fixed** statement allows you to both get a pointer to the memory used by the byte array and to mark the instance so that the garbage collector won't move it.

At the end of the **fixed** block, the instance will be marked so that it can be moved. This capability is known as declarative pinning. The nice part about pinning is that there is very little overhead unless a garbage collection occurs in the **fixed** block, which is an unlikely occurrence.

Example 3

This example reads and displays the Win32 version number of the executable file, which is the same as the assembly version number in this example. The executable file, in this example, is printversion.exe. The example uses the *Platform SDK* functions VerQueryValue, GetFileVersionInfoSize, and GetFileVersionInfo to retrieve specified version information from the specified version-information resource.

This example uses pointers because it simplifies the use of methods whose signatures use pointers to pointers, which are common in the Win32 APIs.

```
// printversion.cs
// compile with: /unsafe
using System;
using System.Reflection;
using System.Runtime.InteropServices;

// Give this assembly a version number:
[assembly:AssemblyVersion("4.3.2.1")]

public class Win32Imports
{
      [DllImport("version.dll")]
      public static extern bool GetFileVersionInfo (string sFileName,
            int handle, int size, byte[] infoBuffer);
      [DllImport("version.dll")]
      public static extern int GetFileVersionInfoSize (string sFileName,
            out int handle);

      // The third parameter - "out string pValue" - is automatically
      // marshaled from ANSI to Unicode:
      [DllImport("version.dll")]
      unsafe public static extern bool VerQueryValue (byte[] pBlock,
            string pSubBlock, out string pValue, out uint len);
      // This VerQueryValue overload is marked with 'unsafe' because
      // it uses a short*:
      [DllImport("version.dll")]
      unsafe public static extern bool VerQueryValue (byte[] pBlock,
            string pSubBlock, out short *pValue, out uint len);
}

public class C
{
      // Main is marked with 'unsafe' because it uses pointers:
      unsafe public static int Main ()
      {
            try
            {
                  int handle = 0;
                  // Figure out how much version info there is:
                  int size =
                        Win32Imports.GetFileVersionInfoSize("printversion.exe",
                        out handle);

                  if (size == 0) return -1;

                  byte[] buffer = new byte[size];
```

```
                    if (!Win32Imports.GetFileVersionInfo("printversion.exe", handle, size, buff
er))
                    {
                          Console.WriteLine("Failed to query file version information.");
                          return 1;
                    }

                    short *subBlock = null;
                    uint len = 0;
                    // Get the locale info from the version info:
                    if (!Win32Imports.VerQueryValue (buffer, @"\VarFileInfo\Translation", out s
ubBlock, out len))
                    {
                          Console.WriteLine("Failed to query version information.");
                          return 1;
                    }

                    string spv = @"\StringFileInfo\" + subBlock[0].ToString("X4") + subBlock[1]
.ToString("X4") + @"\ProductVersion";

                    byte *pVersion = null;
                    // Get the ProductVersion value for this program:
                    string versionInfo;

                    if (!Win32Imports.VerQueryValue (buffer, spv, out versionInfo, out len))
                    {
                          Console.WriteLine("Failed to query version information.");
                          return 1;
                    }

                   Console.WriteLine ("ProductVersion == {0}", versionInfo);
              }
              catch (Exception e)
              {
                    Console.WriteLine ("Caught unexpected exception " + e.Message);
              }

              return 0;
        }
}
```

Example Output

```
ProductVersion == 4.3.2.1
```

See Also

C# Tutorials

OLE DB Tutorial

OLE DB is a COM-based application programming interface (API) for accessing data. OLE DB supports accessing data stored in any format (databases, spreadsheets, text files, and so on) for which an OLE DB provider is available. Each OLE DB provider exposes data from a particular type of data source (for example SQL Server databases, Microsoft Access databases, or Microsoft Excel spreadsheets).

This tutorial demonstrates using a Microsoft Access database from a C# application.

Sample Files

See OLE DB Sample to download and build the sample files discussed in this tutorial.

Further Reading

- System.Data.OleDb Namespace
- OLE DB Programming Overview (using OLE DB in Visual C++)
- OLE DB Templates (using the OLE DB Templates)
- OLE DB Programmer's Reference (the technology underlying the System.Data.OleDb namespace and the OLE DB Templates)
- Choosing Data Access Technologies
- Choosing Tools and Technologies

Tutorial

This tutorial demonstrates how to use a Microsoft Access database from C#. It shows how you can create a dataset and add tables to it from a database. The BugTypes.mdb database used in the sample program is a Microsoft Access 2000 .MDB file.

Example

This program accesses the BugTypes.mdb database, creates a dataset, adds the tables to it, and displays the number of tables, columns, and rows. It also displays the titles of each row.

```
// OleDbSample.cs
using System;
using System.Data;
using System.Data.OleDb;
using System.Xml.Serialization;

public class MainClass {
    public static void Main ()
    {
            // Set Access connection and select strings.
            // The path to BugTypes.MDB must be changed if you build
            // the sample from the command line:
#if USINGPROJECTSYSTEM
            string strAccessConn = "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=..\\..\\BugT
ypes.MDB";
#else
            string strAccessConn = "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=BugTypes.MDB
";
#endif
            string strAccessSelect = "SELECT * FROM Categories";

            // Create the dataset and add the Categories table to it:
            DataSet myDataSet = new DataSet();
            OleDbConnection myAccessConn = null;
            try
            {
                  myAccessConn = new OleDbConnection(strAccessConn);
            }
            catch(Exception ex)
            {
                  Console.WriteLine("Error: Failed to create a database connection. \n{0}", e
```

```
x.Message);
                return;
            }

            try
            {

                OleDbCommand myAccessCommand = new OleDbCommand(strAccessSelect,myAccessCon
n);
                OleDbDataAdapter myDataAdapter = new OleDbDataAdapter(myAccessCommand);

                myAccessConn.Open();
                myDataAdapter.Fill(myDataSet,"Categories");

            }
            catch (Exception ex)
            {
                Console.WriteLine("Error: Failed to retrieve the required data from the Dat
aBase.\n{0}", ex.Message);
                return;
            }
            finally
            {
                myAccessConn.Close();
            }

            // A dataset can contain multiple tables, so let's get them
            // all into an array:
            DataTableCollection dta = myDataSet.Tables;
            foreach (DataTable dt in dta)
            {
            Console.WriteLine("Found data table {0}", dt.TableName);
            }

            // The next two lines show two different ways you can get the
            // count of tables in a dataset:
            Console.WriteLine("{0} tables in data set", myDataSet.Tables.Count);
            Console.WriteLine("{0} tables in data set", dta.Count);
            // The next several lines show how to get information on
            // a specific table by name from the dataset:
            Console.WriteLine("{0} rows in Categories table", myDataSet.Tables["Categories"].
Rows.Count);
            // The column info is automatically fetched from the database,
            // so we can read it here:
            Console.WriteLine("{0} columns in Categories table", myDataSet.Tables["Categories
"].Columns.Count);
            DataColumnCollection drc = myDataSet.Tables["Categories"].Columns;
            int i = 0;
            foreach (DataColumn dc in drc)
            {
                // Print the column subscript, then the column's name
                // and its data type:
                Console.WriteLine("Column name[{0}] is {1}, of type {2}",i++ , dc.ColumnNam
e, dc.DataType);
            }
            DataRowCollection dra = myDataSet.Tables["Categories"].Rows;
            foreach (DataRow dr in dra)
            {
                // Print the CategoryID as a subscript, then the CategoryName:
                Console.WriteLine("CategoryName[{0}] is {1}", dr[0], dr[1]);
            }

    }
}
```

Output

The `Categories` table of the BugTypes.mdb database contains the following information.

Category ID	Category Name
1	Bugbash Stuff
2	Appweek Bugs
3	.NET Framework Reports
4	Internal Support

When you run the sample, the following output will be displayed on the screen:

```
Found data table Categories
1 tables in data set
1 tables in data set
4 rows in Categories table
2 columns in Categories table
Column name[0] is CategoryID, of type System.Int32
Column name[1] is CategoryName, of type System.String
CategoryName[1] is Bugbash Stuff
CategoryName[2] is Appweek Bugs
CategoryName[3] is .NET Framework Reports
CategoryName[4] is Internal Support
```

See Also

C# Tutorials

C# Keywords

Keywords are predefined reserved identifiers that have special meanings to the compiler. They cannot be used as identifiers in your program unless they include `@` as a prefix. For example, `@if` is a legal identifier but `if` is not because it is a keyword.

abstract	event	new	struct
as	explicit	null	switch
base	extern	object	this
bool	false	operator	throw
break	finally	out	true
byte	fixed	override	try
case	float	params	typeof
catch	for	private	uint
char	foreach	protected	ulong
checked	goto	public	unchecked
class	if	readonly	unsafe
const	implicit	ref	ushort
continue	in	return	using
decimal	int	sbyte	virtual
default	interface	sealed	volatile
delegate	internal	short	void
do	is	sizeof	while
double	lock	stackalloc	
else	long	static	
enum	namespace	string	

See Also

C# Preprocessor Directives | C# Operators | C# Programmer's Reference

Types

The C# typing system contains the following categories:

- Value types
- Reference types
- A.2 Pointer types

Variables of the value types store data, while those of the reference types store references to the actual data. Reference types are also referred to as objects. Pointer types can be used only in unsafe mode.

It is possible to convert a value type to a reference type and vice versa, by using boxing and unboxing.

This section also introduces the void type.

See Also

C# Keywords | Types Reference Tables

Value Types

The value types consist of two main categories:

- Struct type
- Enumeration type

The struct types contain the user-defined struct types and the following built-in simple types:

- Numeric types
    - Integral types
    - Floating-point types
    - decimal
- bool

In addition to explaining each category of the value types, the following topics are discussed in this section:

- Main Features of Value Types
- Main Features of Simple Types
- Initializing Value Types

Main Features of Value Types

A variable of a value type always contains a value of that type. The assignment to a variable of a value type creates a copy of the assigned value, while the assignment to a variable of a reference type creates a copy of the reference but not of the referenced object.

All value types are derived implicitly from the Object class.

Unlike reference types, it is not possible to derive a new type from a value type. However, like reference types, structs can implement interfaces.

Unlike reference types, it is not possible for a value type to contain the **null** value.

Each value type has an implicit default constructor that initializes the default value of that type. For information on default values of value types, see Default Values Table.

Main Features of Simple Types

All of the simple types are aliases of the .NET Framework System types. For example, **int** is an alias of **System.Int32**. For a complete list of aliases, see Built-in Types Table.

Constant expressions, whose operands are all simple type constants, are evaluated at compilation time. For more information, see 7.15 Constant expressions.

Simple types can be initialized using literals. For example, 'A' is a literal of the type **char** and 2001 is a literal of the type **int**.

Initializing Value Types

Local variables in C# must be initialized before being used. Therefore, if you declare a local variable without initialization like this:

```
int myInt;
```

you cannot use it before you initialize it. You can initialize it using the following statement:

```
myInt = new int();  // Invoke the default constructor for the type int
```

which is equivalent to:

```
myInt = 0;          // Assign an initial value, 0 in this example
```

You can, of course, have the declaration and the initialization in the same statement like this:

```
int myInt = new int();
```

–or–

```
int myInt = 0;
```

Using the **new** operator calls the default constructor of the specific type and assigns the default value to the variable. In the preceding example, the default constructor assigned the value `0` to `myInt`. For more information on values assigned by calling default constructors, see Default Values Table.

With user-defined types, use **new** to invoke the default constructor. For example, the following statement invokes the default constructor of the `Point` struct:

```
Point p = new Point();   // Invoke the default constructor for the struct
```

After this call, the struct is considered to be definitely assigned; that is, all of its members are initialized to their default values. For more information, see 5.3 Definite assignment.

For more information on the new operator, see new.

For information on formatting the output of numeric types, see Formatting Numeric Results Table.

See Also

C# Keywords | Types | Types Reference Tables | Reference Types

bool

The **bool** keyword is an alias of **System.Boolean**. It is used to declare variables to store the Boolean values, true and false.

Literals

You can assign a Boolean value to a **bool** variable, for example:

```
bool MyVar = true;
```

You can also assign an expression that evaluates to **bool** to a **bool** variable, for example:

```
bool Alphabetic = (c > 64 && c < 123);
```

Conversions

In C++, a value of type **bool** can be converted to a value of type **int**; in other words, **false** is equivalent to zero and **true** is equivalent to nonzero values. In C#, there is no conversion between the **bool** type and other types. For example, the following **if** statement is invalid in C#, while it is legal in C++:

```
int x = 123;
if (x)   // Invalid in C#
{
   printf("The value of x is nonzero.");
}
```

To test a variable of the type **int**, you have to explicitly compare it to a value (for example, zero), that is:

```
int x = 123;
if (x != 0)   // The C# way
{
   Console.Write("The value of x is nonzero.");
}
```

Example

In this example, you enter a character from the keyboard and the program checks if the input character is a letter. If so, it checks if it is lowercase or uppercase. In each case, the proper message is displayed.

```
// keyword_bool.cs
// Character Tester
using System;
public class BoolTest
{
   public static void Main()
   {
      Console.Write("Enter a character: ");
      char c = (char) Console.Read();

      if (Char.IsLetter(c))
         if (Char.IsLower(c))
            Console.WriteLine("The character is lowercase.");
         else
            Console.WriteLine("The character is uppercase.");
      else
         Console.WriteLine("The character is not an alphabetic character.");
   }
}
```

Input

```
X
```

Sample Output

```
Enter a character: X
The character is uppercase.
```

Additional sample runs might look as follow:

```
Enter a character: x
The character is lowercase.

Enter a character: 2
The character is not an alphabetic character.
```

See Also

C# Keywords | Default Values Table | Built-in Types Table | Operator Overloading Tutorial

byte

The **byte** keyword denotes an integral type that stores values as indicated in the following table.

Type	Range	Size	.NET Framework type
byte	0 to 255	Unsigned 8-bit integer	**System.Byte**

Literals

You can declare and initialize a **byte** variable like this example:

```
byte myByte = 255;
```

In the preceding declaration, the integer literal `255` is implicitly converted from **int** to **byte**. If the integer literal exceeds the range of **byte**, a compilation error will occur.

Conversions

There is a predefined implicit conversion from **byte** to **short**, **ushort**, **int**, **uint**, **long**, **ulong**, **float**, **double**, or **decimal**.

You cannot implicitly convert nonliteral numeric types of larger storage size to **byte** (see Integral Types Table for the storage sizes of integral types). Consider, for example, the following two **byte** variables `x` and `y`:

```
byte x = 10, y = 20;
```

The following assignment statement will produce a compilation error, because the arithmetic expression on the right-hand side of the assignment operator evaluates to **int** by default.

```
byte z = x + y;   // Error: conversion from int to byte
```

To fix this problem, use a cast:

```
byte z = (byte)(x + y);   // OK: explicit conversion
```

It is possible though to use the following statements, where the destination variable has the same storage size or a larger storage size:

```
int x = 10, y = 20;
int m = x + y;
long n = x + y;
```

Also, there is no implicit conversion from floating-point types to **byte**. For example, the following statement generates a compiler error unless an explicit cast is used:

```
byte x = 3.0;         // Error: no implicit conversion from double
byte y = (byte)3.0;   // OK: explicit conversion
```

When calling overloaded methods, a cast must be used. Consider, for example, the following overloaded methods that use **byte** and **int** parameters:

```
public static void MyMethod(int i) {}
public static void MyMethod(byte b) {}
```

Using the **byte** cast guarantees that the correct type is called, for example:

```
MyMethod(5);        // Calling the method with the int parameter
```

```
MyMethod((byte)5);  // Calling the method with the byte parameter
```

For information on arithmetic expressions with mixed floating-point types and integral types, see float and double.

For more information on implicit numeric conversion rules, see the Implicit Numeric Conversions Table.

See Also

C# Keywords | Integral Types Table | Built-in Types Table | Implicit Numeric Conversions Table | Explicit Numeric Conversions Table | Byte Structure

char

The **char** keyword is used to declare a Unicode character in the range indicated in the following table. Unicode characters are 16-bit characters used to represent most of the known written languages throughout the world.

Type	Range	Size	.NET Framework type
char	U+0000 to U+ffff	Unicode 16-bit character	**System.Char**

Literals

Constants of the **char** type can be written as character literals, hexadecimal escape sequence, or Unicode representation. You can also cast the integral character codes. All of the following statements declare a **char** variable and initialize it with the character `X`:

```
char MyChar = 'X';        // Character literal
char MyChar = '\x0058';   // Hexadecimal
char MyChar = (char)88;   // Cast from integral type
char MyChar = '\u0058';   // Unicode
```

Conversions

A **char** can be implicitly converted to **ushort**, **int**, **uint**, **long**, **ulong**, **float**, **double**, or **decimal**. However, there are no implicit conversions from other types to the **char** type.

See Also

C# Keywords | Integral Types Table | Built-in Types Table | Implicit Numeric Conversions Table | Explicit Numeric Conversions Table | Char Structure

decimal

The **decimal** keyword denotes a 128-bit data type. Compared to floating-point types, the **decimal** type has a greater precision and a smaller range, which makes it suitable for financial and monetary calculations. The approximate range and precision for the **decimal** type are shown in the following table.

Type	Approximate Range	Precision	.NET Framework type
decimal	$\pm 1.0 \times 10^{-28}$ to $\pm 7.9 \times 10^{28}$	28-29 significant digits	**System.Decimal**

Literals

If you want a numeric real literal to be treated as **decimal**, use the suffix m or M, for example:

```
decimal myMoney = 300.5m;
```

Without the suffix m, the number is treated as a **double**, thus generating a compiler error.

Conversions

The integral types are implicitly converted to **decimal** and the result evaluates to **decimal**. Therefore you can initialize a decimal variable using an integer literal, without the suffix, for example:

```
decimal myMoney = 300;
```

There is no implicit conversion between floating-point types and the **decimal** type; therefore, a cast must be used to convert between these two types. For example:

```
decimal myMoney = 99.9m;
double x = (double) myMoney;
myMoney = (decimal) x;
```

You can also mix **decimal** and numeric integral types in the same expression. However, mixing **decimal** and floating-point types without a cast results in a compilation error.

For more information on implicit numeric conversions, see the Implicit Numeric Conversions Table.

For more information on explicit numeric conversions, see the Explicit Numeric Conversions Table.

Example 1

In this example, a **decimal** and an **int** are mixed in the same expression. The result evaluates to the **decimal** type.

```
// keyword_decimal.cs
// decimal conversion
using System;
public class TestDecimal {
   public static void Main ()
   {
      decimal d = 9.1m;
      int y = 3;
      Console.WriteLine(d + y);   // Result converted to decimal
   }
}
```

Output

```
12.1
```

In the preceding example, if you attempt to add the **double** and **decimal** variables by using a statement like this:

```
Console.WriteLine(d + x);   // error
```

you will get the following error:

```
Operator '+' cannot be applied to operands of type 'double' and 'decimal'
```

Formatting Decimal Output

You can format the results by using the **String.Format** method, or through the **Console.Write** method, which calls **String.Format()**. The currency format is specified using the standard currency format string "C" or "c," as shown in Example 2.

Example 2

In this example, the output is formatted using the currency format string. Notice that `x` is rounded because the decimal places exceed $0.99. The variable `y`, which represents the maximum exact digits, is displayed exactly in the proper format.

```
// keyword_decimal2.cs
// Decimal type formatting
using System;
public class TestDecimalFormat
{
   public static void Main ()
   {
      decimal x = 0.999m;
      decimal y = 9999999999999999999999999999m;
      Console.WriteLine("My amount = {0:C}", x);
      Console.WriteLine("Your amount = {0:C}", y);
   }
}
```

Output

```
My amount = $1.00
Your amount = $9,999,999,999,999,999,999,999,999,999.00
```

For more information on the **String.Format** method, see String.Format Method. For more information on the decimal type precision, see 4.1.6 The decimal type.

See Also

C# Keywords | Default Values Table | Built-in Types Table | Floating-Point Types Table | Implicit Numeric Conversions Table | Explicit Numeric Conversions Table | Decimal Structure

double

The **double** keyword denotes a simple type that stores 64-bit floating-point values. The following table shows the precision and approximate range for the **double** type.

Type	Approximate range	Precision	.NET Framework type
double	$\pm 5.0 \times 10^{-324}$ to $\pm 1.7 \times 10^{308}$	15-16 digits	**System.Double**

Literals

By default, a real numeric literal on the right-hand side of the assignment operator is treated as **double**. However, if you want an integer number to be treated as **double**, use the suffix d or D, for example:

```
double x = 3D;
```

Conversions

You can mix numeric integral types and floating-point types in an expression. In this case, the integral types are converted to floating-point types. The evaluation of the expression is performed according to the following rules:

- If one of the floating-point types is **double**, the expression evaluates to **double** (or **bool** in the case of relational or Boolean expressions).
- If there is no **double** type in the expression, it evaluates to **float** (or **bool** in the case of relational or Boolean expressions).

A floating-point expression can contain the following sets of values:

- Positive and negative zero
- Positive and negative infinity
- Not-a-Number value (NaN)
- The finite set of nonzero values

For more information on these values, refer to IEEE Standard for Binary Floating-Point Arithmetic, available on the Web site http://www.ieee.org/.

For more information on floating-point value sets, see 4.1.5 Floating point types.

Example

In the following example, an **int**, a **short**, a **float**, and a **double** are added together giving a **double** result.

```
// keyword_double.cs
// Mixing types in expressions
using System;
class MixedTypes
{
   public static void Main()
   {
      int x = 3;
      float y = 4.5f;
      short z = 5;
      double w = 1.7E+3;
      Console.WriteLine("The sum is {0}", x + y + z + w);   // double result
   }
}
```

Output

```
The sum is 1712.5
```

See Also

C# Keywords | Default Values Table | Built-in Types Table | Floating-Point Types Table | Implicit Numeric Conversions Table | Explicit Numeric Conversions Table | Double Structure

enum

The **enum** keyword is used to declare an enumeration, a distinct type consisting of a set of named constants called the enumerator list. Every enumeration type has an underlying type, which can be any integral type except **char**. This declaration takes the following form::

```
[attributes] [modifiers] enum identifier [:base-type] {enumerator-list} [;]
```

where:

attributes (Optional)
Additional declarative information. For more information on attributes and attribute classes, see 17. Attributes.
modifiers (Optional)
The allowed modifiers are new and the four access modifiers.
identifier
The **enum** name.
base-type (Optional)
The underlying type that specifies the storage allocated for each enumerator. It can be one of the integral types except **char**. The default is **int**.
enumerator-list
The enumerators' identifiers separated by commas, optionally including a value assignment.

Remarks

The default underlying type of the enumeration elements is **int**. By default, the first enumerator has the value 0, and the value of each successive enumerator is increased by 1. For example:

```
enum Days {Sat, Sun, Mon, Tue, Wed, Thu, Fri};
```

In this enumeration, `Sat` is `0`, `Sun` is `1`, `Mon` is `2`, and so forth. Enumerators can have initializers to override the default values. For example:

```
enum Days {Sat=1, Sun, Mon, Tue, Wed, Thu, Fri};
```

In this enumeration, the sequence of elements is forced to start from `1` instead of `0`.

The default value of an **enum** E is the value produced by the expression `(E)0`.

The underlying type specifies how much storage is allocated for each enumerator. However, an explicit cast is needed to convert from **enum** type to an integral type. For example, the following statement assigns the enumerator `Sun` to a variable of the type **int** using a cast to convert from **enum** to **int**:

```
int x = (int) Days.Sun;
```

For more information on enumeration types, see 14. Enums.

Example 1

In this example, an enumeration, `Days`, is declared. Two enumerators are explicitly converted to **int** and assigned to **int** variables.

```
// keyword_enum.cs
// enum initialization:
using System;
public class EnumTest
{
   enum Days {Sat=1, Sun, Mon, Tue, Wed, Thu, Fri};

   public static void Main()
   {
      int x = (int) Days.Sun;
```

```
        int y = (int) Days.Fri;
        Console.WriteLine("Sun = {0}", x);
        Console.WriteLine("Fri = {0}", y);
    }
}
```

Output

```
Sun = 2
Fri = 7
```

Notice that if you remove the initializer from `Sat=1`, the result will be:

```
Sun = 1
Fri = 6
```

Example 2

In this example, the *base-type* option is used to declare an **enum** whose members are of the type **long**. Notice that even though the underlying type of the enumeration is **long**, the enumeration members must still be explicitly converted to type **long** using a cast.

```
// keyword_enum2.cs
// Using long enumerators
using System;
public class EnumTest
{
    enum Range :long {Max = 2147483648L, Min = 255L};
    public static void Main()
    {
        long x = (long) Range.Max;
        long y = (long) Range.Min;
        Console.WriteLine("Max = {0}", x);
        Console.WriteLine("Min = {0}", y);
    }
}
```

Output

```
Max = 2147483648
Min = 255
```

See Also

C# Keywords | Default Values Table | Built-in Types Table | Types | Value Types

float

The **float** keyword denotes a simple type that stores 32-bit floating-point values. The following table shows the precision and approximate range for the **float** type.

Type	Approximate range	Precision	.NET Framework type
float	$\pm 1.5 \times 10^{-45}$ to $\pm 3.4 \times 10^{38}$	7 digits	**System.Single**

Literals

By default, a real numeric literal on the right-hand side of the assignment operator is treated as **double**. Therefore, to initialize a float variable use the suffix `f` or `F`, for example:

```
float x = 3.5F;
```

If you don't use the suffix in the previous declaration, you will get a compilation error because you are attempting to store a **double** value into a **float** variable.

Conversions

You can mix numeric integral types and floating-point types in an expression. In this case, the integral types are converted to floating-point types. The evaluation of the expression is performed according to the following rules:

- If one of the floating-point types is **double**, the expression evaluates to **double** (or **bool** in the case of relational or Boolean expressions).
- If there is no **double** type in the expression, the expression evaluates to **float** (or **bool** in the case of relational or Boolean expressions).

A floating-point expression can contain the following sets of values:

- Positive and negative zero
- Positive and negative infinity
- Not-a-Number value (NaN)
- The finite set of nonzero values

For more information on these values, refer to IEEE Standard for Binary Floating-Point Arithmetic, available on the Web site http://www.ieee.org/.

For more information on floating-point value sets, see 4.1.5 Floating point types.

Example

In the following example, an **int**, a **short**, and a **float** are included in a mathematical expression giving a **float** result (notice that there is no **double** in the expression).

```
// keyword_float.cs
// Mixing types in expressions
using System;
class MixedTypes
{
   public static void Main()
   {
      int x = 3;
      float y = 4.5f;
      short z = 5;
      Console.WriteLine("The result is {0}", x*y/z);
   }
}
```

Output

```
The result is 2.7
```

See Also

C# Keywords | Default Values Table | Built-in Types Table | Floating-Point Types Table | Implicit Numeric Conversions Table | Explicit Numeric Conversions Table | Single Structure

int

The **int** keyword denotes an integral type that stores values according to the size and range shown in the following table.

Type	Range	Size	.NET Framework type
int	-2,147,483,648 to 2,147,483,647	Signed 32-bit integer	**System.Int32**

Literals

You can declare and initialize a variable of the type **int** like this example:

```
int myInt = 123;
```

When an integer literal has no suffix, its type is the first of these types in which its value can be represented: **int**, **uint**, **long**, **ulong**. In this example, it is of the type **int**.

Conversions

There is a predefined implicit conversion from **int** to **long**, **float**, **double**, or **decimal**. For example:

```
float myFloat = 123;   // OK: implicit conversion to float
```

There is a predefined implicit conversion from **sbyte**, **byte**, **short**, **ushort**, or **char** to **int**. For example, if you have a **long** variable, `myLong`, the following assignment statement will produce a compilation error without a cast:

```
long myLong = 22;
int myInt = myLong;       // Error: no implicit conversion from long
int myInt = (int)myLong;  // OK: explicit conversion
```

Notice also that there is no implicit conversion from floating-point types to **int**. For example, the following statement generates a compiler error unless an explicit cast is used:

```
int x = 3.0;         // Error: no implicit conversion from double
int y = (int)3.0;    // OK: explicit conversion
```

For information on arithmetic expressions with mixed floating-point types and integral types, see float and double.

See Also

C# Keywords | Integral Types Table | Built-in Types Table | Implicit Numeric Conversions Table | Explicit Numeric Conversions Table | Int32 Structure

long

The **long** keyword denotes an integral type that stores values according to the size and range shown in the following table.

Type	Range	Size	.NET Framework type
long	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	Signed 64-bit integer	**System.Int64**

Literals

You can declare and initialize a **long** variable like this example:

```
long myLong = 4294967296;
```

When an integer literal has no suffix, its type is the first of these types in which its value can be represented: **int**, **uint**, **long**, **ulong**. In the preceding example, it is of the type **long** because it exceeds the range of **uint** (see Integral Types Table for the storage sizes of integral types).

You can also use the suffix L with the **long** type like this:

```
long myLong = 4294967296L;
```

When you use the suffix L, the type of the literal integer is determined to be either **long** or **ulong** according to its size. In the case it is **long** because it less than the range of **ulong**.

A common use of the suffix is with calling overloaded methods. Consider, for example, the following overloaded methods that use **long** and **int** parameters:

```
public static void MyMethod(int i) {}
public static void MyMethod(long l) {}
```

Using the suffix L guarantees that the correct type is called, for example:

```
MyMethod(5);    // Calling the method with the int parameter
MyMethod(5L);   // Calling the method with the long parameter
```

You can use the **long** type with other numeric integral types in the same expression, in which case the expression is evaluated as **long** (or **bool** in the case of relational or Boolean expressions). For example, the following expression evaluates as **long**:

```
898L + 88
```

> **Note** You can also use the lowercase letter "l" as a suffix. However, this generates a compiler warning because the letter "l" is easily confused with the digit "1." Use "L" for clarity.

For information on arithmetic expressions with mixed floating-point types and integral types, see float and double.

Conversions

There is a predefined implicit conversion from **long** to **float**, **double**, or **decimal**. Otherwise a cast must be used. For example, the following statement will produce a compilation error without an explicit cast:

```
int x = 8L;        // Error: no implicit conversion from long to int
int x = (int)8L;   // OK: explicit conversion to int
```

There is a predefined implicit conversion from **sbyte**, **byte**, **short**, **ushort**, **int**, **uint**, or **char** to **long**.

Notice also that there is no implicit conversion from floating-point types to **long**. For example, the following statement generates a compiler error unless an explicit cast is used:

```
long x = 3.0;         // Error: no implicit conversion from double
long y = (long)3.0;   // OK: explicit conversion
```

See Also

C# Keywords | Integral Types Table | Built-in Types Table | Implicit Numeric Conversions Table | Explicit Numeric Conversions Table | Int64 Structure

sbyte

The **sbyte** keyword denotes an integral type that stores values according to the size and range shown in the following table.

Type	Range	Size	.NET Framework type
sbyte	-128 to 127	Signed 8-bit integer	**System.SByte**

Literals

You can declare and initialize an **sbyte** variable like this example:

```
sbyte mySbyte = 127;
```

In the preceding declaration, the integer literal 127 is implicitly converted from **int** to **sbyte**. If the integer literal exceeds the range of **sbyte**, a compilation error will occur.

A cast must be used when calling overloaded methods. Consider, for example, the following overloaded methods that use **sbyte** and **int** parameters:

```
public static void MyMethod(int i) {}
public static void MyMethod(sbyte b) {}
```

Using the **sbyte** cast guarantees that the correct type is called, for example:

```
MyMethod(5);         // Calling the method with the int parameter
MyMethod((sbyte)5);  // Calling the method with the sbyte parameter
```

Conversions

There is a predefined implicit conversion from **sbyte** to **short**, **int**, **long**, **float**, **double**, or **decimal**.

You cannot implicitly convert nonliteral numeric types of larger storage size to **sbyte** (see Integral Types Table for the storage sizes of integral types). Consider, for example, the following two **sbyte** variables `x` and `y`:

```
sbyte x = 10, y = 20;
```

The following assignment statement will produce a compilation error, because the arithmetic expression on the right-hand side of the assignment operator evaluates to **int** by default.

```
sbyte z = x + y;   // Error: conversion from int to sbyte
```

To fix this problem, use a cast:

```
sbyte z = (byte)(x + y);   // OK: explicit conversion
```

It is possible though to use the following statements, where the destination variable has the same storage size or a larger storage size:

```
sbyte x = 10, y = 20;
int m = x + y;
long n = x + y;
```

Notice also that there is no implicit conversion from floating-point types to **sbyte**. For example, the following statement generates a compiler error unless an explicit cast is used:

```
sbyte x = 3.0;         // Error: no implicit conversion from double
sbyte y = (sbyte)3.0;  // OK: explicit conversion
```

For information on arithmetic expressions with mixed floating-point types and integral types, see float and double.

For more information on implicit numeric conversion rules, see the Implicit Numeric Conversions Table.

See Also

C# Keywords | Integral Types Table | Built-in Types Table | Implicit Numeric Conversions Table | Explicit Numeric Conversions Table | SByte Structure

short

The **short** keyword denotes an integral data type that stores values according to the size and range shown in the following table.

Type	Range	Size	.NET Framework type
short	-32,768 to 32,767	Signed 16-bit integer	**System.Int16**

Literals

You can declare and initialize a **short** variable like this example:

```
short x = 32767;
```

In the preceding declaration, the integer literal `32767` is implicitly converted from **int** to **short**. If the integer literal does not fit into a **short** storage location, a compilation error will occur.

A cast must be used when calling overloaded methods. Consider, for example, the following overloaded methods that use **short** and **int** parameters:

```
public static void MyMethod(int i) {}
public static void MyMethod(short s) {}
```

Using the **short** cast guarantees that the correct type is called, for example:

```
MyMethod(5);         // Calling the method with the int parameter
MyMethod((short)5);  // Calling the method with the short parameter
```

Conversions

There is a predefined implicit conversion from **short** to **int**, **long**, **float**, **double**, or **decimal**.

You cannot implicitly convert nonliteral numeric types of larger storage size to **short** (see Integral Types Table for the storage sizes of integral types). Consider, for example, the following two **short** variables `x` and `y`:

```
short x = 5, y = 12;
```

The following assignment statement will produce a compilation error, because the arithmetic expression on the right-hand side of the assignment operator evaluates to **int** by default.

```
short z = x + y;   // Error: no conversion from int to short
```

To fix this problem, use a cast:

```
short z = (short)(x + y);   // OK: explicit conversion
```

It is possible though to use the following statements, where the destination variable has the same storage size or a larger storage size:

```
int m = x + y;
long n = x + y;
```

There is no implicit conversion from floating-point types to **short**. For example, the following statement generates a compiler error unless an explicit cast is used:

```
short x = 3.0;          // Error: no implicit conversion from double
short y = (short)3.0;   // OK: explicit conversion
```

For information on arithmetic expressions with mixed floating-point types and integral types, see float and double.

For more information on implicit numeric conversion rules, see the Implicit Numeric Conversions Table.

See Also

C# Keywords | Integral Types Table | Built-in Types Table | Implicit Numeric Conversions Table | Explicit Numeric Conversions Table | Int16 Structure

struct

A **struct** type is a value type that can contain constructors, constants, fields, methods, properties, indexers, operators, events, and nested types. The declaration of a **struct** takes the following form:

```
[attributes] [modifiers] struct identifier [:interfaces] body [;]
```

where:

attributes (Optional)
Additional declarative information. For more information on attributes and attribute classes, see 17. Attributes.
modifiers (Optional)
The allowed modifiers are new and the four access modifiers.
identifier
The **struct** name.
interfaces (Optional)
A list that contains the interfaces implemented by the struct, all separated by commas.
body
The struct body that contains member declarations.

Remarks

The **struct** type is suitable for representing lightweight objects such as Point, Rectangle, and Color. Although it is possible to represent a point as a class, a struct is more efficient in some scenarios. For example, if you declare an array of 1000 Point objects, you will allocate additional memory for referencing each object. In this case, the struct is less expensive.

It is an error to declare a default (parameterless) constructor for a struct. A default constructor is always provided to initialize the struct members to their default values.

It is an error to initialize an instance field in a struct.

When you create a struct object using the **new** operator, it gets created and the appropriate constructor is called. Unlike classes, structs can be instantiated without using the **new** operator. If you do not use **new**, the fields will remain unassigned and the object cannot be used until all of the fields are initialized.

There is no inheritance for structs as there is for classes. A struct cannot inherit from another struct or class, and it cannot be the base of a class. Structs, however, inherit from the base class **Object**. A struct can implement interfaces, and it does that exactly as classes do.

Unlike C++, you cannot declare a class using the keyword **struct**. In C#, classes and structs are semantically different. A struct is a value type, while a class is a reference type. For more information on the features of value types, see Value Types.

Unless you need reference type semantics, a class that is smaller than 16 bytes may be more efficiently handled by the system as a struct.

For more information on structs, see 11. Structs and the Structs Tutorial.

In Managed Extensions for C++, the equivalents to a C# class and a C# struct are as follows:

C#	Managed Extensions for C++	For more information
class	__gc struct or __gc class	__gc keyword
struct	__value struct or __value class	__value keyword

Example 1

This example demonstrates **struct** initialization using both default and parameterized constructors.

```
// keyword_struct.cs
// struct declaration and initialization
```

```
using System;
public struct Point
{
   public int x, y;

   public Point(int p1, int p2)
   {
      x = p1;
      y = p2;
   }
}

class MainClass
{
   public static void Main()
   {
      // Initialize:
      Point myPoint = new Point();
      Point yourPoint = new Point(10,10);

      // Display results:
      Console.Write("My Point:   ");
      Console.WriteLine("x = {0}, y = {1}", myPoint.x, myPoint.y);
      Console.Write("Your Point: ");
      Console.WriteLine("x = {0}, y = {1}", yourPoint.x, yourPoint.y);
   }
}
```

Output

```
My Point:   x = 0, y = 0
Your Point: x = 10, y = 10
```

Example 2

This example demonstrates a feature that is unique to structs. It creates a Point object without using the **new** operator. If you replace the word **struct** with the word **class**, the program won't compile.

```
// keyword_struct2.cs
// Declare a struct object without "new"
using System;
public struct Point
{
   public int x, y;

   public Point(int x, int y)
   {
      this.x = x;
      this.y = y;
   }
}

class MainClass
{
   public static void Main()
   {
      // Declare an object:
      Point myPoint;

      // Initialize:
      myPoint.x = 10;
      myPoint.y = 20;

      // Display results:
      Console.WriteLine("My Point:");
```

```
        Console.WriteLine("x = {0}, y = {1}", myPoint.x, myPoint.y);
    }
}
```

Output

```
My Point:
x = 10, y = 20
```

See Also

C# Keywords | Structs Tutorial | Default Values Table | Built-in Types Table | Types | Value Types | class | interface

uint

The **uint** keyword denotes an integral type that stores values according to the size and range shown in the following table.

Type	Range	Size	.NET Framework type
uint	0 to 4,294,967,295	Unsigned 32-bit integer	**System.UInt32**

Literals

You can declare and initialize a variable of the type **uint** like this example:

```
uint myUint = 4294967290;
```

When an integer literal has no suffix, its type is the first of these types in which its value can be represented: **int**, **uint**, **long**, **ulong**. In this example, it is **uint**.

You can also use the suffix u or U, like this:

```
uint myUint = 123U;
```

When you use the suffix U or u, the literal type is determined to be either **uint** or **ulong** according to its size. In this example, it is **uint**.

Conversions

There is a predefined implicit conversion from **uint** to **long**, **ulong**, **float**, **double**, or **decimal**. For example:

```
float myFloat = 4294967290;   // OK: implicit conversion to float
```

There is a predefined implicit conversion from **byte**, **ushort**, or **char** to **uint**. Otherwise you must use a cast. For example, if you have a **long** variable, `myLong`, the following assignment statement will produce a compilation error without a cast:

```
long myLong = 22;
uint myUint = myLong;        // Error: no implicit conversion from long
uint myUint = (uint)myLong;  // OK: explicit conversion
```

Notice also that there is no implicit conversion from floating-point types to **uint**. For example, the following statement generates a compiler error unless an explicit cast is used:

```
uint x = 3.0;         // Error: no implicit conversion from double
uint y = (uint)3.0;   // OK: explicit conversion
```

For information on arithmetic expressions with mixed floating-point types and integral types, see float and double.

For more information on implicit numeric conversion rules, see the Implicit Numeric Conversions Table.

See Also

C# Keywords | Integral Types Table | Built-in Types Table | Implicit Numeric Conversions Table | Explicit Numeric Conversions Table | UInt32 Structure

ulong

The **ulong** keyword denotes an integral type that stores values according to the size and range shown in the following table.

Type	Range	Size	.NET Framework type
ulong	0 to 18,446,744,073,709,551,615	Unsigned 64-bit integer	**System.UInt64**

Literals

You can declare and initialize a **ulong** variable like this example:

```
ulong myUlong = 9223372036854775808;
```

When an integer literal has no suffix, its type is the first of these types in which its value can be represented: **int**, **uint**, **long**, **ulong**. In this example, it is of the type **ulong**.

You can also use suffixes to specify the type of the literal according to the following rules:

- If you use L or l, the type of the literal integer will be either **long** or **ulong** according to its size.

  Note You can use the lowercase letter "l" as a suffix. However, this generates a compiler warning because the letter "l" is easily confused with the digit "1." Use "L" for clarity.

- If you use U or u, the type of the literal integer will be either **uint** or **ulong** according to its size.
- If you use UL, ul, Ul, uL, LU, lu, Lu, or lU, the type of the literal integer will be **ulong**.

  For example, the output of the following three statements will be the system type **UInt64**, which corresponds to the alias **ulong**:

  ```
  Console.WriteLine(9223372036854775808L.GetType());
  Console.WriteLine(123UL.GetType());
  Console.WriteLine((123UL + 456).GetType());
  ```

A common use of the suffix is with calling overloaded methods. Consider, for example, the following overloaded methods that use **ulong** and **int** parameters:

```
public static void MyMethod(int i) {}
public static void MyMethod(ulong l) {}
```

Using a suffix with the **ulong** parameter guarantees that the correct type is called, for example:

```
MyMethod(5);    // Calling the method with the int parameter
MyMethod(5UL);  // Calling the method with the ulong parameter
```

Conversions

There is a predefined implicit conversion from **ulong** to **float**, **double**, or **decimal**.

There is no implicit conversion from **ulong** to any integral type. For example, the following statement will produce a compilation error without an explicit cast:

```
long myLong = 8UL;   // Error: no implicit conversion from ulong
```

There is a predefined implicit conversion from **byte, ushort**, **uint**, or **char** to **ulong**.

Also, there is no implicit conversion from floating-point types to **ulong**. For example, the following statement generates a compiler error unless an explicit cast is used:

```
ulong x = 3.0;         // Error: no implicit conversion from double
```

```
ulong y = (ulong)3.0;  // OK: explicit conversion
```

For information on arithmetic expressions with mixed floating-point types and integral types, see float and double.

For more information on implicit numeric conversion rules, see the Implicit Numeric Conversions Table.

See Also

C# Keywords | Integral Types Table | Built-in Types Table | Implicit Numeric Conversions Table | Explicit Numeric Conversions Table | UInt64 Structure

ushort

The **ushort** keyword denotes an integral data type that stores values according to the size and range shown in the following table.

Type	Range	Size	.NET Framework type
ushort	0 to 65,535	Unsigned 16-bit integer	**System.UInt16**

Literals

You can declare and initialize a **ushort** variable like this example:

```
ushort myShort = 65535;
```

In the preceding declaration, the integer literal `65535` is implicitly converted from **int** to **ushort**. If the integer literal exceeds the range of **ushort**, a compilation error will occur.

A cast must be used when calling overloaded methods. Consider, for example, the following overloaded methods that use **ushort** and **int** parameters:

```
public static void MyMethod(int i) {}
public static void MyMethod(ushort s) {}
```

Using the **ushort** cast guarantees that the correct type is called, for example:

```
MyMethod(5);          // Calling the method with the int parameter
MyMethod((ushort)5);  // Calling the method with the ushort parameter
```

Conversions

There is a predefined implicit conversion from **ushort** to **int**, **uint**, **long**, **ulong**, **float**, **double**, or **decimal**.

There is a predefined implicit conversion from **byte** or **char** to **ushort**. Otherwise a cast must be used to perform an explicit conversion. Consider, for example, the following two **ushort** variables `x` and `y`:

```
ushort x = 5, y = 12;
```

The following assignment statement will produce a compilation error, because the arithmetic expression on the right-hand side of the assignment operator evaluates to **int** by default.

```
ushort z = x + y;   // Error: conversion from int to ushort
```

To fix this problem, use a cast:

```
ushort z = (ushort)(x + y);   // OK: explicit conversion
```

It is possible though to use the following statements, where the destination variable has the same storage size or a larger storage size:

```
int m = x + y;
long n = x + y;
```

Notice also that there is no implicit conversion from floating-point types to **ushort**. For example, the following statement generates a compiler error unless an explicit cast is used:

```
ushort x = 3.0;          // Error: no implicit conversion from double
ushort y = (ushort)3.0;  // OK: explicit conversion
```

For information on arithmetic expressions with mixed floating-point types and integral types, see float and double.

For more information on implicit numeric conversion rules, see the Implicit Numeric Conversions Table.

See Also

C# Keywords | Integral Types Table | Built-in Types Table | Implicit Numeric Conversions Table | Explicit Numeric Conversions Table | UInt16 Structure

Boxing and Unboxing

Boxing and unboxing enable value types to be treated as objects. Value types, including both struct types and built-in types, such as **int**, can be converted to and from the type **object**.

See Also

Reference Types | Value Types

Boxing Conversion

Boxing is an implicit conversion of a value type to the type **object** or to any interface type implemented by this value type. Boxing a value of a value allocates an object instance and copies the value into the new object.

Consider the following declaration of a value-type variable:

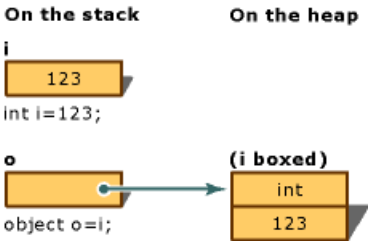
```
int i = 123;
```

The following statement implicitly applies the boxing operation on the variable `i`:

```
object o = i;
```

The result of this statement is creating an object `o`, on the stack, that references a value of the type **int**, on the heap. This value is a copy of the value-type value assigned to the variable `i`. The difference between the two variables, `i` and `o`, is illustrated in the following figure.

Boxing Conversion



It also possible, but never needed, to perform the boxing explicitly as in the following example:

```
int i = 123;
object o = (object) i;
```

Example

This example converts an integer variable `i` to an object `o` via boxing. Then the value stored in the variable `i` is changed from `123` to `456`. The example shows that the object keeps the original copy of the contents, `123`.

```
// boxing.cs
// Boxing an integer variable
using System;
class TestBoxing
{
   public static void Main()
   {
      int i = 123;
      object o = i;  // Implicit boxing
      i = 456;       // Change the contents of i
      Console.WriteLine("The value-type value = {0}", i);
      Console.WriteLine("The object-type value = {0}", o);
   }
}
```

Output

```
The value-type value = 456
The object-type value = 123
```

See Also

Unboxing Conversion

Unboxing is an explicit conversion from the type **object** to a value type or from an interface type to a value type that implements the interface. An unboxing operation consists of:

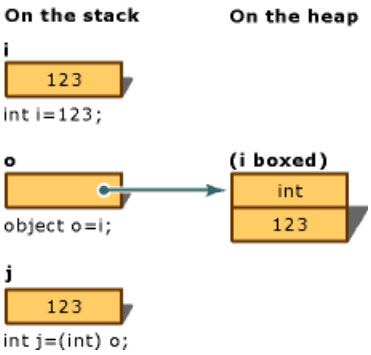
- Checking the object instance to make sure it is a boxed value of the given value type.
- Copying the value from the instance into the value-type variable.

The following statements demonstrate both boxing and unboxing operations:

```
int i = 123;          // A value type
object box = i;       // Boxing
int j = (int)box;     // Unboxing
```

The following figure demonstrates the result of the preceding statements.

Unboxing Conversion



For an unboxing conversion to a given value type to succeed at run time, the value of the source argument must be a reference to an object that was previously created by boxing a value of that value type. If the source argument is **null** or a reference to an incompatible object, an **InvalidCastException** is thrown.

Example

The following example demonstrates a case of invalid unboxing, of how incorrect unboxing leads to InvalidCastException. By using **try** and **catch**, an error message is displayed when the error occurs.

```
using System;
public class UnboxingTest
{
   public static void Main()
   {
      int intI = 123;

      // Boxing
      object o = intI;

      // Reference to incompatible object produces InvalidCastException
      try
      {
         int intJ = (short) o;
         Console.WriteLine("Unboxing OK.");
      }

      catch (InvalidCastException e)
      {
         Console.WriteLine("{0} Error: Incorrect unboxing.",e);
      }
   }
}
```

Output

```
System.InvalidCastException
   at UnboxingTest.Main() Error: Incorrect unboxing.
```

If you change the statement:

```
int intJ = (short) o;
```

to:

```
int intJ = (int) o;
```

the conversion will be performed, and you will get the output `Unboxing OK`.

See Also

Boxing and Unboxing | Boxing Conversion

Reference Types

Variables of reference types, referred to as objects, store references to the actual data. This section introduces the following keywords used to declare reference types:

- class
- interface
- delegate

This section also introduces the following built-in reference types:

- object
- string

See Also

C# Keywords | Types | Value Types

class

Classes are declared using the keyword **class**. It takes the following form::

```
[attributes] [modifiers] class identifier [:base-list] { class-body }[;]
```

where:

attributes (Optional)
Additional declarative information. For more information on attributes and attribute classes, see 17. Attributes.
modifiers (Optional)
The allowed modifiers are new, abstract, sealed, and the four access modifiers.
identifier
The class name.
base-list (Optional)
A list that contains the one base class and any implemented interfaces, all separated by commas.
class-body
Declarations of the class members.

Remarks

Unlike C++, only single inheritance is allowed in C#. In other words, a class can inherit implementation from one base class only. However, a class can implement more that one interface. The following table shows examples of class inheritance and interface implementation:

Inheritance	Example
None	`class ClassA { }`
Single	`class DerivedClass: BaseClass { }`
None, implements two interfaces	`class ImplClass: IFace1, IFace2 { }`
Single, implements one interface	`class ImplDerivedClass: BaseClass, IFace1 { }`

The access levels protected and private are only allowed on nested classes.

A class can contain declarations of the following members:

- Constructors (see Constructors and Destructors, 10.10 Instance constructors)
- Destructors (see Destructors, 10.12 Destructors)
- Constants (see 10.3 Constants)
- Fields (see 10.4 Fields)
- Methods (see 10.5 Methods)
- Properties (see Properties, 10.6 Properties)
- Indexers (see Indexers, 10.8 Indexers)
- Operators (see C# Operators, 10.9 Operators)
- Events (see event, 10.7 Events)
- Delegates (see delegate, 15. Delegates)
- Classes (see 10. Classes)
- Interfaces (see interface, 13. Interfaces)
- Structs (see struct, 11. Structs)

In Managed Extensions for C++, the equivalents to a C# class and a C# struct are as follows:

C#	Managed Extensions for C++	For more information
class	__gc struct or __gc class	__gc keyword
struct	__value struct or __value class	__value keyword

Example

The following example demonstrates declaring class fields, constructors, and methods. It also demonstrates object instantiation and printing instance data. In this example, two classes are declared, the `Kid` class, which contains two private fields (`name` and `age`) and two public methods. The second class is `MainClass`, used for data processing.

```
// keyword_class.cs
// class example
using System;
public class Kid
{
   private int age;
   private string name;

   // Default constructor:
   public Kid()
   {
      name = "N/A";
   }

   // Constructor:
   public Kid(string name, int age)
   {
      this.name = name;
      this.age = age;
   }

   // Printing method:
   public void PrintKid()
   {
      Console.WriteLine("{0}, {1} years old.", name, age);
   }
}

public class MainClass
{
   public static void Main()
   {
      // Create objects
      // Objects must be created using the new operator:
      Kid kid1 = new Kid("Craig", 11);
      Kid kid2 = new Kid("Sally", 10);

      // Create an object using the default constructor:
      Kid kid3 = new Kid();

      // Display results:
      Console.Write("Kid #1: ");
      kid1.PrintKid();
      Console.Write("Kid #2: ");
      kid2.PrintKid();
      Console.Write("Kid #3: ");
      kid3.PrintKid();
   }
}
```

Output

```
Kid #1: Craig, 11 years old.
Kid #2: Sally, 10 years old.
Kid #3: N/A, 0 years old.
```

Notice, in the preceding example, that the private fields (`name` and `age`) can only be accessed through the public methods of the `Kid` class. For example, you cannot print the kid's name, from the `Main` method, using a statement like this:

```
Console.Write(kid1.name);   // Error
```

However, it is possible to do that if the `Main` method is a member of the `Kid` class.

Notice also that the **private** modifier is the default modifier for class members. If you remove the modifier, you still get private members. For more information on access level modifiers, see Access Modifiers.

Notice also that for the object created using the default constructor (`kid3`), the age field was initialized to zero.

For more information on classes, see 10. Classes.

See Also

C# Keywords | Reference Types | C. Grammar | interface | struct

delegate

A **delegate** declaration defines a reference type that can be used to encapsulate a method with a specific signature. A delegate instance encapsulates a static or an instance method. Delegates are roughly similar to function pointers in C++; however, delegates are type-safe and secure.

This declaration takes the following form::

```
[attributes] [modifiers] delegate result-type identifier ([formal-parameters]);
```

where:

attributes (Optional)
Additional declarative information. For more information on attributes and attribute classes, see 17. Attributes.
modifiers (Optional)
The allowed modifiers are new and the four access modifiers.
result-type
The result type, which matches the return type of the method.
identifier
The delegate name.
formal-parameters (Optional)
Parameter list. If a parameter is a pointer, the delegate must be declared with the unsafe modifier.

Remarks

A delegate lets you pass a function as a parameter. The type safety of delegates requires the function you pass as a delegate to have the same signature as the delegate declaration. See the Delegates Tutorial for more information on using delegates.

The Delegates Tutorial shows how to compose delegates, that is, create delegates from other delegates. A delegate that contains an out parameter cannot be composed.

Delegates are the basis for events.

For more information on delegates, see 15. Delegates.

Example 1

The following is a simple example of declaring and using a delegate.

```
// keyword_delegate.cs
// delegate declaration
delegate void MyDelegate(int i);

class Program
{
   public static void Main()
   {
      TakesADelegate(new MyDelegate(DelegateFunction));
   }

   public static void TakesADelegate(MyDelegate SomeFunction)
   {
      SomeFunction(21);
   }

   public static void DelegateFunction(int i)
   {
      System.Console.WriteLine("Called by delegate with number: {0}.", i);
   }
}
```

Output

```
Called by delegate with number: 21.
```

Example 2

In the following example, one delegate is mapped to both static and instance methods and returns specific information from each.

```
// keyword_delegate2.cs
// Calling both static and instance methods from delegates
using System;

// delegate declaration
delegate void MyDelegate();

public class MyClass
{
   public void InstanceMethod()
   {
      Console.WriteLine("A message from the instance method.");
   }

   static public void StaticMethod()
   {
      Console.WriteLine("A message from the static method.");
   }
}

public class MainClass
{
   static public void Main()
   {
      MyClass p = new MyClass();

      // Map the delegate to the instance method:
      MyDelegate d = new MyDelegate(p.InstanceMethod);
      d();

      // Map to the static method:
      d = new MyDelegate(MyClass.StaticMethod);
      d();
   }
}
```

Output

```
A message from the instance method.
A message from the static method.
```

See Also

C# Keywords | Reference Types | C. Grammar | Delegates Tutorial

interface

An interface defines a contract. A class or struct that implements an interface must adhere to its contract. The declaration takes the following form::

```
[attributes] [modifiers] interface identifier [:base-list] {interface-body}[;]
```

where:

attributes (Optional)
Additional declarative information. For more information on attributes and attribute classes, see 17. Attributes.
modifiers (Optional)
The allowed modifiers are new and the four access modifiers.
identifier
The interface name.
base-list (Optional)
A list that contains one or more explicit base interfaces separated by commas.
interface-body
Declarations of the interface members.

Remarks

An interface can be a member of a namespace or a class and can contain signatures of the following members:

- Methods
- Properties
- Indexers
- Events

An interface can inherit from one or more base interfaces. In the following example, the interface `IMyInterface` inherits from two base interfaces, `IBase1` and `IBase2`:

```
interface IMyInterface: IBase1, IBase2
{
   void MethodA();
   void MethodB();
}
```

Interfaces can be implemented by classes and structs. The identifier of the implemented interface appears in the class base list. For example:

```
class Class1: Iface1, Iface2
{
   // class members
}
```

When a class base list contains a base class and interfaces, the base class comes first in the list. For example:

```
class ClassA: BaseClass, Iface1, Iface2
{
   // class members
}
```

For more information on interfaces, see Interfaces.

For more information on properties and indexers, see Property Declaration and Indexer Declaration.

Example

The following example demonstrates interface implementation. In this example, the interface `IPoint` contains the property

declaration, which is responsible for setting and getting the values of the fields. The class `MyPoint` contains the property implementation.

```
// keyword_interface.cs
// Interface implementation
using System;
interface IPoint
{
   // Property signatures:
   int x
   {
      get;
      set;
   }

   int y
   {
      get;
      set;
   }
}

class MyPoint : IPoint
{
   // Fields:
   private int myX;
   private int myY;

   // Constructor:
   public MyPoint(int x, int y)
   {
      myX = x;
      myY = y;
   }

   // Property implementation:
   public int x
   {
      get
      {
         return myX;
      }

      set
      {
         myX = value;
      }
   }

   public int y
   {
      get
      {
         return myY;
      }
      set
      {
         myY = value;
      }
   }
}

class MainClass
{
   private static void PrintPoint(IPoint p)
   {
      Console.WriteLine("x={0}, y={1}", p.x, p.y);
   }
```

```
    public static void Main()
    {
       MyPoint p = new MyPoint(2,3);
       Console.Write("My Point: ");
       PrintPoint(p);
    }
}
```

Output

```
My Point: x=2, y=3
```

See Also

C# Keywords | Reference Types | Properties | Indexers | C. Grammar | Explicit Interface Implementation Tutorial | class | struct

object

The **object** type is an alias for **System.Object** in the .NET Framework. You can assign values of any type to variables of type **object**.

All data types, predefined and user-defined, inherit from the **System.Object** class. The **object** data type is the type to and from which objects are boxed.

Example

The following sample shows how variables of type **object** can accept values of any data type and how variables of type **object** can use methods on **System.Object** from the .NET Framework.

```
// keyword_object.cs
using System;
public class MyClass1
{
   public int i = 10;
}

public class MyClass2
{
   public static void Main()
   {
      object a;
      a = 1;   // an example of boxing
      Console.WriteLine(a);
      Console.WriteLine(a.GetType());
      Console.WriteLine(a.ToString());
      Console.WriteLine();

      a = new MyClass1 ();
      MyClass1 ref_MyClass1;
      ref_MyClass1 = (MyClass1)a;
      Console.WriteLine(ref_MyClass1.i);
   }
}
```

Output

```
1
System.Int32
1

10
```

See Also

C# Keywords | Reference Types | Value Types

string

The **string** type represents a string of Unicode characters. **string** is an alias for System.String in the .NET Framework.

Although **string** is a reference type, the equality operators (== and !=) are defined to compare the *values* of **string** objects, not references (7.9.7 String equality operators). This makes testing for string equality more intuitive.

```
string a = "hello";
string b = "h";
b += "ello"; // append to b
Console.WriteLine( a == b );          // output: True -- same value
Console.WriteLine( (object)a == b );  // False -- different objects
```

The + operator concatenates strings:

```
string a = "good " + "morning";
```

The [] operator accesses individual characters of a **string**:

```
char x = "test"[2];  // x = 's';
```

String literals are of type **string** and can be written in two forms, quoted and @-quoted. Quoted string literals are enclosed in double quotation marks ("):

```
"good morning"  // a string literal
```

and can contain any character literal, including escape sequences:

```
string a = "\\\u0066\n";  // backslash, letter f, new line
```

> **Note** The escape code \u*dddd* (where *dddd* is a four-digit number) represents the Unicode character U+*dddd*. Eight-digit Unicode escape codes are also recognized: \u*dddd*\u*dddd*.

@-quoted string literals start with @ and are enclosed in double quotation marks. For example:

```
@"good morning"  // a string literal
```

The advantage of @-quoting is that escape sequences are *not* processed, which makes it easy to write, for example, a fully qualified file name:

```
@"c:\Docs\Source\a.txt"  // rather than "c:\\Docs\\Source\\a.txt"
```

To include a double quotation mark in an @-quoted string, double it:

```
@"""Ahoy!"" cried the captain." // "Ahoy!" cried the captain.
```

Another use of the @ symbol is to use referenced (/reference) identifiers that happen to be C# keywords. For more information, see 2.4.2 Identifiers.

Example

```
// keyword_string.cs
using System;
class test
{
   public static void Main( String[] args )
   {
```

```
        string a = "\u0068ello ";
        string b = "world";
        Console.WriteLine( a + b );
        Console.WriteLine( a + b == "hello world" );
    }
}
```

Output

```
hello world
True
```

See Also

C# Keywords | Reference Types | Value Types | Formatting Numeric Results Table

void

When used as the return type for a method, **void** specifies that the method does not return a value.

void is not allowed in a method's parameter list. A method that takes no parameters and returns no value is declared as follows:

```
void MyMethod();
```

void is also used in an unsafe context to declare a pointer to an unknown type. For more information, see A.2 Pointer types.

See Also

C# Keywords | Reference Types | Value Types

Types Reference Tables

The following reference tables summarize the C# types:

- Built-in types
- Integral types
- Floating-point types
- Default values
- Value types
- Implicit numeric conversions
- Explicit numeric conversions

For information on formatting the output of numeric types, see Formatting Numeric Results Table.

See Also

Reference Types | Value Types

Built-in Types Table

The following table shows the keywords for built-in C# types, which are aliases of predefined types in the **System** namespace.

C# Type	.NET Framework type
bool	**System.Boolean**
byte	**System.Byte**
sbyte	**System.SByte**
char	**System.Char**
decimal	**System.Decimal**
double	**System.Double**
float	**System.Single**
int	**System.Int32**
uint	**System.UInt32**
long	**System.Int64**
ulong	**System.UInt64**
object	**System.Object**
short	**System.Int16**
ushort	**System.UInt16**
string	**System.String**

Remarks

All of the types in the table, except **object** and **string**, are referred to as simple types.

The C# type keywords and their aliases are interchangeable. For example, you can declare an integer variable by using either of the following declarations:

```
int x = 123;
System.Int32 x = 123;
```

To display the actual type for any C# type use the system method `GetType()`. For example, the following statement displays the system alias that represents the type of `myVariable`:

```
Console.WriteLine(myVariable.GetType());
```

You can also use the typeof operator.

See Also

C# Keywords | Value Types | Default Values Table | Formatting Numeric Results Table | Types Reference Tables

Integral Types Table

The following table shows the sizes and ranges of the integral types, which constitute a subset of simple types.

Type	Range	Size
sbyte	-128 to 127	Signed 8-bit integer
byte	0 to 255	Unsigned 8-bit integer
char	U+0000 to U+ffff	Unicode 16-bit character
short	-32,768 to 32,767	Signed 16-bit integer
ushort	0 to 65,535	Unsigned 16-bit integer
int	-2,147,483,648 to 2,147,483,647	Signed 32-bit integer
uint	0 to 4,294,967,295	Unsigned 32-bit integer
long	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	Signed 64-bit integer
ulong	0 to 18,446,744,073,709,551,615	Unsigned 64-bit integer

Remarks

If the value represented by an integer literal exceeds the range of **ulong**, a compilation error will occur.

See Also

C# Keywords | Built-in Types Table | Floating-Point Types Table | Default Values Table | Formatting Numeric Results Table | Types Reference Tables

Floating-Point Types Table

The following table shows the precision and approximate ranges for the floating-point types.

Type	Approximate range	Precision
float	$\pm 1.5 \times 10^{-45}$ to $\pm 3.4 \times 10^{38}$	7 digits
double	$\pm 5.0 \times 10^{-324}$ to $\pm 1.7 \times 10^{308}$	15-16 digits

See Also

Default Values Table | Built-in Types Table | Integral Types Table | Formatting Numeric Results Table | Types Reference Tables | decimal

Default Values Table

The following table shows the default values of value types returned by the default constructors. Default constructors are invoked by using the **new** operator, for example:

```
int myInt = new int();
```

The preceding statement has the same effect as the following statement:

```
int myInt = 0;
```

Remember that using uninitialized variables in C# is not allowed.

Value type	Default value
bool	**false**
byte	0
char	'\0'
decimal	0.0M
double	0.0D
enum	The value produced by the expression (E)0, where E is the enum identifier.
float	0.0F
int	0
long	0L
sbyte	0
short	0
struct	The value produced by setting all value-type fields to their default values and all reference-type fields to **null**.
uint	0
ulong	0
ushort	0

See Also

Value Types | Built-in Types Table | Types Reference Tables

Value Types Table

The following table lists the C# value types by category.

Value type	Category
bool	Boolean
byte	Unsigned, numeric, integral
char	Unsigned, numeric, integral
decimal	Numeric, decimal
double	Numeric, floating-point
enum	Enumeration
float	Numeric, floating-point
int	Signed, numeric, integral
long	Signed, numeric, integral
sbyte	Signed, numeric, integral
short	Signed, numeric, integral
struct	User-defined structure
uint	Unsigned, numeric, integral
ulong	Unsigned, numeric, integral
ushort	Unsigned, numeric, integral

See Also

Value Types | Formatting Numeric Results Table | Types Reference Tables

Implicit Numeric Conversions Table

The following table shows the predefined implicit numeric conversions. Implicit conversions might occur in many situations, including method invoking and assignment statements.

From	To
sbyte	**short**, **int**, **long**, **float**, **double**, or **decimal**
byte	**short**, **ushort**, **int**, **uint**, **long**, **ulong**, **float**, **double**, or **decimal**
short	**int**, **long**, **float**, **double**, or **decimal**
ushort	**int**, **uint**, **long**, **ulong**, **float**, **double**, or **decimal**
int	**long**, **float**, **double**, or **decimal**
uint	**long**, **ulong**, **float**, **double**, or **decimal**
long	**float**, **double**, or **decimal**
char	**ushort**, **int**, **uint**, **long**, **ulong**, **float**, **double**, or **decimal**
float	**double**
ulong	**float**, **double**, or **decimal**

Remarks

- The conversions from **int**, **uint**, or **long** to **float** and from **long** to **double** may cause a loss of precision, but not a loss of magnitude.
- There are no implicit conversions to the **char** type.
- There are no implicit conversions between floating-point types and the **decimal** type.
- A constant expression of type **int** can be converted to **sbyte**, **byte**, **short**, **ushort**, **uint**, or **ulong**, provided the value of the constant expression is within the range of the destination type.

For more information on implicit conversion, see 6.1 Implicit conversions.

For more information on constant expressions, see 7.15 Constant expressions.

See Also

Integral Types Table | Built-in Types Table | Explicit Numeric Conversions Table

Explicit Numeric Conversions Table

Explicit numeric conversion is used to convert any numeric type to any other numeric type, for which there is no implicit conversion, by using a cast expression. The following table shows these conversions.

From	To
sbyte	**byte**, **ushort**, **uint**, **ulong**, or **char**
byte	**sbyte** or **char**
short	**sbyte**, **byte**, **ushort**, **uint**, **ulong**, or **char**
ushort	**sbyte**, **byte**, **short**, or **char**
int	**sbyte**, **byte**, **short**, **ushort**, **uint**, **ulong**, or **char**
uint	**sbyte**, **byte**, **short**, **ushort**, **int**, or **char**
long	**sbyte**, **byte**, **short**, **ushort**, **int**, **uint**, **ulong**, or **char**
ulong	**sbyte**, **byte**, **short**, **ushort**, **int**, **uint**, **long**, or **char**
char	**sbyte**, **byte**, or **short**
float	**sbyte**, **byte**, **short**, **ushort**, **int**, **uint**, **long**, **ulong**, **char**, or **decimal**
double	**sbyte**, **byte**, **short**, **ushort**, **int**, **uint**, **long**, **ulong**, **char**, **float**, or **decimal**
decimal	**sbyte**, **byte**, **short**, **ushort**, **int**, **uint**, **long**, **ulong**, **char**, **float**, or **double**

Remarks

- The explicit numeric conversion may cause loss of precision or result in throwing exceptions.
- When you convert a **decimal** value to an integral type, this value is rounded towards zero to the nearest integral value. If the resulting integral value is outside the range of the destination type, an **OverflowException** is thrown.
- When you convert from a **double** or **float** value to an integral type, the value is rounded towards zero to the nearest integral value. If the resulting integral value is outside the range of the destination value, the result depends on the overflow checking context. In a checked context, an **OverflowException** is thrown, while in an unchecked context, the result is an unspecified value of the destination type.
- When you convert **double** to **float**, the **double** value is rounded to the nearest **float** value. If the **double** value is too small or too large to fit into the destination type, the result will be zero or infinity.
- When you convert **float** or **double** to **decimal**, the source value is converted to **decimal** representation and rounded to the nearest number after the 28th decimal place if required. Depending on the value of the source value, one of the following results may occur:
    - If the source value is too small to be represented as a **decimal**, the result becomes zero.
    - If the source value is NaN (not a number), infinity, or too large to be represented as a **decimal**, an **OverflowException** is thrown.
- When you convert **decimal** to **float** or **double**, the **decimal** value is rounded to the nearest **double** or **float** value.

For more information on explicit conversion, see 6.2 Explicit conversions.

See Also

Integral Types Table | Built-in Types Table | Implicit Numeric Conversions Table

Formatting Numeric Results Table

You can format numeric results by using the **String.Format** method, or through the **Console.Write** method, which calls **String.Format**. The format is specified using format strings. The following table contains the supported standard format strings. The format string takes the following form: *Axx*, where *A* is the *format specifier* and *xx* is the *precision specifier*. The format specifier controls the type of formatting applied to the numerical value, and the precision specifier controls the number of significant digits or decimal places of the formatted output.

For more information on standard and custom formatting strings, see Formatting Overview. For more information on the **String.Format** method, see String.Format Method.

Character	Description	Examples	Output
C or c	Currency	Console.Write("{0:C}", 2.5); Console.Write("{0:C}", -2.5);	$2.50 ($2.50)
D or d	Decimal	Console.Write("{0:D5}", 25);	00025
E or e	Scientific	Console.Write("{0:E}", 250000);	2.500000E+005
F or f	Fixed-point	Console.Write("{0:F2}", 25); Console.Write("{0:F0}", 25);	25.00 25
G or g	General	Console.Write("{0:G}", 2.5);	2.5
N or n	Number	Console.Write("{0:N}", 2500000);	2,500,000.00
X or x	Hexadecimal	Console.Write("{0:X}", 250); Console.Write("{0:X}", 0xffff);	FA FFFF

See Also

Types Reference Tables | string

Modifiers

Modifiers are used to modify declarations of types and type members. This section introduces the C# modifiers:

Modifier	Purpose
Access Modifiers • public • private • internal • protected	Specify the declared accessibility of types and type members.
abstract	Indicate that a class is intended only to be a base class of other classes.
const	Specify that the value of the field or the local variable cannot be modified.
event	Declare an event.
extern	Indicate that the method is implemented externally.
override	Provide a new implementation of a virtual member inherited from a base class.
readonly	Declare a field that can only be assigned values as part of the declaration or in a construct or in the same class.
sealed	Specify that a class cannot be inherited.
static	Declare a member that belongs to the type itself rather than to a specific object.
unsafe	Declare an unsafe context.
virtual	Declare a method or an accessor whose implementation can be changed by an overriding member in a derived class.
volatile	Indicate that a field can be modified in the program by something such as the operating s ystem, the hardware, or a concurrently executing thread.

See Also

C# Keywords

Access Modifiers

Access modifiers are keywords used to specify the declared accessibility of a member or a type. This section introduces the four access modifiers:

- public
- protected
- internal
- private

The following five accessibility levels can be specified using the access modifiers:

public **protected** **internal** **internal protected** **private**

This section also introduces the following topics:

- Accessibility Levels
- Accessibility Domain
- Restrictions on Using Accessibility Levels

See Also

C# Keywords | Modifiers | 3.5 Member access | 10.2.3 Access modifiers

Accessibility Levels

When access is allowed to a member, it said to be accessible. Otherwise, it is inaccessible. Use the access modifiers, public, protected, internal, or private, to specify one of the following declared accessibilities for members.

Declared accessibility	Meaning
public	Access is not restricted.
protected	Access is limited to the containing class or types derived from the containing class.
internal	Access is limited to the current assembly.
protected internal	Access is limited to the current assembly or types derived from the containing class .
private	Access is limited to the containing type.

Only one access modifier is allowed for a member or type, except for the **protected internal** combination.

Access modifiers are not allowed on namespaces. Namespaces have no access restrictions.

Depending on the context in which a member declaration takes place, only certain declared accessibilities are permitted. If no access modifier is specified in a member declaration, a default accessibility is used.

Top-level types, which are not nested into other types, can only have **internal** or **public** accessibility. The default accessibility for these types is **internal**.

Nested types, which are members of other types, can have declared accessibilities as indicated in the following table.

Members of	Default member accessibility	Allowed declared accessibility of the member
enum	**public**	None
class	**private**	**public** **protected** **internal** **private** **protected internal**
interface	**public**	None
struct	**private**	**public** **internal** **private**

The accessibility of a nested type depends on its accessibility domain, which is determined by both the declared accessibility of the member and the accessibility domain of the immediately containing type. However, the accessibility domain of a nested type cannot exceed that of the containing type.

See Also

Access Modifiers | Accessibility Domain | Restrictions on Using Accessibility Levels | 3.5 Member access | 3.5.1 Declared accessibility

Accessibility Domain

The accessibility domain of a member specifies where, in the program sections, a member can be referenced. If the member is nested within another type, its accessibility domain is determined by both the accessibility level of the member and the accessibility domain of the immediately containing type.

The accessibility domain of a top-level type is at least the program text of the project in which it is declared. That is, the entire source files of this project. The accessibility domain of a nested type is at least the program text of the type in which it is declared. That is, the type body, including any nested types. The accessibility domain of a nested type never exceeds that of the containing type. These concepts are demonstrated in the following example.

For more information, see 3.5.2 Accessibility domains.

Example

This example contains a top-level type, `T1`, and two nested classes, `M1` and `M2`. The classes contain fields with different declared accessibilities. In the `Main` method, a comment follows each statement to indicate the accessibility domain of each member. Notice that the statements that attempt to reference the inaccessible members are commented out. If you want to see the compiler errors caused by referencing an inaccessible member, remove the comments one at a time.

```
// cs_Accessibility_Domain.cs
using System;
namespace MyNameSpace
{
   public class T1
   {
      public static int myPublicInt;
      internal static int myInternalInt;
      private static int myPrivateInt = 0;

      public class M1
      {
         public static int myPublicInt;
         internal static int myInternalInt;
         private static int myPrivateInt = 0;
      }

      private class M2
      {
         public static int myPublicInt = 0;
         internal static int myInternalInt = 0;
         private static int myPrivateInt = 0;
      }
   }

   public class MainClass
   {
      public static int Main()
      {
         // Access to T1 fields:
         T1.myPublicInt = 1;      // Access is unlimited
         T1.myInternalInt = 2;    // Accessible only in current assembly
         // T1.myPrivateInt = 3;  // Error: inaccessible outside T1

         // Access to the M1 fields:
         T1.M1.myPublicInt = 1;     // Access is unlimited
         T1.M1.myInternalInt = 2;   // Accessible only in current assembly
         // T1.M1.myPrivateInt = 3; // Error: inaccessible outside M1

         // Access to the M2 fields:
         // T1.M2.myPublicInt = 1;   // Error: inaccessible outside T1
         // T1.M2.myInternalInt = 2; // Error: inaccessible outside T1
         // T1.M2.myPrivateInt = 3;  // Error: inaccessible outside M2

         return 0;
      }
```

```
    }
}
```

See Also

Access Modifiers | Accessibility Levels | Restrictions on Using Accessibility Levels | 10.2.3 Access modifiers | 3.5.2 Accessibility domains

Restrictions on Using Accessibility Levels

When you declare a type, it is essential to see if that type has to be *at least as accessible as* another member or type. For example, the direct base class must be at least as accessible as the derived class. The following declarations will result in a compiler error, because the base class `BaseClass` is less accessible than `MyClass`:

```
class BaseClass {...}
public class MyClass: BaseClass {...} // Error
```

The following table summarizes the restrictions on using declared accessibility levels.

Context	Remarks
Classes	The direct base class of a class type must be at least as accessible as the class type itself.
Interfaces	The explicit base interfaces of an interface type must be at least as accessible as the interface type itself.
Delegates	The return type and parameter types of a delegate type must be at least as accessible as the delegate type itself.
Constants	The type of a constant must be at least as accessible as the constant itself.
Fields	The type of a field must be at least as accessible as the field itself.
Methods	The return type and parameter types of a method must be at least as accessible as the method itself.
Properties	The type of a property must be at least as accessible as the property itself.
Events	The type of an event must be at least as accessible as the event itself.
Indexers	The type and parameter types of an indexer must be at least as accessible as the indexer itself.
Operators	The return type and parameter types of an operator must be at least as accessible as the operator itself.
Constructors	The parameter types of a constructor must be at least as accessible as the constructor itself.

Example

The following example contains erroneous declarations of different types. The comment following each declaration indicates the expected compiler error.

```
// Restrictions_on_Using_Accessibility_Levels.cs
// CS0052 expected as well as CS0053, CS0056, and CS0057
// To make the program work, change access level of both class B
// and MyPrivateMethod() to public.

using System;

// A delegate:
delegate int MyDelegate();

class B
{
   // A private method:
   static int MyPrivateMethod()
   {
      return 0;
   }
}

public class A
{
   // Fields:
   public B myField = new B();   // Error: The type B is less accessible
                                 // than the field A.myField.
   // Constants:
   public readonly B myConst = new B();   // Error: The type B is less accessible
                                          // than the constant A.myConst.

   // Methods:
```

```
    public B MyMethod()
    {
       return new B();   // Error: The type B is less accessible
    }                    // than the method A.MyMethod.

    // Properties:
    public  B MyProp
    {
       set
       {
       }
    }   // Error: The type B is less accessible than the property A.MyProp

    // Delegates:
    MyDelegate d = new MyDelegate(B.MyPrivateMethod);
    // Even when B is declared public, you still get the error:
    // "The parameter B.MyPrivateMethod is not accessible due to
    // protection level."

    // Operators:
    public static B operator + (A m1, B m2)
    {
       return new B();   // Error: The type B is less accessible
                         // than the operator A.operator +(A,B)
    }

    static void Main()
    {
       Console.Write("Compiled successfully");
    }
  }
```

See Also

Access Modifiers | Accessibility Domain | Accessibility Levels | 3.5 Member access

internal

The **internal** keyword is an access modifier for types and type members. Internal members are accessible only within files in the same assembly. For more information on assemblies, see Components and Assemblies.

A common use of internal access is in component-based development because it enables a group of components to cooperate in a private manner without being exposed to the rest of the application code. For example, a framework for building graphical user interfaces could provide Control and Form classes that cooperate using members with internal access. Since these members are internal, they are not exposed to code that is using the framework.

It is an error to reference a member with internal access outside the assembly within which it was defined.

> **Caution** An **internal virtual** method can be overridden in some languages, such as textual Microsoft intermediate language (MSIL) using Ilasm.exe, even though it cannot be overridden using C#.

For a comparison of **internal** with the other access modifiers, see Accessibility Levels.

Example

This example contains two files, `Assembly1.cs` and `Assembly2.cs`. The first file contains an internal base class, `BaseClass`. In the second file, an attempt to access the member of the base class will produce an error.

File Assembly1.cs:

```
// Assembly1.cs
// compile with: /target:library
internal class BaseClass
{
   public static int IntM = 0;
}
```

File Assembly2.cs

```
// Assembly2.cs
// compile with: /reference:Assembly1.dll
// CS0122 expected
class TestAccess
{
   public static void Main()
   {
      BaseClass myBase = new BaseClass();   // error, BaseClass not visible outside assembly
   }
}
```

See Also

C# Keywords | Access Modifiers | Accessibility Levels | Modifiers | 3.5.1 Declared accessibility

private

The **private** keyword is a member access modifier. Private access is the least permissive access level. Private members are accessible only within the body of the class or the struct in which they are declared.

Nested types in the same body can also access those private members.

It is a compile-time error to reference a private member outside the class or the struct in which it is declared.

For a comparison of **private** with the other access modifiers, see Accessibility Levels.

Example

In this example, the `Employee` class contains a public member, `Name`, and a private member, `Salary`. The public member can be accessed directly, while the private member must be accessed through the public method `AccessSalary()`.

```
// private_keyword.cs
using System;
class Employee
{
   public string name = "xx";
   double salary = 100.00;   // private access by default
   public double AccessSalary() {
      return salary;
   }
}

class MainClass
{
   public static void Main()
   {
      Employee e = new Employee();

      // Accessing the public field:
      string n = e.name;

      // Accessing the private field:
      double s = e.AccessSalary();
   }
}
```

In the preceding example, if you attempt to access the private members directly by using a statement like this:

```
double s = e.salary;
```

you will get the error message:

```
'Employee.Salary' is inaccessible due to its protection level.
```

See Also

C# Keywords | Access Modifiers | Accessibility Levels | Modifiers | 3.5.1 Declared accessibility

protected

The **protected** keyword is a member access modifier. A protected member is accessible from within the class in which it is declared, and from within any class derived from the class that declared this member.

A protected member of a base class is accessible in a derived class only if the access takes place through the derived class type. For example, consider the following code segment:

```
class A
{
   protected int x = 123;
}

class B : A
{
   void F()
   {
      A a = new A();
      B b = new B();
      a.x = 10;   // Error
      b.x = 10;   // OK
   }
}
```

The statement `a.x =10` generates an error because A is not derived from B.

Struct members cannot be protected because the struct cannot be inherited.

It is an error to reference a protected member from a class, which is not derived from the protected member's class.

For more information on protected members, see 3.5.3 Protected access for instance members.

For a comparison of **protected** with the other access modifiers, see Accessibility Levels.

Example

In this example, the class `MyDerivedC` is derived from `MyClass`; therefore, you can access the protected members of the base class directly from the derived class.

```
// protected_keyword.cs
using System;
class MyClass
{
   protected int x;
   protected int y;
}

class MyDerivedC: MyClass
{
   public static void Main()
   {
      MyDerivedC mC = new MyDerivedC();

      // Direct access to protected members:
      mC.x = 10;
      mC.y = 15;
      Console.WriteLine("x = {0}, y = {1}", mC.x, mC.y);
   }
}
```

Output

```
x = 10, y = 15
```

If you change the access levels of `x` and `y` to **private**, the compiler will issue the error messages:

```
'MyClass.y' is inaccessible due to its protection level.
'MyClass.x' is inaccessible due to its protection level.
```

See Also

C# Keywords | Access Modifiers | Accessibility Levels | Modifiers | 3.5.1 Declared accessibility

public

The **public** keyword is an access modifier for types and type members. Public access is the most permissive access level. There are no restrictions on accessing public members.

For a comparison of **public** with the other access modifiers, see Accessibility Levels.

Example

In the following example, two classes are declared, `MyClass1` and `MyClass2`. The public members x and y of the `MyClass1` are accessed directly from `MyClass2`.

```
// protected_public.cs
// Public access
using System;
class MyClass1
{
   public int x;
   public int y;
}

class MyClass2
{
   public static void Main()
   {
      MyClass1 mC = new MyClass1();

      // Direct access to public members:
      mC.x = 10;
      mC.y = 15;
      Console.WriteLine("x = {0}, y = {1}", mC.x, mC.y);
   }
}
```

Output

```
x = 10, y = 15
```

If you change the **public** access level to **private** or **protected**, you will get the error message:

```
'MyClass1.y' is inaccessible due to its protection level.
```

See Also

C# Keywords | Access Modifiers | Accessibility Levels | Modifiers | 3.5.1 Declared accessibility

abstract

The **abstract** modifier can be used with classes, methods, properties, indexers, and events.

Use the **abstract** modifier in a class declaration to indicate that a class is intended only to be a base class of other classes.

Abstract classes have the following features:

- An abstract class cannot be instantiated.
- An abstract class may contain abstract methods and accessors.
- It is not possible to modify an abstract class with the sealed modifier, which means that the class cannot be inherited.
- A non-abstract class derived from an abstract class must include actual implementations of all inherited abstract methods and accessors.

Use the **abstract** modifier in a method or property declaration to indicate that the method or property does not contain implementation.

Abstract methods have the following features:

- An abstract method is implicitly a virtual method.
- Abstract method declarations are only permitted in abstract classes.
- Because an abstract method declaration provides no actual implementation, there is no method body; the method declaration simply ends with a semicolon and there are no braces ({ }) following the signature. For example:

  ```
  public abstract void MyMethod();
  ```

- The implementation is provided by an overriding method, which is a member of a non-abstract class.
- It is an error to use the **static** or **virtual** modifiers in an abstract method declaration.

Abstract properties behave like abstract methods, except for the differences in declaration and invocation syntax.

- It is an error to use the **abstract** modifier on a static property.
- An abstract inherited property can be overridden in a derived class by including a property declaration that uses the **override** modifier.

An abstract class must provide implementation for all interface members.

An abstract class that implements an interface might map the interface methods onto abstract methods. For example:

```
interface I
{
   void M();
}
abstract class C: I
{
   public abstract void M();
}
```

For more information, see 10.1.1.1 Abstract classes and 10.5.6 Abstract methods.

Example

In this example, the class `MyDerivedC` is derived from an abstract class `MyBaseC`. The abstract class contains an abstract method, `MyMethod()`, and two abstract properties, `GetX()` and `GetY()`.

```
// abstract_keyword.cs
// Abstract Classes
using System;
abstract class MyBaseC   // Abstract class
{
   protected int x = 100;
   protected int y = 150;
   public abstract void MyMethod();   // Abstract method
```

```
    public abstract int GetX   // Abstract property
    {
       get;
    }

    public abstract int GetY   // Abstract property
    {
       get;
    }
 }

 class MyDerivedC: MyBaseC
 {
    public override void MyMethod()
    {
       x++;
       y++;
    }

    public override int GetX   // overriding property
    {
       get
       {
          return x+10;
       }
    }

    public override int GetY   // overriding property
    {
       get
       {
          return y+10;
       }
    }

    public static void Main()
    {
       MyDerivedC mC = new MyDerivedC();
       mC.MyMethod();
       Console.WriteLine("x = {0}, y = {1}", mC.GetX, mC.GetY);
    }
 }
```

Output

```
x = 111, y = 161
```

In the preceding example, if you attempt to instantiate the abstract class by using a statement like this:

```
MyBaseC mC1 = new MyBaseC();   // Error
```

you will get the following error message:

```
Cannot create an instance of the abstract class 'MyBaseC'.
```

See Also

virtual | override | C# Keywords | Modifiers | 10.1.1.1 Abstract classes | 10.5.6 Abstract methods | 10.6.3 Virtual, sealed, override, and abstract accessors

const

The **const** keyword is used to modify a declaration of a field or local variable. It specifies that the value of the field or the local variable cannot be modified. A constant declaration introduces one or more constants of a given type. The declaration takes the form:

```
[attributes] [modifiers] const type declarators;
```

where:

attributes (optional)
Optional declarative information. For more information on attributes and attribute classes, see C# Attributes.
modifiers (optional)
Optional modifiers that include the **new** modifier and one of the four access modifiers.
type
One of the types: **byte**, **char**, **short**, **int**, **long**, **float**, **double**, **decimal**, **bool**, **string**, an enum type, or a reference type.
declarators
A comma-separated list of declarators. A declarator takes the form:

identifier = constant-expression

The *attributes* and *modifiers* apply to all of the members declared by the constant declaration.

The *type* of a constant declaration specifies the type of the members introduced by the declaration. A constant expression must yield a value of the target type, or of a type that can be implicitly converted to the target type.

A constant expression is an expression that can be fully evaluated at compile time. Therefore, the only possible values for constants of reference types are **string** and **null**.

Remarks

The constant declaration can declare multiple constants, for example:

```
public const double x = 1.0, y = 2.0, z = 3.0;
```

The **static** modifier is not allowed in a constant declaration.

A constant can participate in a constant expression, for example:

```
public const int c1 = 5.0;
public const int c2 = c1 + 100;
```

Example

```
// const_keyword.cs
// Constants
using System;
public class ConstTest
{
   class MyClass
   {
      public int x;
      public int y;
      public const int c1 = 5;
      public const int c2 = c1 + 5;

      public MyClass(int p1, int p2)
      {
         x = p1;
         y = p2;
      }
   }
```

```
    public static void Main()
    {
       MyClass mC = new MyClass(11, 22);
       Console.WriteLine("x = {0}, y = {1}", mC.x, mC.y);
       Console.WriteLine("c1 = {0}, c2 = {1}", MyClass.c1, MyClass.c2 );
    }
}
```

Output

```
x = 11, y = 22
c1 = 5, c2 = 10
```

Example

This example demonstrates using constants as local variables.

```
// const_keyword2.cs
using System;
public class TestClass
{
    public static void Main()
    {
       const int c = 707;
       Console.WriteLine("My local constant = {0}", c);
    }
}
```

Output

```
My local constant = 707
```

> **Note** The readonly keyword is different from the **const** keyword. A **const** field can only be initialized at the declaration of the field. A **readonly** field can be initialized either at the declaration or in a constructor. Therefore, **readonly** fields can have different values depending on the constructor used. Also, while a **const** field is a compile-time constant, the **readonly** field can be used for runtime constants, as in the following example:
>
> `public static readonly uint l1 = (uint) DateTime.Now.Ticks;`

See Also

C# Keywords | Modifiers

event

Specifies an event.

```
[attributes] [modifiers] event type declarator;
[attributes] [modifiers] event type member-name {accessor-declarations};
```

where:

attributes (optional)
Optional declarative information. For more information on attributes and attribute classes, see C# Attributes.
modifiers (optional)
Optional modifiers that include:

- abstract
- new
- override
- static
- virtual
- extern
- one of the four access modifiers

type
The delegate to which you want to associate this event.
declarator
The name of the event.
member-name
The name of the event.
accessor-declarations (optional)
Declaration of the accessors, which are used to add and remove event handlers in client code. The accessor functions are `add` and `remove`. It is an error to define one but not the other.

Remarks

The **event** keyword lets you specify a delegate that will be called upon the occurrence of some "event" in your code. The delegate can have one or more associated methods that will be called when your code indicates that the event has occurred. An event in one program can be made available to other programs that target the .NET Framework common language runtime.

The following steps must be taken in order to create and use C# events:

1. Create or identify a delegate. If you are defining your own event, you must also ensure that there is a delegate to use with the event keyword. If the event is predefined, in the .NET Framework for example, then consumers of the event need only know the name of the delegate.
2. Create a class that contains:
   a. An event created from the delegate.
   b. (optional) A method that verifies that an instance of the delegate declared with the **event** keyword exists. Otherwise, this logic must be placed in the code that fires the event.
   c. Methods that call the event. These methods can be overrides of some base class functionality.

   This class defines the event.

3. Define one or more classes that connect methods to the event. Each of these classes will include:

   - Associate one or more methods, using the += and -= operators, with the event in the base class.
   - The definition of the method(s) that will be associated with the event.

4. Use the event:

   - Create an object of the class that contains the event declaration.
   - Create an object of the class that contains the event definition, using the constructor that you defined.

For more information on events, see:

- 10.7 Events
- Events Tutorial
- Events and Delegates

Examples

A basic example of defining and using events can be found in the Events Tutorial. The following examples in this topic also demonstrate events.

Example	Comment
Example 1	Declaring an event in an interface and implementing it in a class.
Example 2	Using a hash table to store event instances.
Example 3	Implementing, via event properties, two interfaces that have an event with the same name .

Example 1

This example shows that it is possible to declare an event in an interface and implement it in a class:

```
// event_keyword.cs
using System;
public delegate void MyDelegate();   // delegate declaration

public interface I
{
   event MyDelegate MyEvent;
   void FireAway();
}

public class MyClass: I
{
   public event MyDelegate MyEvent;

   public void FireAway()
   {
      if (MyEvent != null)
         MyEvent();
   }
}

public class MainClass
{
   static private void f()
   {
      Console.WriteLine("This is called when the event fires.");
   }

   static public void Main ()
   {
      I i = new MyClass();

      i.MyEvent += new MyDelegate(f);
      i.FireAway();
   }
}
```

Example 2

One use for *accessor-declarations* is to expose a large number of events without allocating a field for each event, but instead using a hash table to store the event instances. This is only useful if you have a very large number of events, but you expect most of the events will not be implemented.

```
// event_keyword2.cs
using System;
using System.Collections;

public delegate void MyDelegate1(int i);
public delegate void MyDelegate2(string s);
public delegate void MyDelegate3(int i, object o);
public delegate void MyDelegate4();

public class PropertyEventsSample
{
   private Hashtable eventTable = new Hashtable();

   public event MyDelegate1 Event1
   {
      add
      {
         eventTable["Event1"] = (MyDelegate1)eventTable["Event1"] + value;
      }
      remove
      {
         eventTable["Event1"] = (MyDelegate1)eventTable["Event1"] - value;
      }
   }

   public event MyDelegate1 Event2
   {
      add
      {
         eventTable["Event2"] = (MyDelegate1)eventTable["Event2"] + value;
      }
      remove
      {
         eventTable["Event2"] = (MyDelegate1)eventTable["Event2"] - value;
      }
   }

   public event MyDelegate2 Event3
   {
      add
      {
         eventTable["Event3"] = (MyDelegate2)eventTable["Event3"] + value;
      }
      remove
      {
         eventTable["Event3"] = (MyDelegate2)eventTable["Event3"] - value;
      }
   }

   public event MyDelegate3 Event4
   {
      add
      {
         eventTable["Event4"] = (MyDelegate3)eventTable["Event4"] + value;
      }
      remove
      {
         eventTable["Event4"] = (MyDelegate3)eventTable["Event4"] - value;
      }
   }

   public event MyDelegate3 Event5
   {
      add
      {
         eventTable["Event5"] = (MyDelegate3)eventTable["Event5"] + value;
      }
      remove
```

```
        {
            eventTable["Event5"] = (MyDelegate3)eventTable["Event5"] - value;
        }
    }

    public event MyDelegate4 Event6
    {
        add
        {
            eventTable["Event6"] = (MyDelegate4)eventTable["Event6"] + value;
        }
        remove
        {
            eventTable["Event6"] = (MyDelegate4)eventTable["Event6"] - value;
        }
    }
}

public class MyClass
{
    public static void Main()
    {
    }
}
```

Example 3

Another use for event properties covers the situation where you are implementing two interfaces, each with an event of the same name. In such a case, you must use an explicit implementation event property:

```
// event_keyword3.cs
using System;

public delegate void MyDelegate1();

public interface I1
{
    event MyDelegate1 MyEvent;
}

public delegate int MyDelegate2(string s);

public interface I2
{
    event MyDelegate2 MyEvent;
}

public class ExplicitEventsSample: I1, I2
{
    public event MyDelegate1 MyEvent;  // normal implementation of I1.MyEvent.

    event MyDelegate2 I2.MyEvent   // explicit implementation of I2.MyEvent
    {
        add
        {
            MyEvent2Storage += value;
        }
        remove
        {
            MyEvent2Storage -= value;
        }
    }

    private MyDelegate2 MyEvent2Storage;  // underlying storage for I2.MyEvent.

    private void FireEvents()
    {
```

```
        if (MyEvent != null)
           MyEvent();
        if (MyEvent2Storage != null)
           MyEvent2Storage("hello");
     }
  }

  public class MyClass
  {
     public static void Main()
     {
     }
  }
```

See Also

C# Keywords | Modifiers | 1.7.5 Events

extern

Use the **extern** modifier in a method declaration to indicate that the method is implemented externally. A common use of the extern modifier is with the **DllImport** attribute. (See DllImportAttribute Class for more information.)

It is an error to use the abstract and **extern** modifiers together to modify the same member. Using the **extern** modifier means that the method is implemented outside the C# code, while using the **abstract** modifier means that the method implementation is not provided in the class.

Because an external method declaration provides no actual implementation, there is no method body; the method declaration simply ends with a semicolon and there are no braces ({ }) following the signature. For example:

```
public static extern int MyMethod(int x);
```

> **Note** The **extern** keyword is more limited in use than in C++. To compare with the C++ keyword, see Using extern to Specify Linkage in the *C++ Language Reference*.

For more information on external methods, see 10.5.7 External methods.

For more information on attributes, see 17. Attributes.

Example

In this example, the program receives a string from the user and displays it inside a message box. The program uses the `MessageBox` method imported from the `User32.dll` library.

```
using System;
using System.Runtime.InteropServices;
class MyClass
{
   [DllImport("User32.dll")]
   public static extern int MessageBox(int h, string m, string c, int type);

   public static int Main()
   {
      string myString;
      Console.Write("Enter your message: ");
      myString = Console.ReadLine();
      return MessageBox(0, myString, "My Message Box", 0);
   }
}
```

Sample Run

```
Enter your message: Where do you want to go today?
```

When the previous text is entered, a message box that contains the text will pop up on the screen.

Example

This example uses two files, `CM.cs` and `Cmdll.c`, to demonstrate **extern**. The C file is an external DLL that is invoked from within the C# program.

File: Cmdll.c

```
// cmdll.c
// compile with: /LD /MD
int __declspec(dllexport) MyMethod(int i)
{
   return i*10;
}
```

File: CM.cs

```
// cm.cs
using System;
using System.Runtime.InteropServices;
public class MyClass
{
   [DllImport("Cmdll.dll")]
   public static extern int MyMethod(int x);
   public static void Main()
   {
      Console.WriteLine("MyMethod() returns {0}.", MyMethod(5));
   }
}
```

Output

```
MyMethod() returns 50.
```

Compilation

To build the project, use the following steps:

- Compile `Cmdll.c` to a DLL using the Visual C++ command line:

  ```
  cl /LD /MD Cmdll.c
  ```

- Compile `CM.cs` using the command line:

  ```
  csc CM.cs
  ```

This will create the executable file `CM.exe`. When you run this program, `MyMethod` will pass the value 5 to the DLL file, which returns the value multiplied by 10.

See Also

C# Keywords | Modifiers

override

Use the **override** modifier to modify a method, a property, an indexer, or an event. An override method provides a new implementation of a member inherited from a base class. The method overridden by an override declaration is known as the overridden base method. The overridden base method must have the same signature as the override method.

You cannot override a non-virtual or static method. The overridden base method must be virtual, abstract, or override.

An override declaration cannot change the accessibility of the virtual method. Both the override method and the virtual method must have the same access level modifier.

You cannot use the following modifiers to modify an override method:

new static virtual abstract

An overriding property declaration must specify the exact same access modifier, type, and name as the inherited property, and the overridden property must be virtual, abstract, or override.

For more information on accessing the base class members, see 7.5.8 Base access.

For more information on overriding methods, see 10.5.4 Override methods.

Example

See the example for the virtual keyword.

From within the derived class that has an override method, you still can access the overridden base method that has the same name by using the **base** keyword. For example, if you have a virtual method `MyMethod()`, and an override method on a derived class, you can access the virtual method from the derived class by using the call:

```
base.MyMethod()
```

Compare this to the C++ way, where you use the scope resolution operator (`::`) and the base class name, for example:

```
My_Base_Class_Name::MyMethod()
```

Example

In this example, there is a base class, `Square`, and a derived class, `Cube`. Because the area of a cube is the sum of the areas of six squares, it is possible to calculate it by calling the `Area()` method on the base class.

```
// cs_override_keyword.cs
// Calling overriden methods from the base class
using System;
class TestClass
{
   public class Square
   {
      public double x;

      // Constructor:
      public Square(double x)
      {
         this.x = x;
      }

      public virtual double Area()
      {
         return x*x;
      }
   }

   class Cube: Square
   {
      // Constructor:
```

```
        public Cube(double x): base(x)
        {
        }

        // Calling the Area base method:
        public override double Area()
        {
           return (6*(base.Area()));
        }
     }

     public static void Main()
     {
        double x = 5.2;
        Square s = new Square(x);
        Square c = new Cube(x);
        Console.WriteLine("Area of Square = {0:F2}", s.Area());
        Console.WriteLine("Area of Cube = {0:F2}", c.Area());
     }
  }
```

Output

```
Area of Square = 27.04
Area of Cube = 162.24
```

See Also

abstract | virtual | C# Keywords | Modifiers | Versioning Tutorial | 10.5.4 Override methods

readonly

The **readonly** keyword is a modifier that you can use on fields. When a field declaration includes a **readonly** modifier, assignments to the fields introduced by the declaration can only occur as part of the declaration or in a constructor in the same class.

You can assign a value to a readonly field only in the following contexts:

- When the variable is initialized in the declaration, for example:

  ```
  public readonly int y = 5;
  ```

- For an instance field, in the instance constructors of the class that contains the field declaration, or for a static field, in the static constructor of the class that contains the field declaration. These are also the only contexts in which it is valid to pass a `readonly` field as an **out** or **ref** parameter.

Example

```
// cs_readonly_keyword.cs
// Readonly fields
using System;
public class ReadOnlyTest
{
   class MyClass
   {
      public int x;
      public readonly int y = 25; // Initialize a readonly field
      public readonly int z;

      public MyClass()
      {
         z = 24;   // Initialize a readonly instance field
      }

      public MyClass(int p1, int p2, int p3)
      {
         x = p1;
         y = p2;
         z = p3;
      }
   }

   public static void Main()
   {
      MyClass p1= new MyClass(11, 21, 32);   // OK
      Console.WriteLine("p1: x={0}, y={1}, z={2}" , p1.x, p1.y, p1.z);
      MyClass p2 = new MyClass();
      p2.x = 55;   // OK
      Console.WriteLine("p2: x={0}, y={1}, z={2}" , p2.x, p2.y, p2.z);
   }
}
```

Output

```
p1: x=11, y=21, z=32
p2: x=55, y=25, z=24
```

In the preceding example, if you use a statement like this:

```
p2.y = 66;        // Error
```

you will get the compiler error message:

```
The left-hand side of an assignment must be an l-value
```

which is the same error you get when you attempt to assign a value to a constant.

> **Note** The **readonly** keyword is different from the const keyword. A **const** field can only be initialized at the declaration of the field. A **readonly** field can be initialized either at the declaration or in a constructor. Therefore, **readonly** fields can have different values depending on the constructor used. Also, while a **const** field is a compile-time constant, the **readonly** field can be used for runtime constants as in the following example:
>
> ```
> public static readonly uint l1 = (uint) DateTime.Now.Ticks;
> ```

See Also

C# Keywords | Modifiers

sealed

A sealed class cannot be inherited. It is an error to use a sealed class as a base class. Use the **sealed** modifier in a class declaration to prevent inheritance of the class.

It is not permitted to use the abstract modifier with a sealed class.

Structs are implicitly sealed; therefore, they cannot be inherited.

Example

```
// cs_sealed_keyword.cs
// Sealed classes
using System;
sealed class MyClass
{
   public int x;
   public int y;
}

class MainClass
{
   public static void Main()
   {
      MyClass mC = new MyClass();
      mC.x = 110;
      mC.y = 150;
      Console.WriteLine("x = {0}, y = {1}", mC.x, mC.y);
   }
}
```

Output

```
x = 110, y = 150
```

In the preceding example, if you attempt to inherit from the sealed class by using a statement like this:

```
class MyDerivedC: MyClass {}   // Error
```

you will get the error message:

```
'MyDerivedC' cannot inherit from sealed class 'MyBaseC'.
```

See Also

C# Keywords | Modifiers

static

Use the **static** modifier to declare a static member, which belongs to the type itself rather than to a specific object. The **static** modifier can be used with fields, methods, properties, operators, events and constructors, but cannot be used with indexers, destructors, or types.

Remarks

- A constant or type declaration is implicitly a static member.
- A static member cannot be referenced through an instance. Instead, it is referenced through the type name. For example, consider the following class:

  ```
  public class MyBaseC
  {
     public struct MyStruct {
        public static int x = 100;
     }
  }
  ```

  To refer to the static member `x`, use the fully qualified name (unless it is accessible from the same scope):

  ```
  MyBaseC.MyStruct.x
  ```

- While an instance of a class contains a separate copy of all instance fields of the class, there is only one copy of each static field.
- It is not possible to use **this** to reference static methods or property accessors.

> **Note** The **static** keyword is more limited in use than in C++. To compare with the C++ keyword, see static in the *C++ Language Reference*.

To demonstrate instance members, consider a class that represents a company employee. Assume that the class contains a method to count employees and a field to store the number of employees. Both the method and the field do not belong to any instance employee. Instead they belong to the company class. Therefore, they should be declared as static members of the class.

For more information on constructors, see 10.10 Instance constructors.

Example

This example reads the name and ID of a new employee, increments the employee counter by one, and displays the information for the new employee as well as the new number of employees. For simplicity, this program reads the current number of employees from the keyboard. In a real application, this information should be read from a file.

```
// cs_static_keyword.cs
// Static members
using System;
public class Employee
{
   public string id;
   public string name;

   public Employee ()
   {
   }

   public Employee (string name, string id)
   {
      this.name = name;
      this.id = id;
   }

   public static int employeeCounter;
```

```
   public static int AddEmployee()
   {
      return ++employeeCounter;
   }
}

class MainClass: Employee
{
   public static void Main()
   {
      Console.Write("Enter the employee's name: ");
      string name = Console.ReadLine();
      Console.Write("Enter the employee's ID: ");
      string id = Console.ReadLine();
      // Create the employee object:
      Employee e = new Employee (name, id);
      Console.Write("Enter the current number of employees: ");
      string n = Console.ReadLine();
      Employee.employeeCounter = Int32.Parse(n);
      Employee.AddEmployee();
      // Display the new information:
      Console.WriteLine("Name: {0}", e.name);
      Console.WriteLine("ID:   {0}", e.id);
      Console.WriteLine("New Number of Employees: {0}",
                        Employee.employeeCounter);
   }
}
```

Input

```
Tara Strahan
AF643G
15
```

Sample Output

```
Enter the employee's name: Tara Strahan
Enter the employee's ID: AF643G
Enter the current number of employees: 15
Name: Tara Strahan
ID:   AF643G
New Number of Employees: 16
```

See Also

C# Keywords | Modifiers

unsafe

The **unsafe** keyword denotes an unsafe context, which is required for any operation involving pointers.

You can use the **unsafe** modifier in the declaration of a type or a member. The entire textual extent of the type or member is therefore considered an unsafe context. For example, the following is a method declared with the **unsafe** modifier:

```
unsafe static void FastCopy ( byte[] src, byte[] dst, int count )
{
   // unsafe context: can use pointers here
}
```

The scope of the unsafe context extends from the parameter list to the end of the method, so pointers can also be used in the parameter list:

```
unsafe static void FastCopy ( byte* ps, byte* pd, int count ) {...}
```

You can also use an unsafe block to enable the use of an unsafe code inside this block. For example:

```
unsafe
{
   // unsafe context: can use pointers here
}
```

To compile unsafe code, you must specify the /unsafe compiler option. Unsafe code is not verifiable by the common language runtime.

Example

```
// cs_unsafe_keyword.cs
// compile with: /unsafe
using System;
class UnsafeTest
{
   // unsafe method: takes pointer to int:
   unsafe static void SquarePtrParam (int* p)
   {
      *p *= *p;
   }
   unsafe public static void Main()
   {
      int i = 5;
      // unsafe method: uses address-of operator (&)
      SquarePtrParam (&i);
      Console.WriteLine (i);
   }
}
```

Output

```
25
```

See Also

C# Keywords | fixed Statement | Unsafe Code Tutorial | A. Unsafe code

virtual

The **virtual** keyword is used to modify a method or property declaration, in which case the method or the property is called a virtual member. The implementation of a virtual member can be changed by an overriding member in a derived class.

When a virtual method is invoked, the run-time type of the object is checked for an overriding member. The overriding member in the most derived class is called, which might be the original member, if no derived class has overridden the member. (For more information on run-time type and most derived implementation, see 10.5.3 Virtual methods.)

By default, methods are non-virtual. You cannot override a non-virtual method.

You cannot use the **virtual** modifier with the following modifiers:

static abstract override

Virtual properties behave like abstract methods, except for the differences in declaration and invocation syntax.

- It is an error to use the **virtual** modifier on a static property.
- A virtual inherited property can be overridden in a derived class by including a property declaration that uses the **override** modifier.

For more information on virtual methods, see 10.5.3 Virtual methods.

Example

In this example, the class `Dimensions` contains the two coordinates `x`, `y`, and the `Area()` virtual method. Different shape classes such as `Circle`, `Cylinder`, and `Sphere` inherit the `Dimensions` class, and the surface area is calculated for each figure. Each derived class has it own override implementation of `Area()`. The program calculates and displays the proper area for each figure by invoking the proper implementation of `Area()` according to the object associated with the method.

```
// cs_virtual_keyword.cs
// Virtual and override
using System;
class TestClass
{
   public class Dimensions
   {
      public const double pi = Math.PI;
      protected double x, y;
      public Dimensions()
      {
      }
      public Dimensions (double x, double y)
      {
         this.x = x;
         this.y = y;
      }

      public virtual double Area()
      {
         return x*y;
      }
   }

   public class Circle: Dimensions
   {
      public Circle(double r): base(r, 0)
      {
      }

      public override double Area()
      {
         return pi * x * x;
      }
   }
```

```
    class Sphere: Dimensions
    {
       public Sphere(double r): base(r, 0)
       {
       }

       public override double Area()
       {
          return 4 * pi * x * x;
       }
    }

    class Cylinder: Dimensions
    {
       public Cylinder(double r, double h): base(r, h)
       {
       }

       public override double Area()
       {
          return 2*pi*x*x + 2*pi*x*y;
       }
    }

    public static void Main()
    {
       double r = 3.0, h = 5.0;
       Dimensions c = new Circle(r);
       Dimensions s = new Sphere(r);
       Dimensions l = new Cylinder(r, h);
       // Display results:
       Console.WriteLine("Area of Circle   = {0:F2}", c.Area());
       Console.WriteLine("Area of Sphere   = {0:F2}", s.Area());
       Console.WriteLine("Area of Cylinder = {0:F2}", l.Area());
    }
 }
```

Output

```
Area of Circle   = 28.27
Area of Sphere   = 113.10
Area of Cylinder = 150.80
```

In the preceding example, notice that the inherited classes `Circle`, `Sphere`, and `Cylinder` are all using constructors that initialize the base class, for example:

```
public Cylinder(double r, double h): base(r, h) {}
```

This is analogous to the C++ initialization list.

See Also

abstract | override | C# Keywords | Modifiers | Versioning Tutorial | 10.5.3 Virtual methods

volatile

The **volatile** keyword indicates that a field can be modified in the program by something such as the operating system, the hardware, or a concurrently executing thread.

```
volatile declaration
```

where:

declaration
The declaration of a field.

Remarks

The system always reads the current value of a volatile object at the point it is requested, even if the previous instruction asked for a value from the same object. Also, the value of the object is written immediately on assignment.

The **volatile** modifier is usually used for a field that is accessed by multiple threads without using the lock statement to serialize access. Using the **volatile** modifier ensures that one thread retrieves the most up-to-date value written by another thread.

The type of a field marked as volatile is restricted to the following types:

- Any reference type.
- Any pointer type (in an unsafe context).
- The types sbyte, byte, short, ushort, int, uint, char, float, bool.
- An enum type with an enum base type of byte, sbyte, short, ushort, int, or uint.

For more information on volatile, see 10.4.3 Volatile fields.

Example

The following sample shows how to declare a public field variable as volatile.

```
// csharp_volatile.cs
class Test
{
   public volatile int i;

   Test(int _i)
   {
      i = _i;
   }
   public static void Main()
   {

   }
}
```

See Also

C# Keywords | Modifiers | 10.4.3 Volatile fields

Statements

Statements are program instructions. Except as described, statements are executed in sequence. C# has the following categories of statements.

Category	C# keywords
Selection statements	if, else, switch, case
Iteration statements	do, for, foreach, in, while
Jump statements	break, continue, default, goto, return
Exception handling statements	throw, try-catch, try-finally, try-catch-finally
Checked and unchecked	checked, unchecked
fixed Statement	fixed
lock Statement	lock

See Also

C# Keywords

Selection Statements

A selection statement causes the program control to be transferred to a specific flow based upon whether a certain condition is true or not.

The following keywords are used in selection statements:

- if
- else
- switch
- case

See Also

C# Keywords | Statements | 8.7 Selection statements

if-else

The **if** statement selects a statement for execution based on the value of a Boolean expression. It takes the following form:

```
if (expression)
   statement1
[else
   statement2]
```

where:

expression
An expression that can be implicitly converted to **bool** or a type that contains overloading of the true and false operators.
statement1
The embedded statement(s) to be executed if *expression* is **true**.
statement2
The embedded statement(s) to be executed if *expression* is **false**.

Remarks

If *expression* is **true**, *statement1* is executed. If the optional **else** clause exists and *expression* evaluates to **false**, *statement2* is executed. After executing the **if** statement, control is transferred to the next statement.

If any of the two results of the **if** statement (**true** or **false**) results in executing more than one statement, multiple statements can be conditionally executed by including them into blocks.

The statement(s) to be executed upon testing the condition can be of any kind, including another **if** statement nested into the original **if** statement. In nested **if** statements, the **else** clause belongs to the last **if** that does not have a corresponding **else**. For example:

```
if (x > 10)
   if (y > 20)
      Console.Write("Statement_1");
   else
      Console.Write("Statement_2");
```

In this example, `Statement_2` will be displayed if the condition `(y > 20)` evaluates to **false**. However, if you want to associate `Statement_2` with the condition `(x >10)`, use braces:

```
if (x > 10)
{
   if (y > 20)
      Console.Write("Statement_1");
}
else
   Console.Write("Statement_2");
```

In this case, `Statement_2` will be displayed if the condition `(x > 10)` evaluates to **false**

Example

In this example, you enter a character from the keyboard and the program checks if the input character is an alphabetic character. If so, it checks if it is lowercase or uppercase. In each case, the proper message is displayed.

```
// statements_if_else.cs
// if-else example
using System;
public class IfTest
{
   public static void Main()
   {
      Console.Write("Enter a character: ");
```

```
      char c = (char) Console.Read();
      if (Char.IsLetter(c))
         if (Char.IsLower(c))
            Console.WriteLine("The character is lowercase.");
         else
            Console.WriteLine("The character is uppercase.");
      else
         Console.WriteLine("The character is not an alphabetic character.");
   }
}
```

Input

```
2
```

Sample Output

```
Enter a character: 2
The character is not an alphabetic character.
```

Additional sample might look as follows:

Run #2:

```
Enter a character: A
The character is uppercase.
```

Run #3:

```
Enter a character: h
The character is lowercase.
```

It is also possible to extend the **if** statement to handle multiple conditions using the following else-if arrangement:

```
if (Condition_1)
   Statement_1;
else if (Condition_2)
   Statement_2;
else if (Condition_3)
   Statement_3;
...
else
   Statement_n;
```

Example

This example checks if the input character is lowercase, uppercase, or a number. Otherwise, it is not an alphanumeric character. The program makes use of the else-if ladder.

```
// statements_if_else2.cs
// else-if
using System;
public class IfTest
{
   public static void Main()
   {
      Console.Write("Enter a character: ");
      char c = (char) Console.Read();

      if (Char.IsUpper(c))
```

```
            Console.WriteLine("The character is uppercase.");
        else if (Char.IsLower(c))
            Console.WriteLine("The character is lowercase.");
        else if (Char.IsDigit(c))
            Console.WriteLine("The character is a number.");
        else
            Console.WriteLine("The character is not alphanumeric.");
    }
}
```

Input

```
E
```

Sample Output

```
Enter a character: E
The character is uppercase.
```

Additional sample runs might look as follows:

Run #2

```
Enter a character: e
The character is lowercase.
```

Run #3:

```
Enter a character: 4
The character is a number.
```

Run #4:

```
Enter a character: $
The character is not alphanumeric.
```

See Also

C# Keywords | ?: Operator | Compare to C++ | switch | C. Grammar

switch

The **switch** statement is a control statement that handles multiple selections by passing control to one of the **case** statements within its body. It takes the following form:

```
switch (expression)
{
   case constant-expression:
      statement
      jump-statement
   [default:
      statement
      jump-statement]
}
```

Where:

expression
An integral or string type expression.
statement
The embedded statement(s) to be executed if control is transferred to the **case** or the **default**.
jump-statement
A jump statement that transfers control out of the **case** body.
constant-expression
Control is transferred to a specific **case** according to the value of this expression.

Remarks

Control is transferred to the **case** statement whose *constant-expression* matches *expression*. The **switch** statement can include any number of **case** instances, but no two case constants within the same **switch** statement can have the same value. Execution of the statement body begins at the selected statement and proceeds until the *jump-statement* transfers control out of the **case** body.

Notice that the *jump-statement* is required after each block, including the last block whether it is a **case** statement or a **default** statement. Unlike the C++ **switch** statement, C# does not support an explicit fall through from one case label to another. If you want, you can use **goto** a switch-case, or **goto default**.

If *expression* does not match any *constant-expression*, control is transferred to the *statement*(s) that follow the optional **default** label. If there is no **default** label, control is transferred outside the **switch**.

Example

```
// statements_switch.cs
using System;
class SwitchTest
{
   public static void Main()
   {
      Console.WriteLine("Coffee sizes: 1=Small 2=Medium 3=Large");
      Console.Write("Please enter your selection: ");
      string s = Console.ReadLine();
      int n = int.Parse(s);
      int cost = 0;
      switch(n)
      {
         case 1:
            cost += 25;
            break;
         case 2:
            cost += 25;
            goto case 1;
         case 3:
            cost += 50;
            goto case 1;
```

```
            default:
               Console.WriteLine("Invalid selection. Please select 1, 2, or 3.");
               break;
         }
         if (cost != 0)
            Console.WriteLine("Please insert {0} cents.", cost);
         Console.WriteLine("Thank you for your business.");
      }
   }
```

Input

```
2
```

Sample Output

```
Coffee sizes: 1=Small 2=Medium 3=Large
Please enter your selection: 2
Please insert 50 cents.
Thank you for your business.
```

Code Discussion

- In the preceding example, an integral type variable, n, was used for the switch cases. Notice that you can also use the string variable, `s`, directly. In this case, you can use switch cases like these:

  ```
  switch(s)
  {
     case "1":
        ...
     case "2":
        ...
  }
  ```

- Although fall through from one case label to another is not supported, it is allowed to stack case labels, for example:

  ```
     case 0:
     case 1:
        // do something;
  ```

See Also

C# Keywords | Compare to C++ | if | C. Grammar

Iteration Statements

You can create loops by using the iteration statements. Iteration statements cause embedded statements to be executed a number of times, subject to the loop-termination criteria. These statements are executed in order, except when a jump statement is encountered.

The following keywords are used in iteration statements:

- do
- for
- foreach
- in
- while

See Also

C# Keywords | Statements | 8.8 Iteration statements

do

The **do** statement executes a statement or a block of statements repeatedly until a specified expression evaluates to **false**. It takes the following form:

```
do statement while (expression);
```

where:

expression
An expression that can be implicitly converted to **bool** or a type that contains overloading of the true and false operators. The expression is used to test the loop-termination criteria.

statement
The embedded statement(s) to be executed.

Remarks

Unlike the while statement, the body loop of the **do** statement is executed at least once regardless of the value of the *expression*.

Example

```
// statements_do.cs
using System;
public class TestDoWhile
{
   public static void Main ()
   {
      int x;
      int y = 0;

      do
      {
         x = y++;
         Console.WriteLine(x);
      }

      while(y < 5);
   }
}
```

Output

```
0
1
2
3
4
```

Example

Notice in this example that although the condition evaluates to **false**, the loop will be executed once.

```
// statements_do2.cs
using System;
class DoTest {
   public static void Main()
   {
      int n = 10;
      do
      {
```

```
            Console.WriteLine("Current value of n is {0}", n);
            n++;
        } while (n < 6);
    }
}
```

Output

```
Current value of n is 10
```

See Also

C# Keywords | Compare to C++ | Iteration Statements | C. Grammar

for

The **for** loop executes a statement or a block of statements repeatedly until a specified expression evaluates to **false**. It takes the following form:

```
for ([initializers]; [expression]; [iterators]) statement
```

where:

initializers
A comma separated list of expressions or assignment statements to initialize the loop counters.
expression
An expression that can be implicitly converted to **bool** or a type that contains overloading of the true and false operators. The expression is used to test the loop-termination criteria.
iterators
Expression statement(s) to increment or decrement the loop counters.
statement
The embedded statement(s) to executed.

Remarks

The **for** statement executes the *statement* repeatedly as follows:

- First, the *initializers* are evaluated.
- Then, while the *expression* evaluates to **true**, the *statement*(s) are executed and the *iterators* are evaluated.
- When the *expression* becomes **false**, control is transferred outside the loop.

Because the test of *expression* takes place before the execution of the loop, a **for** statement executes zero or more times.

All of the expressions of the **for** statement are optional; for example, the following statement is used to write an infinite loop:

```
for (;;) {
   ...
}
```

Example

```
// statements_for.cs
// for loop
using System;
public class ForLoopTest
{
   public static void Main()
   {
      for (int i = 1; i <= 5; i++)
         Console.WriteLine(i);
   }
}
```

Output

```
1
2
3
4
5
```

See Also

foreach, in

The **foreach** statement repeats a group of embedded statements for each element in an array or an object collection. The **foreach** statement is used to iterate through the collection to get the desired information, but should not be used to change the contents of the collection to avoid unpredictable side effects. The statement takes the following form:

```
foreach (type identifier in expression) statement
```

where:

type
The type of *identifier*.

identifier
The iteration variable that represents the collection element. If the iteration variable is a value type, it is effectively a read-only variable that cannot be modified.

expression
Object collection or array expression. The type of the collection element must be convertible to the *identifier* type. Do not use an expression that evaluates to **null**.

Evaluates to a type that implements **IEnumerable** or a type that declares a **GetEnumerator** method. In the latter case, **GetEnumerator** should either return a type that implements **IEnumerator** or declares all the methods defined in **IEnumerator**.

statement
The embedded statement(s) to be executed.

Remarks

The embedded statements continue to execute for each element in the array or collection. After the iteration has been completed for all the elements in the collection, control is transferred to the next statement following the **foreach** block.

For more information on the **foreach** keyword, including code samples, see the following topics:

- Using foreach with Arrays
- Using foreach with Collections

See Also

C# Keywords | Iteration Statements | C. Grammar

Using foreach with Arrays

When used with an array, the foreach statement repeats the embedded statement(s) for each element in the array.

Example

In this example, an array of integers is searched for even and odd numbers. A counter for each type of number stores the number of occurrences.

```
// statements_foreach_arrays.cs
// Using foreach with arrays
using System;
class MainClass
{
   public static void Main()
   {
      int odd = 0, even = 0;
      int[] arr = new int [] {0,1,2,5,7,8,11};

      foreach (int i in arr)
      {
         if (i%2 == 0)
            even++;
         else
            odd++;
      }

      Console.WriteLine("Found {0} Odd Numbers, and {1} Even Numbers.",
                        odd, even) ;
   }
}
```

Output

```
Found 4 Odd Numbers, and 3 Even Numbers.
```

See Also

foreach, in | Arrays | Using foreach with Collections

Using foreach with Collections

To iterate through a collection, the collection must meet specific requirements. For example, in the following foreach statement:

```
foreach (ItemType item in myCollection)
```

`myCollection` must meet the following requirements:

- The collection type:
  - Must be one of the types: **interface**, **class**, or **struct**.
  - Must include an instance method named **GetEnumerator** that returns a type, for example, `Enumerator` (explained below).
- The type `Enumerator` (a class or struct) must contain:
  - A property named **Current** that returns `ItemType` or a type that can be converted to it. The property accessor returns the current element of the collection.
  - A **bool** method, named **MoveNext**, that increments the item counter and returns **true** if there are more items in the collection.

There are three approaches to using collections:

1. Create a collection using the above instructions. This collection can only be used in a C# program.
2. Create a generic collection using the above instructions, in addition to implementing the **IEnumerable** interface. This collection can be used from other languages such as Visual Basic.
3. Use one of the predefined collections in the collection classes.

The following examples demonstrate the three approaches.

Example	Demonstrates	Comment
Example 1	A C#-specific collection.	This example creates a collection by using the instructions above.
Example 2a	A generic collection.	This example creates a collection by using the instructions above and implements the **IEnumerable** and **IEnumerat or** interfaces.
Example 2b	A generic collection with a Dispose method.	Same as Example 2a except that a user-defined enumerat or inherits from **IDisposable** and implements a **Dispose** method.
Example 3	Using one of the predefined collection class es.	This example creates a **Hashtable** instance and uses the **Hashtable** class members to manipulate the collection. T he **Hashtable** class represents a dictionary of associated keys and values, implemented as a hash table.

For more information on the interfaces **IEnumerator** and **IEnumerable**, and the **Hashtable** class, see the System.Collections section in the .NET Framework SDK.

Example 1

```
// statements_foreach_collections.cs
// Using foreach with C#-specific collections:
using System;

// Declare the collection:
public class MyCollection
{
   int[] items;

   public MyCollection()
   {
      items = new int[5] {12, 44, 33, 2, 50};
   }
```

```
    public MyEnumerator GetEnumerator()
    {
       return new MyEnumerator(this);
    }

    // Declare the enumerator class:
    public class MyEnumerator
    {
       int nIndex;
       MyCollection collection;
       public MyEnumerator(MyCollection coll)
       {
          collection = coll;
          nIndex = -1;
       }

       public bool MoveNext()
       {
          nIndex++;
          return(nIndex < collection.items.GetLength(0));
       }

       public int Current
       {
          get
          {
             return(collection.items[nIndex]);
          }
       }
    }
 }

 public class MainClass
 {
    public static void Main()
    {
       MyCollection col = new MyCollection();
       Console.WriteLine("Values in the collection are:");

       // Display collection items:
       foreach (int i in col)
       {
          Console.WriteLine(i);
       }
    }
 }
```

Output

```
Values in the collection are:
12
44
33
2
50
```

Example 2a

This example repeats the same algorithm of Example 1, but uses a generic collection that can be enumerated from other languages such as Visual Basic. This type of collection must implement the **IEnumerable** interface from the **System.Collections** namespace.

```
// statements_foreach_generic_collection.cs
// Using foreach with a generic collection
```

```
using System;
using System.Collections;

// Declare the collection and implement the IEnumerable interface:
public class MyCollection: IEnumerable
{
   int[] items;
   public MyCollection()
   {
      items = new int[5] {12, 44, 33, 2, 50};
   }

   public MyEnumerator GetEnumerator()
   {
      return new MyEnumerator(this);
   }

   // Implement the GetEnumerator() method:
   IEnumerator IEnumerable.GetEnumerator()
   {
      return GetEnumerator();
   }

   // Declare the enumerator and implement the IEnumerator interface:
   public class MyEnumerator: IEnumerator
   {
      int nIndex;
      MyCollection collection;
      public MyEnumerator(MyCollection coll)
      {
         collection = coll;
         nIndex = -1;
      }

      public void Reset()
      {
         nIndex = -1;
      }

      public bool MoveNext() {
         nIndex++;
         return(nIndex < collection.items.GetLength(0));
      }

      public int Current
      {
         get
         {
            return(collection.items[nIndex]);
         }
      }

      // The current property on the IEnumerator interface:
      object IEnumerator.Current
      {
         get
         {
            return(Current);
         }
      }
   }
}

public class MainClass
{
   public static void Main(string [] args)
   {
      MyCollection col = new MyCollection();
```

```
        Console.WriteLine("Values in the collection are:");

        // Display collection items:
        foreach (int i in col)
        {
           Console.WriteLine(i);
        }
     }
  }
```

Output

```
Values in the collection are:
12
44
33
2
50
```

Example 2b

This example is the same as Example 2a except that it also implements a **Dispose** method. **Dispose** is implemented in a user-defined enumerator that inherits from **IDisposable**.

The **foreach** statement provides support for disposable enumerators. If an enumerator implements the **IDisposable** interface, the **foreach** statement guarantees that the **Dispose** method is called on the enumerator no matter how the enumeration loop is terminated.

```
// statements_foreach_generic_collection2.cs
// Using foreach with a generic collection
using System;
using System.Collections;

// Declare the collection and implement the IEnumerable interface:
public class MyCollection: IEnumerable
{
   int[] items;
   public MyCollection()
   {
      items = new int[5] {12, 44, 33, 2, 50};
   }

   public MyEnumerator GetEnumerator()
   {
      return new MyEnumerator(this);
   }

   // Implement the GetEnumerator() method:
   IEnumerator IEnumerable.GetEnumerator()
   {
      return GetEnumerator();
   }

   // Declare the enumerator and implement the IEnumerator
   // and IDisposable interfaces
   public class MyEnumerator: IEnumerator, IDisposable
   {
      int nIndex;
      MyCollection collection;
      public MyEnumerator(MyCollection coll)
      {
         collection = coll;
         nIndex = -1;
      }
```

```
      public void Reset()
      {
         nIndex = -1;
      }

      public bool MoveNext()
      {
         nIndex++;
         return (nIndex < collection.items.GetLength(0));
      }

      public int Current
      {
         get
         {
            return (collection.items[nIndex]);
         }
      }

      // The current property on the IEnumerator interface:
      object IEnumerator.Current
      {
         get
         {
            return (Current);
         }
      }

      public void Dispose()
      {
         Console.WriteLine("In Dispose");
         collection = null;
      }
   }
}

public class MainClass
{
   public static void Main(string [] args)
   {
      MyCollection col = new MyCollection();
      Console.WriteLine("Values in the collection are:");

      // Display collection items:
      foreach (int i in col)
      {
         Console.WriteLine(i);
      }
   }
}
```

Output

```
Values in the collection are:
12
44
33
2
50
In Dispose
```

Example 3

In this example, the predefined **Hashtable** collection class is used. By using the **System.Collections** namespace in your program, you can have access to the **Hashtable** class and its members. To add entries to the **Hashtable** object, use the **Add** method.

```
// statements_foreach_hashtable.cs
// Using the Hashtable collection class
using System;
using System.Collections;
public class MainClass
{
   public static void Main(string [] args)
   {
      // Declare a Hashtable object:
      Hashtable ziphash = new Hashtable();

      // Add entries using the Add() method:
      ziphash.Add("98008", "Bellevue");
      ziphash.Add("98052", "Redmond");
      ziphash.Add("98201", "Everett");
      ziphash.Add("98101", "Seattle");
      ziphash.Add("98371", "Puyallup");

      // Display contents:
      Console.WriteLine("Zip code       City");
      foreach (string zip in ziphash.Keys)
      {
         Console.WriteLine(zip + "          " + ziphash[zip]);
      }
   }
}
```

Output

```
Zip code       City
98201          Everett
98052          Redmond
98101          Seattle
98008          Bellevue
98371          Puyallup
```

See Also

foreach, in | Collection Classes Tutorial | Using foreach with Arrays

while

The **while** statement executes a statement or a block of statements until a specified expression evaluates to **false**. It takes the following form:

```
while (expression) statement
```

where:

expression
An expression that can be implicitly converted to **bool** or a type that contains overloading of the true and false operators. The expression is used to test the loop-termination criteria.

statement
The embedded statement(s) to be executed.

Remarks

Because the test of *expression* takes place before each execution of the loop, a **while** loop executes zero or more times.

A **while** loop can be terminated when a **break**, **goto**, **return**, or **throw** statement transfers control outside the loop. To pass control to the next iteration without exiting the loop, use the **continue** statement.

Example

```
// statements_while.cs
using System;
class WhileTest
{
   public static void Main()
   {
      int n = 1;

      while (n < 6)
      {
         Console.WriteLine("Current value of n is {0}", n);
         n++;
      }
   }
}
```

Output

```
Current value of n is 1
Current value of n is 2
Current value of n is 3
Current value of n is 4
Current value of n is 5
```

See Also

C# Keywords | Compare to C++ | Iteration Statements | C. Grammar

Jump Statements

Branching is performed using jump statements, which cause an immediate transfer of the program control. The following keywords are used in jump statements:

- break
- continue
- default
- goto
- return

See Also

C# Keywords | Statements | 8.9 Jump statements

break

The **break** statement terminates the closest enclosing loop or **switch** statement in which it appears. Control is passed to the statement that follows the terminated statement, if any. This statement takes the following form:

```
break;
```

Example

In this example, the conditional statement contains a counter that is supposed to count from 1 to 100; however, the **break** statement terminates the loop after 4 counts.

```
// statements_break.cs
using System;
class BreakTest
{
   public static void Main()
   {
      for (int i = 1; i <= 100; i++)
      {
         if (i == 5)
            break;
         Console.WriteLine(i);
      }
   }
}
```

Output

```
1
2
3
4
```

Example

This example demonstrates the use of **break** in a **switch** statement.

```
// statements_break2.cs
// break and switch
using System;
class Switch
{
   public static void Main()
   {
      Console.Write("Enter your selection (1, 2, or 3): ");
      string s = Console.ReadLine();
      int n = Int32.Parse(s);

      switch(n)
      {
         case 1:
            Console.WriteLine("Current value is {0}", 1);
            break;
         case 2:
            Console.WriteLine("Current value is {0}", 2);
            break;
         case 3:
            Console.WriteLine("Current value is {0}", 3);
            break;
         default:
```

```
            Console.WriteLine("Sorry, invalid selection.");
            break;
        }
    }
}
```

Input

```
1
```

Sample Output

```
Enter your selection (1, 2, or 3): 1
Current value is 1
```

If you entered `4`, the output would be:

```
Enter your selection (1, 2, or 3): 4
Sorry, invalid selection.
```

See Also

C# Keywords | Compare to C++ | switch | Jump Statements | C. Grammar

continue

The **continue** statement passes control to the next iteration of the enclosing iteration statement in which it appears. It takes the following form:

```
continue;
```

Example

In this example, a counter is initialized to count from 1 to 10. By using the **continue** statement in conjunction with the expression `(i < 9)`, the statements between **continue** and the end of the **for** body are skipped.

```
// statements_continue.cs
using System;
class ContinueTest
{
   public static void Main()
   {
      for (int i = 1; i <= 10; i++)
      {
         if (i < 9)
            continue;
         Console.WriteLine(i);
      }
   }
}
```

Output

```
9
10
```

See Also

C# Keywords | Compare to C++ | Jump Statements | C. Grammar

goto

The **goto** statement transfers the program control directly to a labeled statement. It takes one of the following forms:

```
goto identifier;
goto case constant-expression;
goto default;
```

where:

identifier
A label.

constant-expression
A switch-case label.

Remarks

In the first form, the *identifier* indicates a label located in the current body, the same lexical scope, or an enclosing scope of the **goto** statement.

A common use of **goto** is to transfer control to a specific switch-case label or the default label in a **switch** statement.

The **goto** statement is also useful to get out of deeply nested loops.

A warning message may be issued if the label has never been referenced in the program. For more information on labels, see 3.3 Declarations.

Example

For an example of using **goto** to transfer control to a specific switch-case label, see the switch example.

Example

The following example demonstrates using **goto** to break out from nested loops.

```
// statements_goto.cs
// Nested search loops
using System;
public class GotoTest1
{
   public static void Main()
   {
      int x = 200, y = 4;
      int count = 0;
      string[,] myArray = new string[x,y];

      // Initialize the array:
      for (int i = 0; i < x; i++)
         for (int j = 0; j < y; j++)
            myArray[i,j] = (++count).ToString();

      // Read input:
      Console.Write("Enter the number to search for: ");

      // Input a string:
      string myNumber = Console.ReadLine();

      // Search:
      for (int i = 0; i < x; i++)
         for (int j = 0; j < y; j++)
            if (myArray[i,j].Equals(myNumber))
               goto Found;

      Console.WriteLine("The number {0} was not found.", myNumber);
      goto Finish;
```

```
   Found:
      Console.WriteLine("The number {0} is found.", myNumber);

   Finish:
      Console.WriteLine("End of search.");
   }
}
```

Input

```
44
```

Sample Output

```
Enter the number to search for: 44
The number 44 is found.
End of search.
```

Example

```
// statements_goto2.cs
// CS0159 expected
// Labels outside the scope
using System;
class UnreachableCode
{
   public static void Main()
   {
      int x = 55;
      Console.WriteLine("x = {0}", x);

      if (x == 55)
      {
         x = 135;
         goto A;   // Error
      }

      x = x + 1;

      for (int i=1; i<=5; i++)
      {
         A: Console.WriteLine(i);
      }

      Console.WriteLine("x = {0}", x);
   }
}
```

In the preceding example, the **goto** statement is referencing a label `A` outside its scope. The compiler will issue the error message:

```
No such label 'A' within the scope of the goto statement
```

A warning message may also be issued because the label has never been referenced.

If you move the label `A` to the beginning of the **for** loop, the program would compile and run normally, that is:

```
A:      for (int i=1; i<=5; i++) {   // Now the program compiles.
```

See Also

C# Keywords | Compare to C++ | Jump Statements | C. Grammar

return

The **return** statement terminates execution of the method in which it appears and returns control to the calling method. It can also return the value of the optional *expression*. If the method is of the type **void**, the **return** statement can be omitted. This statement takes the following form:

```
return [expression];
```

where:

expression
The value returned by a method. The *expression* is not used with methods of the type **void**.

Example

In the following example, the method `A()` returns the variable `Area` as a **double** value.

```
// statements_return.cs
using System;
class ReturnTest
{
   static double CalculateArea(int r)
   {
      double area;
      area = r*r*Math.PI;
      return area;
   }

   public static void Main()
   {
      int radius = 5;
      Console.WriteLine("The area is {0:0.00}", CalculateArea(radius));
   }
}
```

Output

```
The area is 78.54
```

See Also

C# Keywords | Compare to C++ | Jump Statements | C. Grammar

Exception Handling Statements

C# provides built-in support for handling anomalous situations, known as exceptions, which may occur during the execution of your program. These exceptions are handled by code that is outside the normal flow of control. The **try**, **throw**, **catch**, and **finally** keywords implement exception handling.

The following exception handling topics are explained in this section:

- throw
- try-catch
- try-finally
- try-catch-finally

See Also

C# Keywords | Statements | 8.10 The try statement

throw

The **throw** statement is used to signal the occurrence of an anomalous situation (exception) during the program execution. It takes the following form:

```
throw [expression];
```

where:

expression
The exception object. This is omitted when rethrowing the current exception object in a **catch** clause.

Remarks

The thrown exception is an object whose class is derived from **System.Exception**, for example:

```
class MyException : System.Exception {}
throw new MyException();
```

Usually the **throw** statement is used with try-catch or try-finally statements. When an exception is thrown, the program looks for the **catch** statement that handles this exception.

You can also rethrow a caught exception using the **throw** statement. For more information and examples, see try-catch, 8.10 The try statement and Throwing Exceptions.

Example

This example demonstrates how to throw an exception using the **throw** statement.

```
// throw example
using System;
public class ThrowTest
{
   public static void Main()
   {
      string s = null;

      if (s == null)
      {
         throw(new ArgumentNullException());
      }

      Console.Write("The string s is null"); // not executed
   }
}
```

Output

The following exception occurs:

```
System.ArgumentNullException
```

Example

See the try-catch, try-finally, and try-catch-finally examples.

See Also

Compare to C++ | C# Keywords | Exception Handling Statements | Throwing Exceptions | C. Grammar

try-catch

The try-catch statement consists of a **try** block followed by one or more **catch** clauses, which specify handlers for different exceptions. This statement takes one of the following forms:

```
try try-block
catch (exception-declaration-1) catch-block-1
catch (exception-declaration-2) catch-block-2
...
try try-block catch catch-block
```

where:

try-block
Contains the code segment expected to raise the exception.
exception-declaration, *exception-declaration-1*, *exception-declaration-2*
The exception object declaration.
catch-block, *catch-block-1*, *catch-block-2*
Contains the exception handler.

Remarks

The *try-block* contains the guarded code block that may cause the exception. The block is executed until an exception is thrown or it is completed successfully. For example, the following attempt to cast a **null** object raises the **NullReferenceException** exception:

```
object o2 = null;
try
{
   int i2 = (int) o2;   // Error
}
```

The **catch** clause can be used without arguments, in which case it catches any type of exception, and referred to as the general catch clause. It can also take an object argument derived from **System.Exception**, in which case it handles a specific exception. For example:

```
catch (InvalidCastException e)
{
}
```

It is possible to use more than one specific **catch** clause in the same try-catch statement. In this case, the order of the **catch** clauses is important because the **catch** clauses are examined in order. Catch the more specific exceptions before the less specific ones.

A throw statement can be used in the **catch** block to rethrow the exception, which has been caught by the **catch** statement. For example:

```
catch (InvalidCastException e)
{
   throw (e);   // Rethrowing exception e
}
```

If you want to rethrow the exception currently handled by a parameterless **catch** clause, use the **throw** statement without arguments. For example:

```
catch
{
   throw;
}
```

When inside a **try** block, only initialize variables that are declared therein; otherwise, an exception can occur before the execution of the block is completed. For example, in the following code example, the variable `x` is initialized inside the **try** block. An attempt to use this variable outside the **try** block in the `Write(x)` statement will generate the compiler error: Use of unassigned local variable.

```
public static void Main()
{
   int x;
   try
   {
      x = 123;   //   Don't do that.
      // ...
   }
   catch
   {
      // ...
   }
   Console.Write(x);   // Error: Use of unassigned local variable 'x'.
}
```

For more information on **catch**, see try-catch-finally.

Example

In this example, the **try** block contains a call to the method `MyFn()` that may cause an exception. The **catch** clause contains the exception handler that simply displays a message on the screen. When the **throw** statement is called from inside `MyFn()`, the system looks for the **catch** statement and displays the message `Exception caught`.

```
// Rethrowing exceptions:
using System;
class MyClass
{
   public static void Main()
   {
      MyClass x = new MyClass();
      try
      {
         string s = null;
         x.MyFn(s);
      }

      catch (Exception e)
      {
         Console.WriteLine("{0} Exception caught.", e);
      }
   }

   public void MyFn(string s)
   {
      if (s == null)
         throw(new ArgumentNullException());
   }
}
```

Output

The following exception occurs:

```
System.ArgumentNullException
```

Example

In this example, two catch statements are used. The most specific exception, which comes first, is caught.

```
// Ordering catch clauses
using System;
class MyClass
{
   public static void Main()
   {
      MyClass x = new MyClass();
      try
      {
         string s = null;
         x.MyFn(s);
      }

      // Most specific:
      catch (ArgumentNullException e)
      {
         Console.WriteLine("{0} First exception caught.", e);
      }

      // Least specific:
      catch (Exception e)
      {
         Console.WriteLine("{0} Second exception caught.", e);
      }

   }

   public void MyFn(string s)
   {
      if (s == null)
         throw new ArgumentNullException();
   }
}
```

Output

The following exception occurs:

```
System.ArgumentNullException
```

In the preceding example, if you start with the least specific catch clause, you will get the error message:

```
A previous catch clause already catches all exceptions of this or a super type ('System.Exception')
```

However, to catch the least specific exception, replace the throw statement by the following one:

```
throw new Exception();
```

See Also

C# Keywords | Compare to C++ | Exception Handling Statements | throw | try-finally | Throwing Exceptions | C. Grammar

try-finally

The **finally** block is useful for cleaning up any resources allocated in the try block. Control is always passed to the finally block regardless of how the try block exits. This statement takes the following form:

```
try try-block finally finally-block
```

where:

try-block
Contains the code segment expected to raise the exception.

finally-block
Contains the exception handler and the cleanup code.

Remarks

Whereas **catch** is used to handle exceptions that occur in a statement block, **finally** is used to guarantee a statement block of code executes regardless of how the preceding **try** block is exited.

Example

In this example, there is one invalid conversion statement that causes an exception. When you run the program, you get a run-time error message, but the **finally** clause will still be executed and display the output.

```
// try-finally
using System;
public class TestTryFinally
{
   public static void Main()
   {
      int i = 123;
      string s = "Some string";
      object o = s;

      try
      {
         // Invalid conversion; o contains a string not an int
         i = (int) o;
      }

      finally
      {
         Console.Write("i = {0}", i);
      }
   }
}
```

Output

The following exception occurs:

```
System.InvalidCastException
```

Although an exception was caught, the output statement included in the **finally** block will still be executed, that is:

```
i = 123
```

For more information on **finally**, see try-catch-finally.

See Also

try-catch-finally

A common usage of **catch** and **finally** together is to obtain and use resources in a **try** block, deal with exceptional circumstances in a **catch** block, and release the resources in the **finally** block.

Example

```
// try-catch-finally
using System;
public class EHClass
{
   public static void Main ()
   {
      try
      {
         Console.WriteLine("Executing the try statement.");
         throw new NullReferenceException();
      }

      catch(NullReferenceException e)
      {
         Console.WriteLine("{0} Caught exception #1.", e);
      }

      catch
      {
         Console.WriteLine("Caught exception #2.");
      }

      finally
      {
         Console.WriteLine("Executing finally block.");
      }
   }
}
```

Output

```
Executing the try statement.
System.NullReferenceException: Attempted to dereference a null object reference.
   at EHClass.Main() Caught exception #1.
Executing finally block.
```

For more information and examples on rethrowing exceptions, see try-catch, 8.10 The try statement and Throwing Exceptions.

See Also

C# Keywords | Compare to C++ | Exception Handling Statements | throw | Throwing Exceptions | C. Grammar

Checked and Unchecked

C# statements can execute in either checked or unchecked context. In a checked context, arithmetic overflow raises an exception. In an unchecked context, arithmetic overflow is ignored and the result is truncated.

- checked Specify checked context.
- unchecked Specify unchecked context.

If neither **checked** nor **unchecked** is specified, the default context depends on external factors such as compiler options.

The following operations are affected by the overflow checking:

- Expressions using the following predefined operators on integral types:

  ++ — - (unary) + - * /

- Explicit numeric conversions between integral types.

The /checked compiler option lets you specify checked or unchecked context for all integer arithmetic statements that are not explicitly in the scope of a **checked** or **unchecked** keyword.

See Also

C# Keywords | Statements | 8.11 The checked and unchecked statements

checked

The **checked** keyword is used to control the overflow-checking context for integral-type arithmetic operations and conversions. It can be used as an operator or a statement according to the following forms.

The **checked** statement:

```
checked block
```

The **checked** operator:

```
checked (expression)
```

where:

block
The statement block that contains the expressions to be evaluated in a checked context.

expression
The expression to be evaluated in a checked context. Notice that the expression must be in parentheses ( ).

Remarks

In a checked context, if an expression produces a value that is outside the range of the destination type, the result depends on whether the expression is constant or non-constant. Constant expressions cause compile time errors, while non-constant expressions are evaluated at run time and raise exceptions.

If neither **checked** nor unchecked is used, a constant expression uses the default overflow checking at compile time, which is **checked**. Otherwise, if the expression is non-constant, the run-time overflow checking depends on other factors such as compiler options and environment configuration.

The following three examples demonstrate the **checked** and **unchecked** operators on non-constant expressions. All use the same algorithm, but different checking contexts. The overflow checking is evaluated at run time.

- Example 1: Using a **checked** expression.
- Example 2: Using an **unchecked** expression.
- Example 3: Using default overflow checking.

Only the first example throws an overflow exception at run time, in which case, you have the option to go to the debugging mode, or abort the program execution. The other two examples produce truncated values.

See also the unchecked examples on using the **checked** and **unchecked** statements.

Example 1

```
// statements_checked.cs
// The overflow of non-constant expressions is checked at run time
using System;

class OverFlowTest
{
   static short x = 32767;   // Max short value
   static short y = 32767;

   // Using a checked expression
   public static int myMethodCh()
   {
      int z = 0;

      try
      {
        z =  checked((short)(x + y));
      }
      catch (System.OverflowException e)
```

```
      {
        System.Console.WriteLine(e.ToString());
      }
      return z;   // Throws the exception OverflowException
   }

   public static void Main()
   {
      Console.WriteLine("Checked output value is: {0}", myMethodCh());
   }
}
```

Sample Output

When you run the program, it throws the exception `OverflowException`. You can debug the program or abort execution.

```
System.OverflowException: An exception of type System.OverflowException was thrown.
   at OverFlowTest.myMethodCh()
Checked output value is: 0
```

Example 2

```
// statements_checked2.cs
// Using unchecked expressions
// The overflow of non-constant expressions is checked at run time
using System;

class OverFlowTest
{
   static short x = 32767;   // Max short value
   static short y = 32767;

   public static int myMethodUnch()
   {
      int z =  unchecked((short)(x + y));
      return z;   // Returns -2
   }

   public static void Main()
   {
      Console.WriteLine("Unchecked value is: {0}", myMethodUnch());
   }
}
```

Output

```
Unchecked value is: -2
```

Example 3

```
// statements_checked3.cs
// Using default overflow checking
// The overflow of non-constant expressions is checked at run time
using System;

class OverFlowTest
{
   static short x = 32767;   // Max short value
   static short y = 32767;

   public static int myMethodUnch()
```

```
    {
       int z =  (short)(x + y);
       return z;   // Returns -2
    }

    public static void Main()
    {
       Console.WriteLine("Default checking ouput value is: {0}", myMethodUnch());
    }
 }
```

Output

```
Default checking ouput value is: -2
```

See Also

C# Keywords | Checked and Unchecked | unchecked

unchecked

The **unchecked** keyword is used to control the overflow-checking context for integral-type arithmetic operations and conversions. It can be used as an operator or a statement according to the following forms.

The **unchecked** statement:

```
unchecked block
```

The **unchecked** operator:

```
unchecked (expression)
```

where:

block
The statement block that contains the expressions to be evaluated in an unchecked context.

expression
The expression to be evaluated in an unchecked context. Notice that the expression must be in parentheses ( ).

Remarks

In an unchecked context, if an expression produces a value that is outside the range of the destination type, the result is truncated.

If neither checked nor **unchecked** is used, a constant expression uses the default overflow checking at compile time, which is **checked**. Otherwise, if the expression is non-constant, the run-time overflow checking depends on other factors such as compiler options and environment configuration.

The following are three examples demonstrating the checked and unchecked statements. All use the same algorithm, but different checking contexts. The three examples are using constant expressions, for which the overflow checking is evaluated at compile time.

- Example 1: Using the **unchecked** statement.
- Example 2: Using default overflow checking.
- Example 3: Using the **checked** statement.

Only the **unchecked** statement produces a truncated value. The other two statements produce compile-time errors.

See also the checked examples on using the **checked** and **unchecked** operators.

Example 1

```
// statements_unchecked.cs
// Using the unchecked statement with constant expressions
// Overflow is checked at compile time
using System;

class TestClass
{
   const int x = 2147483647;   // Max int
   const int y = 2;

   public int MethodUnCh()
   {
      // Unchecked statement:
      unchecked
      {
         int z = x * y;
         return z;   // Returns -2
      }
   }

   public static void Main()
```

```
    {
       TestClass myObject = new TestClass();
       Console.WriteLine("Unchecked output value: {0}",
          myObject.MethodUnCh());
    }
 }
```

Output

```
Unchecked output value: -2
```

Example 2

```
// statements_unchecked2.cs
// CS0220 expected
// Using default overflow checking with constant expressions
// Overflow is checked at compile time
using System;

class TestClass
{
   const int x = 2147483647;   // Max int
   const int y = 2;

   public int MethodDef()
   {
      // Using default overflow checking:
      int z = x * y;
      return z;   // Compiler error
   }

   public static void Main()
   {
      TestClass myObject = new TestClass();
      Console.WriteLine("Default checking value: {0}",
         myObject.MethodDef());
   }
}
```

Output

The warning: `The operation overflows at compile time in checked mode.`

Example 3

```
// statements_unchecked3.cs
// CS0220 expected
// Using the checked statement with constant expressions
// Overflow is checked at compile time
using System;

class TestClass
{
   const int x = 2147483647;   // Max int
   const int y = 2;

   public int MethodCh()
   {
      // Checked statement:
      checked
      {
         int z = (x * y);
```

```
            return z;   // Compiler error
        }
    }

    public static void Main()
    {
        TestClass myObject = new TestClass();
        Console.WriteLine("Checked value: {0}", myObject.MethodCh());
    }
}
```

Output

The warning: `The operation overflows at compile time in checked mode.`

See Also

C# Keywords | Checked and Unchecked | checked

fixed Statement

Prevents relocation of a variable by the garbage collector. It takes the following form:

```
fixed ( type* ptr = expr ) statement
```

where:

type
An unmanaged type or void.
ptr
A pointer name.
expr
An expression that is implicitly convertible to *type**.
statement
Executable statement or block.

Remarks

The **fixed** statement is only permitted in an unsafe context.

The **fixed** statement sets a pointer to a managed variable and "pins" that variable during the execution of *statement*. Without **fixed**, pointers to managed variables would be of little use since garbage collection could relocate the variables unpredictably. (In fact, the C# compiler will not allow you to set a pointer to a managed variable except in a **fixed** statement.)

```
// assume class Point { public int x, y; }
Point pt = new Point();    // pt is a managed variable, subject to g.c.
fixed ( int* p = &pt.x ){  // must use fixed to get address of pt.x and
    *p = 1;                //   pin pt in place while we use the pointer
}
```

You can initialize a pointer with the address of an array or a string:

```
fixed (int* p = arr) ...  // equivalent to p = &arr[0]
fixed (char* p = str) ... // equivalent to p = &str[0]
```

You can initialize multiple pointers, as long as they are all of the same type:

```
fixed (byte* ps = srcarray, pd = dstarray) {...}
```

To initialize pointers of different type, simply nest **fixed** statements:

```
fixed (int* p1 = &p.x)
   fixed (double* p2 = &array[5])
      // do something with p1 and p2
```

Pointers initialized in fixed statements cannot be modified.

After *statement* is executed, any pinned variables are unpinned and subject to garbage collection. Therefore, do not point to those variables outside the **fixed** statement.

In unsafe mode, you can allocate memory on the stack, where it is not subject to garbage collection and therefore does not need to be pinned. See stackalloc for more information.

Example

```
// statements_fixed.cs
// compile with: /unsafe
using System;
```

```
class Point {
   public int x, y;
}

class FixedTest
{
   // unsafe method: takes pointer to int
   unsafe static void SquarePtrParam (int* p)
   {
      *p *= *p;
   }

   unsafe public static void Main()
   {
      Point pt = new Point();
      pt.x = 5;
      pt.y = 6;
      // pin pt in place:
      fixed (int* p = &pt.x)
      {
         SquarePtrParam (p);
      }
      // pt now unpinned
      Console.WriteLine ("{0} {1}", pt.x, pt.y);
   }
}
```

Output

```
25 6
```

See Also

C# Keywords | unsafe | Unsafe Code Tutorial | A.6 The fixed statement

lock Statement

The **lock** keyword marks a statement block as a critical section by obtaining the mutual-exclusion lock for a given object, executing a statement, and then releasing the lock. This statement takes the following form:

```
lock(expression) statement_block
```

where:

expression
Specifies the object that you want to lock on. *expression* must be a reference type.

Typically, *expression* will either be this, if you want to protect an instance variable, or typeof**(***class***)**, if you want to protect a static variable (or if the critical section occurs in a static method in the given class).

statement_block
The statements of the critical section.

Remarks

lock ensures that one thread does not enter a critical section while another thread is in the critical section of code. If another thread attempts to enter a locked code, it will wait (block) until the object is released.

8.12 The lock statement discusses **lock**.

Example 1

The following sample shows a simple use of threads in C#.

```
// statements_lock.cs
using System;
using System.Threading;

class ThreadTest
{
   public void runme()
   {
      Console.WriteLine("runme called");
   }

   public static void Main()
   {
      ThreadTest b = new ThreadTest();
      Thread t = new Thread(new ThreadStart(b.runme));
      t.Start();
   }
}
```

Output

```
runme called
```

Example 2

The following sample uses threads and **lock**. As long as the **lock** statement is present, the statement block is a critical section and `balance` will never become a negative number.

```
// statements_lock2.cs
using System;
using System.Threading;
```

```
class Account
{
   int balance;

   Random r = new Random();

   public Account(int initial)
   {
      balance = initial;
   }

   int Withdraw(int amount)
   {

      // This condition will never be true unless the lock statement
      // is commented out:
      if (balance < 0)
      {
         throw new Exception("Negative Balance");
      }

      // Comment out the next line to see the effect of leaving out
      // the lock keyword:
      lock (this)
      {
         if (balance >= amount)
         {
            Console.WriteLine("Balance before Withdrawal :  " + balance);
            Console.WriteLine("Amount to Withdraw        : -" + amount);
            balance = balance - amount;
            Console.WriteLine("Balance after Withdrawal  :  " + balance);
            return amount;
         }
         else
         {
            return 0; // transaction rejected
         }
      }
   }

   public void DoTransactions()
   {
      for (int i = 0; i < 100; i++)
      {
         Withdraw(r.Next(1, 100));
      }
   }
}

class Test
{
   public static void Main()
   {
      Thread[] threads = new Thread[10];
      Account acc = new Account (1000);
      for (int i = 0; i < 10; i++)
      {
         Thread t = new Thread(new ThreadStart(acc.DoTransactions));
         threads[i] = t;
      }
      for (int i = 0; i < 10; i++)
      {
         threads[i].Start();
      }
   }
}
```

See Also

C# Keywords | Statements | Threading Tutorial | 8.12 The lock statement | MethodImplAttributes Enumeration

Method Parameters

If a parameter is declared for a method without **ref** or **out**, the parameter can have a value associated with it. That value can be changed in the method, but the changed value will not be retained when control passes back to the calling procedure. By using a method parameter keyword, you can change this behavior.

This section describes the keywords you can use when declaring method parameters:

- params
- ref
- out

See Also

C# Keywords

params

The **params** keyword lets you specify a method parameter that takes an argument where the number of arguments is variable.

No additional parameters are permitted after the **params** keyword in a method declaration, and only one **params** keyword is permitted in a method declaration.

Example

```
// cs_params.cs
using System;
public class MyClass
{

   public static void UseParams(params int[] list)
   {
      for ( int i = 0 ; i < list.Length ; i++ )
         Console.WriteLine(list[i]);
      Console.WriteLine();
   }

   public static void UseParams2(params object[] list)
   {
      for ( int i = 0 ; i < list.Length ; i++ )
         Console.WriteLine(list[i]);
      Console.WriteLine();
   }

   public static void Main()
   {
      UseParams(1, 2, 3);
      UseParams2(1, 'a', "test");

      int[] myarray = new int[3] {10,11,12};
      UseParams(myarray);
   }
}
```

Output

```
1
2
3

1
a
test

10
11
12
```

See Also

C# Keywords | Method Parameters

ref

The **ref** method parameter keyword on a method parameter causes a method to refer to the same variable that was passed into the method. Any changes made to the parameter in the method will be reflected in that variable when control passes back to the calling method.

To use a **ref** parameter, the argument must explicitly be passed to the method as a **ref** argument. The value of a **ref** argument will be passed to the **ref** parameter.

An argument passed to a **ref** parameter must first be initialized. Compare this to an out parameter, whose argument does not have to be explicitly initialized before being passed to an **out** parameter.

A property is not a variable and cannot be passed as a **ref** parameter.

An overload will occur if declarations of two methods differ only in their use of **ref**. However, it is not possible to define an overload that only differs by **ref** and **out**. For example, the following overload declarations are valid:

```
class MyClass
{
   public void MyMethod(int i) {i = 10;}
   public void MyMethod(ref int i) {i = 10;}
}
```

but the following overload declarations are invalid:

```
class MyClass
{
   public void MyMethod(out int i) {i = 10;}
   public void MyMethod(ref int i) {i = 10;}
}
```

For information on passing an array, see Passing Arrays Using ref and out.

Example

```
// cs_ref.cs
using System;
public class MyClass
{
   public static void TestRef(ref char i)
   {
      // The value of i will be changed in the calling method
      i = 'b';
   }

   public static void TestNoRef(char i)
   {
      // The value of i will be unchanged in the calling method
      i = 'c';
   }

   // This method passes a variable as a ref parameter; the value of the
   // variable is changed after control passes back to this method.
   // The same variable is passed as a value parameter; the value of the
   // variable is unchanged after control is passed back to this method.
   public static void Main()
   {

      char i = 'a';    // variable must be initialized
      TestRef(ref i);  // the arg must be passed as ref
      Console.WriteLine(i);
      TestNoRef(i);
      Console.WriteLine(i);
   }
```

```
}
```

Output

```
b
b
```

See Also

C# Keywords | Method Parameters

out

The **out** method parameter keyword on a method parameter causes a method to refer to the same variable that was passed into the method. Any changes made to the parameter in the method will be reflected in that variable when control passes back to the calling method.

Declaring an **out** method is useful when you want a method to return multiple values. A method that uses an **out** parameter can still return a value. A method can have more than one **out** parameter.

To use an **out** parameter, the argument must explicitly be passed to the method as an **out** argument. The value of an **out** argument will not be passed to the **out** parameter.

A variable passed as an **out** argument need not be initialized. However, the **out** parameter must be assigned a value before the method returns.

A property is not a variable and cannot be passed as an **out** parameter.

An overload will occur if declarations of two methods differ only in their use of **out**. However, it is not possible to define an overload that only differs by **ref** and **out**. For example, the following overload declarations are valid:

```
class MyClass
{
   public void MyMethod(int i) {i = 10;}
   public void MyMethod(out int i) {i = 10;}
}
```

while the following overload declarations are invalid:

```
class MyClass
{
   public void MyMethod(out int i) {i = 10;}
   public void MyMethod(ref int i) {i = 10;}
}
```

For information on passing an array, see Passing Arrays Using ref and out.

Example

```
// cs_out.cs
using System;
public class MyClass
{
   public static int TestOut(out char i)
   {
      i = 'b';
      return -1;
   }

   public static void Main()
   {
      char i;   // variable need not be initialized
      Console.WriteLine(TestOut(out i));
      Console.WriteLine(i);
   }
}
```

Output

```
-1
b
```

See Also

C# Keywords | Method Parameters

Namespaces

This section describes the keywords that are associated with using namespaces:

- namespace
- using

See Also

C# Keywords

namespace

The **namespace** keyword is used to declare a scope. This namespace scope lets you organize code and gives you a way to create globally-unique types.

```
namespace name[.name1] ...] {
   type-declarations
}
```

where:

name, *name1*
A namespace name can be any legal identifier. A namespace name can contain periods.
type-declarations
Within a namespace, you can declare one or more of the following types:

- another namespace
- class
- interface
- struct
- enum
- delegate

Remarks

Even if you do not explicitly declare one, a default namespace is created. This unnamed namespace, sometimes called the global namespace, is present in every file. Any identifier in the global namespace is available for use in a named namespace.

Namespaces implicitly have public access and this is not modifiable.

3.8 Namespace and type names and 9. Namespaces discuss namespaces in more detail.

See 7.3 Member lookup for information about how the compiler resolves names.

See Access Modifiers for a discussion of the access modifiers you can assign to elements within a namespace.

It is possible to define a namespace in two or more declarations. For example, the following sample defines both classes as part of namespace `MyCompany`:

```
namespace MyCompany.Proj1
{
   class MyClass
   {
   }
}

namespace MyCompany.Proj1
{
   class MyClass1
   {
   }
}
```

Example

The following example shows how to call a static method in a nested namespace.

```
// cs_namespace_keyword.cs
using System;
namespace SomeNameSpace
{
   public class MyClass
   {
```

```
        public static void Main()
        {
           Nested.NestedNameSpaceClass.SayHello();
        }
     }

     namespace Nested   // a nested namespace
     {
        public class NestedNameSpaceClass
        {
           public static void SayHello()
           {
              Console.WriteLine("Hello");
           }
        }
     }
  }
```

Output

```
Hello
```

See Also

C# Keywords | using

using

The **using** keyword has two major uses:

- using Directive Creates an alias for a namespace or imports types defined in other namespaces.
- using Statement Defines a scope at the end of which an object will be disposed.

See Also

C# Keywords | namespace

using Directive

The **using** directive has two uses:

- Create an alias for a namespace (a **using** alias).
- Permit the use of types in a namespace, such that, you do not have to qualify the use of a type in that namespace (a **using** directive).

```
using [alias = ]class_or_namespace;
```

where:

alias (optional)
A user-defined symbol that you want to represent a namespace. You will then be able to use *alias* to represent the namespace name.

class_or_namespace
The namespace name that you want to either use or alias, or the class name that you want to alias.

Remarks

Create a **using** alias to make it easier to qualify an identifier to a namespace or class.

Create a **using** directive to use the types in a namespace without having to specify the namespace. A **using** directive does not give you access to any namespaces that may be nested in the namespace you specify.

Namespaces come in two categories: user-defined and system-defined. User-defined namespaces are namespaces defined in your code. See the documentation for the .NET Framework for a list of the system-defined namespaces.

For examples on referencing methods in other assemblies, see Creating and Using C# DLLs.

Example

The following sample shows how to define and use a **using** alias for a namespace:

```
// cs_using_directive.cs
using MyAlias = MyCompany.Proj.Nested;  // define an alias to represent a namespace

namespace MyCompany.Proj
{
   public class MyClass
   {
      public static void DoNothing()
      {
      }
   }

   namespace Nested   // a nested namespace
   {
      public class ClassInNestedNameSpace
      {
         public static void SayHello()
         {
            System.Console.WriteLine("Hello");
         }
      }
   }

}

public class UnNestedClass
{
   public static void Main()
   {
      MyAlias.ClassInNestedNameSpace.SayHello();   // using alias
```

```
    }
}
```

Output

```
Hello
```

Example

The following sample shows how to define a **using** directive and a **using** alias for a class:

```
// cs_using_directive2.cs
using System;   // using directive
using AliasToMyClass = NameSpace1.MyClass;   // using alias for a class

namespace NameSpace1
{
   public class MyClass
   {
      public override string ToString()
      {
         return "You are in NameSpace1.MyClass";
      }
   }
}

namespace NameSpace2
{
   class MyClass
   {
   }
}

namespace NameSpace3
{
   using NameSpace1;   // using directive
   using NameSpace2;   // using directive

   class Test
   {
      public static void Main()
      {
         AliasToMyClass somevar = new AliasToMyClass();
         Console.WriteLine(somevar);
      }
   }
}
```

Output

```
You are in NameSpace1.MyClass
```

See Also

C# Keywords | namespace | using Statement

using Statement

The **using** statement defines a scope at the end of which an object will be disposed.

```
using (expression | type identifier = initializer) statement
```

where:

expression
An expression you want to call **Dispose** on upon exiting the **using** statement.
type
The type of *identifier*.
identifier
The name, or identifier, of the type *type*. It is possible to define more than one *identifier* of type *type*. Precede each *identifier* **=** *initializer* with a comma.
initializer
An expression that creates an object.
statement
The embedded statement or statements to executed.

Remarks

You create an instance in a **using** statement to ensure that **Dispose** is called on the object when the **using** statement is exited. A **using** statement can be exited either when the end of the **using** statement is reached or if, for example, an exception is thrown and control leaves the statement block before the end of the statement.

The object you instantiate must implement the **System.IDisposable** interface.

Example

```
// cs_using_statement.cs
// compile with /reference:System.Drawing.dll
using System.Drawing;
class a
{
   public static void Main()
   {
      using (Font MyFont = new Font("Arial", 10.0f), MyFont2 = new Font("Arial", 10.0f))
      {
         // use MyFont and MyFont2
      }   // compiler will call Dispose on MyFont and MyFont2

      Font MyFont3 = new Font("Arial", 10.0f);
      using (MyFont3)
      {
         // use MyFont3
      }   // compiler will call Dispose on MyFont3

   }
}
```

See Also

C# Keywords | using Directive | Implementing Finalize and Dispose to Clean Up Unmanaged Resources

Operator Keywords

This section introduces the following keywords:

- as Convert an object to a compatible type.
- is Check the run-time type of an object.
- new
    - new operator Create objects.
    - new modifier Hide an inherited member.
- sizeof Obtain the size of a type.
- typeof Obtain the **System.Type** object for a type.
- true Return the boolean value true.
- false Return the boolean value false.
- stackalloc Allocate a block of memory on the stack.

The following keywords, which can be used as operators and as statements, are covered in the Statements section:

- checked Specify checked context.
- unchecked Specify unchecked context.

See Also

C# Keywords | C# Operators

as

The **as** operator is used to perform conversions between compatible types. The **as** operator is used in an expression of the form:

```
expression as type
```

where:

expression
An expression of a reference type.
type
A reference type.

Remarks

The **as** operator is like a cast except that it yields null on conversion failure instead of raising an exception. More formally, an expression of the form:

```
expression as type
```

is equivalent to:

```
expression is type ? (type)expression : (type)null
```

except that `expression` is evaluated only once.

Note that the **as** operator only performs reference conversions and boxing conversions. The **as** operator cannot perform other conversions, such as user-defined conversions, which should instead be performed using cast expressions.

Example

```
// cs_keyword_as.cs
// The as operator
using System;
class MyClass1
{
}

class MyClass2
{
}

public class IsTest
{
   public static void Main()
   {
      object [] myObjects = new object[6];
      myObjects[0] = new MyClass1();
      myObjects[1] = new MyClass2();
      myObjects[2] = "hello";
      myObjects[3] = 123;
      myObjects[4] = 123.4;
      myObjects[5] = null;

      for (int i=0; i<myObjects.Length; ++i)
      {
         string s = myObjects[i] as string;
         Console.Write ("{0}:", i);
         if (s != null)
            Console.WriteLine ( "'" + s + "'" );
         else
            Console.WriteLine ( "not a string" );
```

```
        }
    }
}
```

Output

```
0:not a string
1:not a string
2:'hello'
3:not a string
4:not a string
5:not a string
```

See Also

C# Keywords | is | ?: | Operator Keywords

is

The **is** operator is used to check whether the run-time type of an object is compatible with a given type. The **is** operator is used in an expression of the form:

```
expression is type
```

Where:

expression
An expression of a reference type.
type
A type.

Remarks

An **is** expression evaluates to true if both of the following conditions are met:

- *expression* is not **null**.
- *expression* can be cast to *type*. That is, a cast expression of the form `(type)(expression)` will complete without throwing an exception. For more information, see 7.6.6 Cast expressions.

A compile-time warning will be issued if the expression `expression is type` is known to always be true or always be false.

The **is** operator cannot be overloaded.

Note that the **is** operator only considers reference conversions, boxing conversions, and unboxing conversions. Other conversions, such as user-defined conversions, are not considered by the **is** operator.

Example

```
// cs_keyword_is.cs
// The is operator
using System;
class Class1
{
}

class Class2
{
}

public class IsTest
{
   public static void Test (object o)
   {
      Class1 a;
      Class2 b;

      if (o is Class1)
      {
         Console.WriteLine ("o is Class1");
         a = (Class1)o;
         // do something with a
      }

      else if (o is Class2)
      {
         Console.WriteLine ("o is Class2");
         b = (Class2)o;
         // do something with b
      }

      else
```

```
        {
           Console.WriteLine ("o is neither Class1 nor Class2.");
        }
     }
     public static void Main()
     {
        Class1 c1 = new Class1();
        Class2 c2 = new Class2();
        Test (c1);
        Test (c2);
        Test ("a string");
     }
  }
```

Output

```
o is Class1
o is Class2
o is neither Class1 nor Class2.
```

See Also

C# Keywords | typeof | as | Operator Keywords

new

In C#, the **new** keyword can be used as an operator or as a modifier.

- new operator Used to create objects on the heap and invoke constructors.
- new modifier Used to hide an inherited member from a base class member.

new Operator

The **new** operator is used to create objects and invoke constructors, for example:

```
Class1 MyClass  = new Class1();
```

It is also used to invoke the default constructor for value types, for example:

```
int myInt = new int();
```

In the preceding statement, `myInt` is initialized to `0`, which is the default value for the type **int**. The statement has the same effect as:

```
int myInt = 0;
```

For a complete list of default values, see Default Values Table.

Remember that it is an error to declare a default constructor for a struct, because every value type implicitly has a public default constructor. It is possible though to declare parameterized constructors on a struct type.

Value-type objects such as structs are created on the stack, while reference-type objects such as classes are created on the heap.

The **new** operator cannot be overloaded.

If the new operator fails to allocate memory, it throws the exception **OutOfMemoryException**.

Example

In the following example, a struct object and a class object are created and initialized by using the **new** operator and then assigned values. The default and the assigned values are displayed

```
// cs_operator_new.cs
// The new operator
using System;
class NewTest
{
   struct MyStruct
   {
      public int x;
      public int y;
      public MyStruct (int x, int y)
      {
         this.x = x;
         this.y = y;
      }
   }

   class MyClass
   {
      public string name;
      public int id;

      public MyClass ()
      {
      }
```

```
      public MyClass (int id, string name)
      {
         this.id = id;
         this.name = name;
      }
   }

   public static void Main()
   {
      // Create objects using default constructors:
      MyStruct Location1 = new MyStruct();
      MyClass Employee1 = new MyClass();

      // Display values:
      Console.WriteLine("Default values:");
      Console.WriteLine("   Struct members: {0}, {1}",
         Location1.x, Location1.y);
      Console.WriteLine("   Class members: {0}, {1}",
         Employee1.name, Employee1.id);

      // Create objects using parameterized constructors::
      MyStruct Location2 = new MyStruct(10, 20);
      MyClass Employee2 = new MyClass(1234, "John Martin Smith");

      // Display values:
      Console.WriteLine("Assigned values:");
      Console.WriteLine("   Struct members: {0}, {1}",
         Location2.x, Location2.y);
      Console.WriteLine("   Class members: {0}, {1}",
         Employee2.name, Employee2.id);
   }
}
```

Output

```
Default values:
   Struct members: 0, 0
   Class members: , 0
Assigned values:
   Struct members: 10, 20
   Class members: John Martin Smith, 1234
```

Notice in the example that the default value of a string is **null**. Therefore, it was not displayed.

new Modifier

Use the **new** modifier to explicitly hide a member inherited from a base class. To hide an inherited member, declare it in the derived class using the same name, and modify it with the **new** modifier.

Consider the following class:

```
public class MyBaseC
{
   public int x;
   public void Invoke() {}
}
```

Declaring a member with the name `Invoke` in a derived class will hide the method `Invoke` in the base class, that is:

```
public class MyDerivedC : MyBaseC
{
   new public void Invoke() {}
}
```

However, the field `x` will not be affected because it is not hidden by a similar name.

Name hiding through inheritance takes one of the following forms:

- A constant, field, property, or type introduced in a class or struct hides all base class members with the same name.
- A method introduced in a class or struct hides properties, fields, and types, with the same name, in the base class. It also hides all base class methods with the same signature. For more information, see 3.6 Signatures and overloading.
- An indexer introduced in a class or struct hides all base class indexers with the same signature.

It is an error to use both **new** and override on the same member.

Using the **new** modifier in a declaration that does not hide an inherited member generates a warning.

For more information on hiding names, see 3.7.1 Name hiding.

For more information on fully qualified names, see 3.8.1 Fully qualified names.

Example

In this example, a base class, `MyBaseC`, and a derived class, `MyDerivedC`, use the same field name `x`, thus hiding the value of the inherited field. The example demonstrates the use of the **new** modifier. It also demonstrates how to access the hidden members of the base class by using the fully qualified names.

```
// cs_modifier_new.cs
// The new modifier
using System;
public class MyBaseC
{
   public static int x = 55;
   public static int y = 22;
}

public class MyDerivedC : MyBaseC
{
   new public static int x = 100;   // Name hiding
   public static void Main()
   {
      // Display the overlapping value of x:
      Console.WriteLine(x);

      // Access the hidden value of x:
      Console.WriteLine(MyBaseC.x);

      // Display the unhidden member y:
      Console.WriteLine(y);
   }
}
```

Output

```
100
55
22
```

If you remove the **new** modifier, the program will still compile and run, but you will get the warning:

```
The keyword new is required on 'MyDerivedC.x' because it hides inherited member 'MyBaseC.x'.
```

You can also use the **new** modifier to modify a nested type if the nested type is hiding another type, as demonstrated in the following example.

Example

In this example, a nested class, `MyClass`, hides a class with the same name in the base class. The example demonstrates using the **new** modifier to eliminate the warning message, as well as accessing the hidden class members by using the fully qualified names.

```
// cs_modifer_new_nested.cs
// Using the new modifier with nested types
using System;
public class MyBaseC
{
   public class MyClass
   {
      public int x = 200;
      public int y;
   }
}

public class MyDerivedC : MyBaseC
{
   new public class MyClass   // nested type hiding the base type members
   {
      public int x = 100;
      public int y;
      public int z;
   }

   public static void Main()
   {
      // Creating object from the overlapping class:
      MyClass S1  = new MyClass();

      // Creating object from the hidden class:
      MyBaseC.MyClass S2 = new MyBaseC.MyClass();

      Console.WriteLine(S1.x);
      Console.WriteLine(S2.x);
   }
}
```

Output

```
100
200
```

See Also

C# Keywords | Operator Keywords | Modifiers

sizeof

The **sizeof** operator is used to obtain the size in bytes for a value type. A **sizeof** expression takes the form:

```
sizeof(type)
```

where:

type
The value type for which the size is obtained.

Remarks

The **sizeof** operator can be applied only to value types, not reference types.

The **sizeof** operator can only be used in the unsafe mode.

The **sizeof** operator cannot be overloaded.

Example

```
// cs_operator_sizeof.cs
// compile with: /unsafe

// Using the sizeof operator
using System;
class SizeClass
{
   // Notice the unsafe declaration of the method:
   unsafe public static void SizesOf()
   {
      Console.WriteLine("The size of short is {0}.", sizeof(short));
      Console.WriteLine("The size of int is {0}.", sizeof(int));
      Console.WriteLine("The size of long is {0}.", sizeof(long));
   }
}

class MainClass
{
   public static void Main()
   {
      SizeClass.SizesOf();
   }
}
```

Output

```
The size of short is 2.
The size of int is 4.
The size of long is 8.
```

See Also

C# Keywords | Operator Keywords

typeof

The **typeof** operator is used to obtain the **System.Type** object for a type. A **typeof** expression takes the form:

```
typeof(type)
```

where:

type
The type for which the **System.Type** object is obtained.

Remarks

The **typeof** operator cannot be overloaded.

To obtain the run-time type of an expression, you can use the .NET Framework method **GetType**.

Example

```
// cs_operator_typeof.cs
// Using typeof operator
using System;
using System.Reflection;

public class MyClass
{
   public int intI;
   public void MyMeth()
   {
   }

   public static void Main()
   {
      Type t = typeof(MyClass);

      // alternatively, you could use
      // MyClass t1 = new MyClass();
      // Type t = t1.GetType();

      MethodInfo[] x = t.GetMethods();
      foreach (MethodInfo xtemp in x)
      {
         Console.WriteLine(xtemp.ToString());
      }

      Console.WriteLine();

      MemberInfo[] x2 = t.GetMembers();
      foreach (MemberInfo xtemp2 in x2)
      {
         Console.WriteLine(xtemp2.ToString());
      }
   }
}
```

Output

```
Int32 GetHashCode()
Boolean Equals(System.Object)
System.String ToString()
Void MyMeth()
Void Main()
System.Type GetType()
```

```
Int32 intI
Int32 GetHashCode()
Boolean Equals(System.Object)
System.String ToString()
Void MyMeth()
Void Main()
System.Type GetType()
Void .ctor()
```

Example

```
// cs_operator_typeof2.cs
// Using GetType method
using System;
class GetTypeTest
{
   public static void Main()
   {
      int radius = 3;
      Console.WriteLine("Area = {0}", radius*radius*Math.PI);
      Console.WriteLine("The type is {0}",
               (radius*radius*Math.PI).GetType());
   }
}
```

Output

```
Area = 28.2743338823081
The type is System.Double
```

See Also

C# Keywords | is | Operator Keywords

true

In C#, the **true** keyword can be used as an overloaded operator or as a literal:

- true Operator
- true Literal

true Operator

User-defined types can define a **true** operator that returns the bool value true to indicate true and returns false otherwise. This is useful for types that represent true, false, and null (neither true nor false), as used in databases.

Such types can be used for the controlling expression in if, do, while, and for statements and in conditional expressions.

If a type defines operator **true**, it must also define operator false.

A type cannot directly overload the conditional logical operators (&& and ||), but an equivalent effect can be achieved by overloading the regular logical operators and operators **true** and **false** (see 7.11.2 User-defined conditional logical operators).

Example

See the example in 11.4.2 Database boolean type.

true Literal

The **true** keyword is a literal of type bool representing the boolean value true.

Example

```
// cs_keyword_true.cs
using System;
class test
{
   public static void Main()
   {
      bool a = true;
      Console.WriteLine( a ? "yes" : "no" );
   }
}
```

Output

```
yes
```

See Also

C# Keywords | C# Operators | false | Operator Overloading Tutorial

false

In C#, the **false** keyword can be used as an overloaded operator or as a literal:

- false Operator
- false Literal

false Operator

User-defined types can define a **false** operator that returns the bool value true to indicate false and returns false otherwise. This is useful for types that represent true, false, and null (neither true nor false), as used in databases.

Such types can be used for the controlling expression in if, do, while, and for statements and in conditional expressions.

If a type defines operator **false**, it must also define operator true.

A type cannot directly overload the conditional logical operators (&& and ||), but an equivalent effect can be achieved by overloading the regular logical operators and operators **true** and **false** (see 7.11.2 User-defined conditional logical operators).

Example

See the example in 11.4.2 Database boolean type.

false Literal

The **false** keyword is a literal of type bool representing the boolean value false.

Example

```
// cs_keyword_false.cs
using System;
class test
{
   public static void Main()
   {
      bool a = false;
      Console.WriteLine( a ? "yes" : "no" );
   }
}
```

Output

```
no
```

See Also

C# Keywords | C# Operators | true | Operator Overloading Tutorial

stackalloc

Allocates a block of memory on the stack.

```
type * ptr = stackalloc type [ expr ];
```

where:

type
An unmanaged type.
ptr
A pointer name.
expr
An integral expression.

Remarks

A block of memory of sufficient size to contain *expr* elements of type *type* is allocated on the stack, not the heap; the address of the block is stored in pointer *ptr*. This memory is not subject to garbage collection and therefore does not have to be pinned (via fixed). The lifetime of the memory block is limited to the lifetime of the method in which it is defined.

stackalloc is only valid in local variable initializers.

Because A.2 Pointer types are involved, **stackalloc** requires unsafe context.

stackalloc is similar to _alloca in the C run-time library.

Example

```
// cs_keyword_stackalloc.cs
// compile with: /unsafe
using System; class Test
{
   public static unsafe void Main()
   {
      int* fib = stackalloc int[100];
      int* p = fib;
      *p++ = *p++ = 1;
      for (int i=2; i<100; ++i, ++p)
         *p = p[-1] + p[-2];
      for (int i=0; i<10; ++i)
         Console.WriteLine (fib[i]);
   }
}
```

Output

```
1
1
2
3
5
8
13
21
34
55
```

See Also

C# Keywords | Operator Keywords

Conversion Keywords

This section describes keywords used in type conversions:

- explicit
- implicit
- operator

See Also

C# Keywords

explicit

The **explicit** keyword is used to declare an explicit user-defined type conversion operator (6.4.4 User-defined explicit conversions). For example:

```
class MyType
{
   public static explicit operator MyType(int i)
   {
      // code to convert from int to MyType
   }
}
```

Unlike implicit conversion, explicit conversion operators must be invoked via a cast.

```
int i;
MyType x = (MyType)i;   // int-to-MyType requires cast
```

Omitting the cast results in a compile-time error.

If a conversion operation can cause exceptions or lose information, you should mark it **explicit**. This prevents the compiler from silently invoking the conversion operation with possibly unforeseen consequences.

Example

The following example defines a struct, `Digit`, that represents a single decimal digit. An operator is defined for conversions from `byte` to `Digit`, but because not all bytes can be converted to a `Digit`, the conversion is explicit.

```
// cs_keyword_explicit.cs
using System;
struct Digit
{
   byte value;
   public Digit(byte value)
   {
      if (value>9) throw new ArgumentException();
      this.value = value;
   }

   // define explicit byte-to-Digit conversion operator:
   public static explicit operator Digit(byte b)
   {
      Digit d = new Digit(b);
      Console.WriteLine("conversion occurred");
      return d;
   }
}

class Test
{
   public static void Main()
   {
     try
      {
      byte b = 3;
      Digit d = (Digit)b; // explicit conversion
      }

      catch (Exception e)
      {
         Console.WriteLine("{0} Exception caught.", e);
      }
   }
}
```

Output

```
conversion occurred
```

See Also

C# Keywords | implicit | operator | User-Defined Conversions Tutorial

implicit

The **implicit** keyword is used to declare an implicit user-defined type conversion operator (6.4.3 User-defined implicit conversions). For example:

```
class MyType
{
   public static implicit operator int(MyType m)
   {
      // code to convert from MyType to int
   }
}
```

Implicit conversion operators can be called implicitly, without being specified by explicit casts in the source code.

```
MyType x;
int i = x; // implicitly call MyType's MyType-to-int conversion operator
```

By eliminating unnecessary casts, implicit conversions can improve source code readability. However, because implicit conversions can occur without the programmer's specifying them, care must be taken to prevent unpleasant surprises. In general, implicit conversion operators should never throw exceptions and never lose information so that they can be used safely without the programmer's awareness. If a conversion operator cannot meet those criteria, it should be marked **explicit**.

Example

The following example defines a struct, `Digit`, that represents a single decimal digit. An operator is defined for conversions from `Digit` to `byte`, and because any `Digit` can be converted to a `byte`, the conversion is implicit.

```
// cs_keyword_implicit.cs
using System;
struct Digit
{
   byte value;

   public Digit(byte value)
   {
      if (value > 9) throw new ArgumentException();
      this.value = value;
   }

   // define implicit Digit-to-byte conversion operator:
   public static implicit operator byte(Digit d)
   {
      Console.WriteLine( "conversion occurred" );
      return d.value;
   }
}

class Test
{
   public static void Main()
   {
      Digit d = new Digit(3);

      // implicit (no cast) conversion from Digit to byte
      byte b = d;
   }
}
```

Output

```
conversion occurred
```

See Also

C# Keywords | explicit | operator | User-Defined Conversions Tutorial

operator

The **operator** keyword is used to declare an operator in a class or struct declaration. An operator declaration can take one of the following four forms:

```
public static result-type operator unary-operator ( op-type operand )
public static result-type operator binary-operator (
   op-type operand,
   op-type2 operand2
)
public static implicit operator conv-type-out ( conv-type-in operand )
public static explicit operator conv-type-out ( conv-type-in operand )
```

where:

result-type
The type of the operator's result.
unary-operator
One of: `+` `-` `!` `~` `++` `—` `true` `false`
op-type
The type of the first (or only) parameter.
operand
The name of the first (or only) parameter.
binary-operator
One of: `+` `-` `*` `/` `%` `&` `|` `^` `<<` `>>` `==` `!=` `>` `<` `>=` `<=`
op-type2
The type of the second parameter.
operand2
The name of the second parameter.
conv-type-out
The target type of a type conversion operator.
conv-type-in
The type of the input to a type conversion operator.

Remarks

The first two forms declare user-defined operators that overload built-in operators. Note that not all the built-in operators can be overloaded (see Overloadable Operators). At least one of *op-type* and *op-type2* must be the enclosing type (that is, the type of which the operator is a member). This prevents redefining the integer addition operator, for instance.

The last two forms declare conversion operators. Exactly one of *conv-type-in* and *conv-type-out* must be the enclosing type (that is, a conversion operator can only convert from its enclosing type to some other type, or from some other type to its enclosing type).

Operators can only take value parameters, not ref or out parameters.

Any operator declaration can be preceded by an optional attribute list.

For more information, refer to 10.9 Operators.

Example

The following is an extremely simplified class for rational numbers. It overloads the + and * operators to perform fractional addition and multiplication, and also provides an operator that converts fractions to doubles.

```
// cs_keyword_operator.cs
using System;
class Fraction
{
   int num, den;
   public Fraction(int num, int den)
   {
      this.num = num;
```

```
        this.den = den;
    }

    // overload operator +
    public static Fraction operator +(Fraction a, Fraction b)
    {
        return new Fraction(a.num * b.den + b.num * a.den,
            a.den * b.den);
    }

    // overload operator *
    public static Fraction operator *(Fraction a, Fraction b)
    {
        return new Fraction(a.num * b.num, a.den * b.den);
    }

    // define operator double
    public static implicit operator double(Fraction f)
    {
        return (double)f.num / f.den;
    }
}

class Test
{
    public static void Main()
    {
        Fraction a = new Fraction(1, 2);
        Fraction b = new Fraction(3, 7);
        Fraction c = new Fraction(2, 3);
        Console.WriteLine((double)(a * b + c));
    }
}
```

Output

```
0.880952380952381
```

See Also

C# Keywords | implicit | explicit | User-Defined Conversions Tutorial

Access Keywords

This section introduces the following access keywords:

- base

  Accesses the members of the base class.

- this

  Refers to the current instance of the class.

See Also

C# Keywords

base

The **base** keyword is used to access members of the base class from within a derived class:

- Call a method on the base class that has been overridden by another method.
- Specify which base-class constructor should be called when creating instances of the derived class.

A base class access is permitted only in a constructor, an instance method, or an instance property accessor.

It is an error to use the **base** keyword from within a static method.

Example

In this example, both the base class, `Person`, and the derived class, `Employee`, have a method named `Getinfo`. By using the **base** keyword, it is possible to call the `Getinfo` method on the base class, from within the derived class.

```
// keywords_base.cs
// Accessing base class members
using System;
   public class Person
   {
      protected string ssn = "444-55-6666";
      protected string name = "John L. Malgraine";

      public virtual void GetInfo()
      {
         Console.WriteLine("Name: {0}", name);
         Console.WriteLine("SSN: {0}", ssn);
      }
   }
   class Employee: Person
   {
      public string id = "ABC567EFG";

      public override void GetInfo()
      {
         // Calling the base class GetInfo method:
         base.GetInfo();
         Console.WriteLine("Employee ID: {0}", id);
      }
   }

class TestClass {
   public static void Main()
   {
      Employee E = new Employee();
      E.GetInfo();
   }
}
```

Output

```
Name: John L. Malgraine
SSN: 444-55-6666
Employee ID: ABC567EFG
```

For additional examples, see new, virtual, and override.

Example

This example shows how to specify the base-class constructor called when creating instances of a derived class.

```
// keywords_base2.cs
using System;
public class MyBase
{
   int num;

   public MyBase()
   {
      Console.WriteLine("in MyBase()");
   }

   public MyBase(int i )
   {
      num = i;
      Console.WriteLine("in MyBase(int i)");
   }

   public int GetNum()
   {
      return num;
   }
}

public class MyDerived: MyBase
{
   // This constructor will call MyBase.MyBase()
   public MyDerived() : base()
   {
   }

    // This constructor will call MyBase.MyBase(int i)
   public MyDerived(int i) : base(i)
   {
   }

   public static void Main()
   {
      MyDerived md = new MyDerived();
      MyDerived md1 = new MyDerived(1);
   }
}
```

Output

```
in MyBase()
in MyBase(int i)
```

See Also

C# Keywords | this

this

The **this** keyword refers to the current instance of the class. Static member functions do not have a **this** pointer. The **this** keyword can be used to access members from within constructors, instance methods, and instance accessors.

The following are common uses of **this**:

- To qualify members hidden by similar names, for example:

```
public Employee(string name, string alias)
{
   this.name = name;
   this.alias = alias;
}
```

- To pass an object as a parameter to other methods, for example:

```
CalcTax(this);
```

- To declare indexers, for example:

```
public int this [int param]
{
      get
      {
         return array[param];
      }
      set
      {
         array[param] = value;
      }
   }
```

It is an error to refer to **this** in a static method, static property accessor, or variable initializer of a field declaration.

Example

In this example, **this** is used to qualify the `Employee` class members, `name` and `alias`, which are hidden by similar names. It is also used to pass an object to the method `CalcTax`, which belongs to another class.

```
// keywords_this.cs
// this example
using System;
public class Employee
{
   public string name;
   public string alias;
   public decimal salary = 3000.00m;

   // Constructor:
   public Employee(string name, string alias)
   {
      // Use this to qualify the fields, name and alias:
      this.name = name;
      this.alias = alias;
   }

   // Printing method:
   public void printEmployee()
   {
      Console.WriteLine("Name: {0}\nAlias: {1}", name, alias);
      // Passing the object to the CalcTax method by using this:
```

```
        Console.WriteLine("Taxes: {0:C}", Tax.CalcTax(this));
    }
}
public class Tax
{
    public static decimal CalcTax(Employee E)
    {
        return (0.08m*(E.salary));
    }
}

public class MainClass
{
    public static void Main()
    {
        // Create objects:
        Employee E1 = new Employee ("John M. Trainer", "jtrainer");

        // Display results:
        E1.printEmployee();
    }
}
```

Output

```
Name: John M. Trainer
Alias: jtrainer
Taxes: $240.00
```

For additional examples, see class and struct.

See Also

C# Keywords | base

Literal Keywords

C# has the following literal keywords:

- null
- true
- false

See Also

C# Keywords

null

The **null** keyword is a literal that represents a null reference, one that does not refer to any object. **null** is the default value of reference-type variables.

See Also

C# Keywords | Literal keywords | Default Values Table

C# Operators

C# provides a large set of operators, which are symbols that specify which operations to perform in an expression. C# predefines the usual arithmetic and logical operators, as well as a variety of others as shown in the following table. In addition, many operators can be overloaded by the user, thus changing their meaning when applied to a user-defined type.

Operator category	Operators
Arithmetic	+ - * / %
Logical (boolean and bitwise)	& | ^ ! ~ && || true false
String concatenation	+
Increment, decrement	++ --
Shift	<< >>
Relational	== != < > <= >=
Assignment	= += -= *= /= %= &= |= ^= <<= >>=
Member access	.
Indexing	[]
Cast	()
Conditional	?:
Delegate concatenation and removal	+ -
Object creation	new
Type information	as is sizeof typeof
Overflow exception control	checked unchecked
Indirection and Address	* -> [] &

Arithmetic Overflow

The arithmetic operators (+, -, *, /) can produce results that are outside the range of possible values for the numeric type involved. You should refer to the *C# Language Reference* section on a particular operator for details, but in general:

- Integer arithmetic overflow either throws an OverflowException or discards the most significant bits of the result (see below). Integer division by zero always throws a DivideByZeroException.
- Floating-point arithmetic overflow or division by zero never throws an exception, because floating-point types are based on IEEE 754 and so have provisions for representing infinity and NaN (Not a Number).
- Decimal arithmetic overflow always throws an OverflowException. **Decimal** division by zero always throws a DivideByZeroException.

When integer overflow occurs, what happens depends on the execution context, which can be checked or unchecked. In a checked context, an OverflowException is thrown. In an unchecked context, the most significant bits of the result are discarded and execution continues. Thus, C# gives you the choice of handling or ignoring overflow.

In addition to the arithmetic operators, integral-type to integral-type casts can cause overflow (for example, casting a long to an int) and are subject to checked or unchecked execution. Also note that bitwise operators and shift operators never cause overflow.

See Also

7.2.1 Operator precedence and associativity | Overloadable Operators | C# Keywords | C# Programmer's Reference

Overloadable Operators

C# allows user-defined types to overload operators by defining static member functions using the operator keyword. Not all operators can be overloaded, however, and others have restrictions, as listed in this table:

Operators	Overloadability
+, -, !, ~, ++, --, true, false	These unary operators can be overloaded.
+, -, *, /, %, &, |, ^, <<, >>	These binary operators can be overloaded.
==, !=, <, >, <=, >=	The comparison operators can be overloaded (but see note below).
&&, ||	The conditional logical operators cannot be overloaded, but they are evaluated using **&** and **|**, which can be overloaded; see 7.11.2 User-defined conditional logical operators.
[]	The array indexing operator cannot be overloaded, but you can define indexers.
()	The cast operator cannot be overloaded, but you can define new conversion operators (see explicit and implicit).
+=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=	Assignment operators cannot be overloaded, but **+=**, for example, is evaluated using **+**, which can be overloaded.
=, ., ?:, ->, new, is, sizeof, typeof	These operators cannot be overloaded.

Note The comparison operators, if overloaded, must be overloaded in pairs; that is, if **==** is overloaded, **!=** must also be overloaded. The reverse is also true, and similar for **<** and **>**, and for **<=** and **>=**.

See Also

C# Operators | Operator Overloading Tutorial

[] Operator

Square brackets ([]) are used for arrays, indexers, and attributes. They can also be used with pointers.

```
type []
array [ indexexpr ]
```

Where:

type
A type.
array
An array.
indexexpr
An index expression.

Remarks

An array type is a type followed by []:

```
int[] fib; // fib is of type int[], "array of int"
fib = new int[100]; // create a 100-element int array
```

To access an element of an array, the index of the desired element is enclosed in brackets:

```
fib[0] = fib[1] = 1;
for( int i=2; i<100; ++i ) fib[i] = fib[i-1] + fib[i-2];
```

An exception is thrown if an array index is out of range.

The array indexing operator cannot be overloaded; however, types can define indexers, properties that take one or more parameters. Indexer parameters are enclosed in square brackets, just like array indices, but indexer parameters can be declared to be of any type (unlike array indices, which must be integral).

For example, the .NET Framework defines a Hashtable type that associates keys and values of arbitrary type:

```
Collections.Hashtable h =  new Collections.Hashtable();
h["a"] = 123; // note: using a string as the index
```

Square brackets are also used to specify attributes:

```
[attribute(AllowMultiple=true)]
public class Attr {
}
```

You can use square brackets to index off a pointer (see A.2 Pointer types):

```
unsafe fixed ( int* p = fib )   // p points to fib from earlier example
{
   p[0] = p[1] = 1;
   for( int i=2; i<100; ++i ) p[i] = p[i-1] + p[i-2];
}
```

No bounds checking is performed.

See Also

C# Operators | Arrays | Indexers | unsafe | fixed

() Operator

In addition to being used to specify the order of operations in an expression, parentheses are used to specify casts (type conversions):

```
( type ) expr
```

Where:

type
The name of the type to which *expr* is to be converted.

expr
An expression.

Remarks

A cast explicitly invokes the conversion operator from *expr*'s type to *type*; the cast will fail if no such conversion operator is defined. To define a conversion operator, see explicit and implicit.

Example

The following program casts a **double** to an **int**. The program won't compile without the cast.

```
// cs_operator_parentheses.cs
using System;
class Test
{
   public static void Main()
   {
      double x = 1234.7;
      int a;
      a = (int)x; // cast double to int
      Console.WriteLine(a);
   }
}
```

Output

```
1234
```

See Also

C# Operators | 7.6.6 Cast expressions | explicit

. Operator

The dot operator is used for member access.

```
name1 . name2
```

Where:

name1
A name.
name2
A name.

Remarks

For example, consider the following class:

```
class Simple
{
   public int a;
   public void b()
   {
   }
}
Simple s = new Simple();
```

The variable `s` has two members, `a` and `b`; to access them, use the dot operator:

```
s.a = 6;   // assign to field a;
s.b();     // invoke member function b;
```

The dot is also used to form qualified names, names that specify the namespace or interface (for example) to which they belong.

```
System.Console.WriteLine("hello"); // class Console in namespace System
```

The using directive makes some name qualification optional:

```
using System;
...
System.Console.WriteLine("hello");
Console.WriteLine("hello");   // same thing
```

But when an identifier is ambiguous, it must be qualified:

```
using System;
using OtherSystem; // a namespace containing another Console class
...
System.Console.WriteLine( "hello" ); // must qualify Console
```

See Also

C# Operators | 7.5.4 Member access

+ Operator

The + operator can function as either a unary or a binary operator.

```
+ expr
expr1 + expr2
```

Where:

expr
An expression.
expr1
An expression.
expr2
An expression.

Remarks

Unary + operators are predefined for all numeric types. The result of a unary + operation on a numeric type is simply the value of the operand.

Binary + operators are predefined for numeric and string types. For numeric types, + computes the sum of its two operands. When one or both operands are of type string, + concatenates the string representations of the operands.

Delegate types also provide a binary + operator, which performs delegate concatenation.

User-defined types can overload the unary + and binary + operators (see operator).

Example

```
// cs_operator_plus.cs
using System;
class Test
{
   public static void Main()
   {
      Console.WriteLine(+5);        // unary plus
      Console.WriteLine(5 + 5);     // addition
      Console.WriteLine(5 + .5);    // addition
      Console.WriteLine("5" + "5"); // string concatenation
      Console.WriteLine(5.0 + "5"); // string concatenation
      // note automatic conversion from double to string
   }
}
```

Output

```
5
10
5.5
55
55
```

See Also

C# Operators | 7.6.1 Unary plus operator | 7.7.4 Addition operator

- Operator

The - operator can function as either a unary or a binary operator.

```
- expr
expr1 - expr2
```

Where:

expr
An expression.
expr1
An expression.
expr2
An expression.

Remarks

Unary - operators are predefined for all numeric types. The result of a unary - operation on a numeric type is the negation of the operand.

Binary - operators are predefined for all numeric and enumeration types to subtract the second operand from the first.

Delegate types also provide a binary - operator, which performs delegate removal.

User-defined types can overload the unary - and binary - operators (see operator).

Example

```
// cs_operator_minus.cs
using System;
class Test
{
   public static void Main()
   {
      int a = 5;
      Console.WriteLine(-a);
      Console.WriteLine(a - 1);
      Console.WriteLine(a - .5);
   }
}
```

Output

```
-5
4
4.5
```

See Also

C# Operators | 7.6.2 Unary minus operator | 7.7.5 Subtraction operator

* Operator

The multiplication operator (*) computes the product of its operands. All numeric types have predefined multiplication operators.

```
expr1 * expr2
```

Where:

expr1
An expression.
expr2
An expression.

Remarks

The * operator is also used to declare pointer types and to dereference pointers (see A.2 Pointer types).

User-defined types can overload the * operator (see operator).

Example

```
// cs_operator_mult.cs
using System;
class Test
{
   public static void Main()
   {
      Console.WriteLine(5 * 2);
      Console.WriteLine(-.5 * .2);
      Console.WriteLine(-.5m * .2m); // decimal type
   }
}
```

Output

```
10
-0.1
-0.10
```

See Also

C# Operators | 7.7.1 Multiplication operator

/ Operator

The division operator (/) divides its first operand by its second. All numeric types have predefined division operators.

```
expr1 / expr2
```

Where:

expr1
An expression.
expr2
An expression.

Remarks

User-defined types can overload the / operator (see operator).

Example

```
// cs_operator_division.cs
using System;
class Test
{
   public static void Main()
   {
      Console.WriteLine(-5/2);
      Console.WriteLine(-5.0/2);
   }
}
```

Output

```
-2
-2.5
```

See Also

C# Operators | 7.7.2 Division operator

% Operator

The modulus operator (%) computes the remainder after dividing its first operand by its second. All numeric types have predefined modulus operators.

```
expr1 % expr2
```

Where:

expr1
An expression.
expr2
An expression.

Remarks

User-defined types can overload the % operator (see operator).

Example

```
// cs_operator_modulus.cs
using System;
class Test
{
   public static void Main()
   {
      Console.WriteLine(5 % 2);       // int
      Console.WriteLine(-5 % 2);      // int
      Console.WriteLine(5.0 % 2.2);   // double
      Console.WriteLine(5.0m % 2.2m); // decimal
      Console.WriteLine(-5.2 % 2.0);  // double
   }
}
```

Output

```
1
-1
0.6
0.6
-1.2
```

Note the round-off errors associated with the double type.

See Also

C# Operators | 7.7.3 Remainder operator

& Operator

The & operator can function as either a unary or a binary operator.

```
& expr
expr1 & expr2
```

Where:

expr
An expression.
expr1
An expression.
expr2
An expression.

Remarks

The unary & operator returns the address of its operand (requires unsafe context).

Binary & operators are predefined for the integral types and **bool**. For integral types, & computes the bitwise AND of its operands. For **bool** operands, & computes the logical AND of its operands; that is, the result is **true** if and only if both its operands are **true**.

User-defined types can overload the binary & operator (see operator).

Example

```
// cs_operator_ampersand.cs
using System;
class Test
{
   public static void Main()
   {
      Console.WriteLine(true & false); // logical and
      Console.WriteLine(true & true);  // logical and
      Console.WriteLine("0x{0:x}", 0xf8 & 0x3f);  // bitwise and
   }
}
```

Output

```
False
True
0x38
```

See Also

C# Operators | 7.10 Logical operators | A.2 Pointer types

| Operator

Binary | operators are predefined for the integral types and **bool**. For integral types, | computes the bitwise OR of its operands. For **bool** operands, | computes the logical OR of its operands; that is, the result is **false** if and only if both its operands are **false**.

```
expr1 | expr2
```

Where:

expr1
An expression.
expr2
An expression.

Remarks

User-defined types can overload the | operator (see operator).

Example

```
// cs_operator_OR.cs
using System;
class Test
{
   public static void Main()
   {
      Console.WriteLine(true | false);  // logical or
      Console.WriteLine(false | false); // logical or
      Console.WriteLine("0x{0:x}", 0xf8 | 0x3f);   // bitwise or
   }
}
```

Output

```
True
False
0xff
```

See Also

C# Operators | 7.10 Logical operators

^ Operator

Binary ^ operators are predefined for the integral types and **bool**. For integral types, ^ computes the bitwise exclusive-OR of its operands. For **bool** operands, ^ computes the logical exclusive-or of its operands; that is, the result is **true** if and only if exactly one of its operands is **true**.

```
expr1 ^ expr2
```

Where:

expr1
An expression.
expr2
An expression.

Remarks

User-defined types can overload the ^ operator (see operator).

Example

```
// cs_operator_bitwise_OR.cs
using System;
class Test
{
   public static void Main()
   {
      Console.WriteLine(true ^ false);  // logical exclusive-or
      Console.WriteLine(false ^ false); // logical exclusive-or
      Console.WriteLine("0x{0:x}", 0xf8 ^ 0x3f);   // bitwise exclusive-or
   }
}
```

Output

```
True
False
0xc7
```

See Also

C# Operators | 7.10.1 Integer logical operators

! Operator

The logical negation operator (!) is a unary operator that negates its operand. It is defined for **bool** and returns **true** if and only if its operand is **false**.

```
! expr
```

Where:

expr
An expression.

Remarks

User-defined types can overload the ! operator (see operator).

Example

```
// cs_operator_negation.cs
using System;
class Test
{
   public static void Main()
{
      Console.WriteLine(!true);
      Console.WriteLine(!false);
   }
}
```

Output

```
False
True
```

See Also

C# Operators | 7.6.3 Logical negation operator

~ Operator

The ~ operator performs a bitwise complement operation on its operand. Bitwise complement operators are predefined for **int**, **uint**, **long**, and **ulong**.

```
~ expr
```

Where:

expr
An expression.

Remarks

User-defined types can overload the ~ operator (see operator).

Example

```
// cs_operator_bitwise_compl.cs
using System;
class Test
{
   public static void Main()
   {
      Console.WriteLine("!0x{0:x8} = 0x{1:x8}", 8, ~8);
      Console.WriteLine("!0x{0:x8} = 0x{1:x8}", -8, ~-8);
   }
}
```

Output

```
!0x00000008 = 0xfffffff7
!0xfffffff8 = 0x00000007
```

See Also

C# Operators | 7.6.4 Bitwise complement operator

= Operator

The assignment operator (=) stores the value of its right-hand operand in the storage location, property, or indexer denoted by its left-hand operand and returns the value as its result. The operands must be of the same type (or the right-hand operand must be implicitly convertible to the type of the left-hand operand).

```
lhs = expr
```

Where:

lhs
A storage location, property, or indexer.
expr
An expression.

Remarks

The assignment operator cannot be overloaded.

Example

```
// cs_operator_assignment.cs
using System;
class Test
{
   public static void Main()
   {
      double x;
      int i;
      i = 5; // int to int assignment
      x = i; // implicit conversion from int to double
      i = (int)x; // needs cast
      Console.WriteLine("i is {0}, x is {1}", i, x);
      object obj = i;
      Console.WriteLine("boxed value = {0}, type is {1}",
         obj, obj.GetType());
      i = (int)obj;
      Console.WriteLine("unboxed: {0}", i);
   }
}
```

Output

```
i is 5, x is 5
boxed value = 5, type is System.Int32
unboxed: 5
```

See Also

C# Operators | 7.13 Assignment operators

< Operator

All numeric and enumeration types define a "less than" relational operator (<) that returns **true** if the first operand is less than the second, **false** otherwise.

```
expr1 < expr2
```

Where:

expr1
An expression.
expr2
An expression.

Remarks

User-defined types can overload the < operator (see operator). If < is overloaded, > must also be overloaded.

Example

```
// cs_operator_less_than.cs
using System;
class Test
{
   public static void Main()
   {
      Console.WriteLine(1 < 1.1);
      Console.WriteLine(1.1 < 1.1);
   }
}
```

Output

```
True
False
```

See Also

C# Operators | 7.9 Relational and type testing operators

> Operator

All numeric and enumeration types define a "greater than" relational operator (>) that returns **true** if the first operand is greater than the second, **false** otherwise.

```
expr1 > expr2
```

Where:

expr1
An expression.
expr2
An expression.

Remarks

User-defined types can overload the > operator (see operator). If > is overloaded, < must also be overloaded.

Example

```
// cs_operator_greater_than.cs
using System;
class Test
{
   public static void Main()
   {
      Console.WriteLine(1.1 > 1);
      Console.WriteLine(1.1 > 1.1);
   }
}
```

Output

```
True
False
```

See Also

C# Operators | 7.9 Relational and type testing operators

?: Operator

The conditional operator (?:) returns one of two values depending on the value of a Boolean expression. The conditional operator is used in an expression of the following form:

```
cond-expr ? expr1 : expr2
```

Where:

cond-expr
An expression of type **bool**.
expr1
An expression.
expr2
An expression.

Remarks

If *cond-expr* is **true**, *expr1* is evaluated and becomes the result; if *cond-expr* is **false**, *expr2* is evaluated and becomes the result. Only one of *expr1* and *expr2* is ever evaluated.

Calculations that might otherwise require an if-else construction can be expressed more concisely and elegantly with the conditional operator. For example, to avoid a division by zero in the calculation of the **sinc** function you could write either

```
if(x != 0.0) s = Math.Sin(x)/x; else s = 1.0;
```

or, using the conditional operator,

```
s = x != 0.0 ? Math.Sin(x)/x : 1.0;
```

The conditional operator is right-associative, so an expression of the form

```
a ? b : c ? d : e
```

is evaluated as

```
a ? b : (c ? d : e)
```

not

```
(a ? b : c) ? d : e
```

The conditional operator cannot be overloaded.

Example

```
// cs_operator_conditional.cs
using System;
class Test
{
   public static double sinc(double x)
   {
      return x != 0.0 ? Math.Sin(x)/x : 1.0;
   }

   public static void Main()
   {
      Console.WriteLine(sinc(0.2));
      Console.WriteLine(sinc(0.1));
```

```
        Console.WriteLine(sinc(0.0));
    }
}
```

Output

```
0.993346653975306
0.998334166468282
1
```

See Also

C# Operators | 7.12 Conditional operator | if-else

++ Operator

The increment operator (++) increments its operand by 1. The increment operator can appear before or after its operand:

```
++ var
var ++
```

Where:

var
An expression that denotes a storage location or a property or an indexer.

Remarks

The first form is a prefix increment operation. The result of the operation is the value of the operand after it has been incremented.

The second form is a postfix increment operation. The result of the operation is the value of the operand *before* it has been incremented.

Numeric and enumeration types have predefined increment operators. User-defined types can overload the ++ operator (see operator).

Example

```
// cs_operator_increment.cs
using System;
class Test
{
   public static void Main()
   {
      double x;
      x = 1.5;
      Console.WriteLine(++x);
      x = 1.5;
      Console.WriteLine(x++);
      Console.WriteLine(x);
   }
}
```

Output

```
2.5
1.5
2.5
```

See Also

C# Operators | 7.5.9 Postfix increment and decrement operators | 7.6.5 Prefix increment and decrement operators

-- Operator

The decrement operator (--) decrements its operand by 1. The decrement operator can appear before or after its operand:

```
-- var
var --
```

Where:

var
An expression that denotes a storage location or a property or an indexer.

Remarks

The first form is a prefix decrement operation. The result of the operation is the value of the operand *after* it has been decremented.

The second form is a postfix decrement operation. The result of the operation is the value of the operand *before* it has been decremented.

Numeric and enumeration types have predefined decrement operators. User-defined types can overload the -- operator (see operator).

Example

```
// cs_operator_decrement.cs
using System;
class Test
{
   public static void Main()
   {
      double x;
      x = 1.5;
      Console.WriteLine(--x);
      x = 1.5;
      Console.WriteLine(x--);
      Console.WriteLine(x);
   }
}
```

Output

```
0.5
1.5
0.5
```

See Also

C# Operators | 7.5.9 Postfix increment and decrement operators | 7.6.5 Prefix increment and decrement operators

&& Operator

The conditional-AND operator (&&) performs a logical-AND of its bool operands, but only evaluates its second operand if necessary.

```
expr1 && expr2
```

Where:

expr1
An expression.
expr2
An expression.

Remarks

The operation

```
x && y
```

corresponds to the operation

```
x & y
```

except that if `x` is `false`, `y` is not evaluated (because the result of the AND operation is `false` no matter what the value of `y` may be). This is known as "short-circuit" evaluation.

The conditional-AND operator cannot be overloaded, but overloads of the regular logical operators and operators true and false are, with certain restrictions, also considered overloads of the conditional logical operators (see 7.11.2 User-defined conditional logical operators).

Example

In the following example, observe that the expression using && evaluates only the first operand.

```
// cs_operator_logical_and.cs
using System;
class Test
{
   static bool fn1()
   {
      Console.WriteLine("fn1 called");
      return false;
   }

   static bool fn2()
   {
      Console.WriteLine("fn2 called");
      return true;
   }

   public static void Main()
   {
      Console.WriteLine("regular AND:");
      Console.WriteLine("result is {0}", fn1() & fn2());
      Console.WriteLine("short-circuit AND:");
      Console.WriteLine("result is {0}", fn1() && fn2());
   }
}
```

Output

```
regular AND:
fn1 called
fn2 called
result is False
short-circuit AND:
fn1 called
result is False
```

See Also

C# Operators | 7.11 Conditional logical operators

|| Operator

The conditional-OR operator (||) performs a logical-OR of its bool operands, but only evaluates its second operand if necessary.

```
expr1 || expr2
```

Where:

expr1
An expression.
expr2
An expression.

Remarks

The operation

```
x || y
```

corresponds to the operation

```
x | y
```

except that if `x` is `true`, `y` is not evaluated (because the result of the OR operation is `true` no matter what the value of `y` might be). This is known as "short-circuit" evaluation.

The conditional-OR operator cannot be overloaded, but overloads of the regular logical operators and operators true and false are, with certain restrictions, also considered overloads of the conditional logical operators (see 7.11.2 User-defined conditional logical operators).

Example

In the following example, observe that the expression using || evaluates only the first operand.

```
// cs_operator_short_circuit_OR.cs
using System;
class Test
{
   static bool fn1()
   {
      Console.WriteLine("fn1 called");
      return true;
   }

   static bool fn2()
   {
      Console.WriteLine("fn2 called");
      return false;
   }

   public static void Main()
   {
      Console.WriteLine("regular OR:");
      Console.WriteLine("result is {0}", fn1() | fn2());
      Console.WriteLine("short-circuit OR:");
      Console.WriteLine("result is {0}", fn1() || fn2());
   }
}
```

Output

```
regular OR:
fn1 called
fn2 called
result is True
short-circuit OR:
fn1 called
result is True
```

See Also

C# Operators | 7.11 Conditional logical operators

<< Operator

The left-shift operator (<<) shifts its first operand left by the number of bits specified by its second operand.

```
expr << count
```

Where:

expr
An expression of type **int**, **uint**, **long**, or **ulong**; the value to be shifted.
count
An expression of type **int**; the shift count.

Remarks

If *expr* is an **int** or **uint** (32-bit quantity), the shift count is given by the low-order five bits of *count* (*count* & 0x1f).

If *expr* is a **long** or **ulong** (64-bit quantity), the shift count is given by the low-order six bits of *count* (*count* & 0x3f).

The high-order bits of *expr* are discarded and the low-order empty bits are zero-filled. Shift operations never cause overflows.

User-defined types can overload the << operator (see operator); the type of the first operand must be the user-defined type, and the type of the second operand must be **int**.

Example

```
// cs_operator_left_shift.cs
using System;
class Test
{
   public static void Main()
   {
      int i = 1;
      long lg = 1;
      Console.WriteLine("0x{0:x}", i << 1);
      Console.WriteLine("0x{0:x}", i << 33);
      Console.WriteLine("0x{0:x}", lg << 33);
   }
}
```

Output

```
0x2
0x2
0x200000000
```

Note that `i<<1` and `i<<33` give the same result, because 1 and 33 have the same low-order five bits.

See Also

C# Operators | 7.8 Shift operators | >> Operator

>> Operator

The right-shift operator (>>) shifts its first operand right by the number of bits specified by its second operand.

```
expr >> count
```

Where:

expr
An expression of type **int**, **uint**, **long**, or **ulong**; the value to be shifted.
count
An expression of type **int**; the shift count.

Remarks

If *expr* is an **int** or **uint** (32-bit quantity), the shift count is given by the low-order five bits of *count* (*count* & 0x1f).

If *expr* is a **long** or **ulong** (64-bit quantity), the shift count is given by the low-order six bits of *count* (*count* & 0x3f).

If *expr* is an **int** or **long**, the right-shift is an arithmetic shift (high-order empty bits are set to the sign bit). If *expr* is of type **uint** or **ulong**, the right-shift is a logical shift (high-order bits are zero-filled).

User-defined types can overload the >> operator (see operator); the type of the first operand must be the user-defined type, and the type of the second operand must be **int**.

Example

```
// cs_operator_right_shift.cs
using System;
class Test
{
   public static void Main()
   {
      int i = -1000;
      Console.WriteLine(i >> 3);
   }
}
```

Output

```
-125
```

See Also

C# Operators | 7.8 Shift operators | << Operator

== Operator

For predefined value types, the equality operator (==) returns true if the values of its operands are equal, false otherwise. For reference types other than **string**, == returns true if its two operands refer to the same object. For the **string** type, == compares the values of the strings.

```
expr1 == expr2
```

Where:

expr1
An expression.
expr2
An expression.

Remarks

User-defined value types can overload the == operator (see operator). So can user-defined reference types, although by default == behaves as described above for both predefined and user-defined reference types. If == is overloaded, != must also be overloaded.

Example

```
// cs_operator_equality.cs
using System;
class Test
{
   public static void Main()
   {
      // Numeric equality: True
      Console.WriteLine((2 + 2) == 4);

      // Reference equality: different objects, same boxed value: False
      object s = 1;
      object t = 1;
      Console.WriteLine(s == t);

      // Define some string
      string a = "hello";
      string b = String.Copy(a);
      string c = "hello";

      // compare string values for a constant and an instance: True
      Console.WriteLine(a == b);

      // compare string references;
      // a is a constant but b is an instance: False
      Console.WriteLine((object)a == (object)b);

      // compare string references, both constants have the same value,
      // so string interning points to same reference: True
      Console.WriteLine((object)a == (object)c);
   }
}
```

Output

```
True
False
True
False
True
```

See Also

C# Operators | 7.9 Relational and type testing operators | != Operator

!= Operator

The inequality operator (!=) returns false if its operands are equal, true otherwise. Inequality operators are predefined for all types, including string and object. User-defined types can overload the != operator.

```
expr1 != expr2
```

Where:

expr1
An expression.
expr2
An expression.

Remarks

For predefined value types, the inequality operator (!=) returns true if the values of its operands are different, false otherwise. For reference types other than **string**, != returns true if its two operands refer to different objects. For the **string** type, != compares the values of the strings.

User-defined value types can overload the != operator (see operator). So can user-defined reference types, although by default != behaves as described above for both predefined and user-defined reference types. If != is overloaded, == must also be overloaded.

Example

```
// cs_operator_inequality.cs
using System;
class Test
{
   public static void Main()
   {
      // Numeric inequality:
      Console.WriteLine((2 + 2) != 4);

      // Reference equality: two objects, same boxed value
      object s = 1;
      object t = 1;
      Console.WriteLine(s != t);

      // String equality: same string value, same string objects
      string a = "hello";
      string b = "hello";

      // compare string values
      Console.WriteLine(a != b);

      // compare string references
      Console.WriteLine((object)a != (object)b);
   }
}
```

Output

```
False
True
False
False
```

See Also

<= Operator

All numeric and enumeration types define a "less than or equal" relational operator (<=) that returns **true** if the first operand is less than or equal to the second, **false** otherwise.

```
expr1 <= expr2
```

Where:

expr1
An expression.
expr2
An expression.

Remarks

User-defined types can overload the <= operator (see operator). If <= is overloaded, >= must also be overloaded.

Example

```
// cs_operator_less_than_or_equal.cs
using System;
class Test
{
   public static void Main()
   {
      Console.WriteLine(1 <= 1.1);
      Console.WriteLine(1.1 <= 1.1);
   }
}
```

Output

```
True
True
```

See Also

C# Operators | 7.9 Relational and type testing operators

>= Operator

All numeric and enumeration types define a "greater than or equal" relational operator (>=) that returns **true** if the first operand is greater than or equal to the second, **false** otherwise.

```
expr1 >= expr2
```

Where:

expr1
An expression.
expr2
An expression.

Remarks

User-defined types can overload the >= operator (see operator). If >= is overloaded, <= must also be overloaded.

Example

```
// cs_operator_greater_than_or_equal.cs
using System;
class Test
{
   public static void Main()
   {
      Console.WriteLine(1.1 >= 1);
      Console.WriteLine(1.1 >= 1.1);
   }
}
```

Output

```
True
True
```

See Also

C# Operators | 7.9 Relational and type testing operators

+= Operator

The addition assignment operator.

```
lhs += expr
```

Where:

lhs
A storage location, property, or indexer.
expr
An expression.

Remarks

An expression using the += assignment operator, such as

```
x += y
```

is equivalent to

```
x = x + y
```

except that `x` is only evaluated once. The meaning of the + operator is dependent on the types of `x` and `y` (addition for numeric operands, concatenation for string operands, and so forth).

The += operator cannot be overloaded directly, but user-defined types can overload the + operator (see operator).

Example

```
// cs_operator_addition_assignment.cs
using System;
class Test
{
   public static void Main()
   {
      int a = 5;
      a += 6;
      Console.WriteLine(a);
      string s = "Micro";
      s += "soft";
      Console.WriteLine(s);
   }
}
```

Output

```
11
Microsoft
```

See Also

C# Operators | 7.13 Assignment operators

-= Operator

The subtraction assignment operator.

```
lhs -= expr
```

Where:

lhs
A storage location, property, or indexer.
expr
An expression.

Remarks

An expression using the -= assignment operator, such as

```
x -= y
```

is equivalent to

```
x = x - y
```

except that `x` is only evaluated once. The meaning of the - operator is dependent on the types of `x` and `y` (subtraction for numeric operands, delegate removal for delegate operands, and so forth).

The -= operator cannot be overloaded directly, but user-defined types can overload the - operator (see operator).

Example

```
// cs_operator_subtraction_assignment.cs
using System;
class Test
{
   public static void Main()
   {
      int a = 5;
      a -= 6;
      Console.WriteLine(a);
   }
}
```

Output

```
-1
```

See Also

C# Operators | 7.13 Assignment operators

*= Operator

The multiplication assignment operator.

```
lhs *= expr
```

Where:

lhs
A storage location, property, or indexer.

expr
An expression.

Remarks

An expression using the *= assignment operator, such as

```
x *= y
```

is equivalent to

```
x = x * y
```

except that `x` is only evaluated once. The * operator is predefined for numeric types to perform multiplication.

The *= operator cannot be overloaded directly, but user-defined types can overload the * operator (see operator).

Example

```
// cs_operator_multiplication_assignment.cs
using System;
class Test
{
   public static void Main()
   {
      int a = 5;
      a *= 6;
      Console.WriteLine(a);
   }
}
```

Output

```
30
```

See Also

C# Operators | 7.13 Assignment operators

/= Operator

The division assignment operator.

```
lhs /= expr
```

Where:

lhs
A storage location, property, or indexer.
expr
An expression.

Remarks

An expression using the /= assignment operator, such as

```
x /= y
```

is equivalent to

```
x = x / y
```

except that `x` is only evaluated once. The / operator is predefined for numeric types to perform division.

The /= operator cannot be overloaded directly, but user-defined types can overload the / operator (see operator).

Example

```
// cs_operator_division_assignment.cs
using System;
class Test
{
   public static void Main()
   {
      int a = 5;
      a /= 6;
      Console.WriteLine(a);
      double b = 5;
      b /= 6;
      Console.WriteLine(b);
   }
}
```

Output

```
0
0.833333333333333
```

See Also

C# Operators | 7.13 Assignment operators

%= Operator

The modulus assignment operator.

```
lhs %= expr
```

Where:

lhs
A storage location, property, or indexer.

expr
An expression.

Remarks

An expression using the %= assignment operator, such as

```
x %= y
```

is equivalent to

```
x = x % y
```

except that `x` is only evaluated once. The % operator is predefined for numeric types to compute the remainder after division.

The %= operator cannot be overloaded directly, but user-defined types can overload the % operator (see operator).

Example

```
// cs_operator_modulus_assignment.cs
using System;
class Test
{
   public static void Main()
   {
      int a = 5;
      a %= 3;
      Console.WriteLine(a);
   }
}
```

Output

```
2
```

See Also

C# Operators | 7.13 Assignment operators

|= Operator

The OR assignment operator.

```
lhs |= expr
```

Where:

lhs
A storage location, property, or indexer.
expr
An expression.

Remarks

An expression using the |= assignment operator, such as

```
x |= y
```

is equivalent to

```
x = x | y
```

except that `x` is only evaluated once. The | operator performs a bitwise OR operation on integral operands and logical OR on bool operands.

The |= operator cannot be overloaded directly, but user-defined types can overload the | operator (see operator).

Example

```
// cs_operator_or_assignment.cs
using System;
class Test
{
   public static void Main()
   {
      int a = 0x0c;
      a |= 0x06;
      Console.WriteLine("0x{0:x8}", a);
      bool b = true;
      b |= false;
      Console.WriteLine(b);
   }
}
```

Output

```
0x0000000e
True
```

See Also

C# Operators | 7.13 Assignment operators

^= Operator

The exclusive-OR assignment operator.

```
lhs ^= expr
```

Where:

lhs
A storage location, property, or indexer.

expr
An expression.

Remarks

An expression of the form

```
x ^= y
```

is evaluated as

```
x = x ^ y
```

except that `x` is only evaluated once. The ^ operator performs a bitwise exclusive-OR operation on integral operands and logical exclusive-OR on bool operands.

The ^= operator cannot be overloaded directly, but user-defined types can overload the ^ operator (see operator).

Example

```
// cs_operator_xor_assignment.cs
using System;
class Test
{
   public static void Main()
   {
      int a = 0x0c;
      a ^= 0x06;
      Console.WriteLine("0x{0:x8}", a);
      bool b = true;
      b ^= false;
      Console.WriteLine(b);
   }
}
```

Output

```
0x0000000a
True
```

See Also

C# Operators | 7.13 Assignment operators

<<= Operator

The left-shift assignment operator.

```
lhs <<= expr
```

Where:

lhs
A storage location, property, or indexer.

expr
An expression.

Remarks

An expression of the form

```
x <<= y
```

is evaluated as

```
x = x << y
```

except that x is only evaluated once. The << operator shifts `x` left by an amount specified by `y`.

The <<= operator cannot be overloaded directly, but user-defined types can overload the << operator (see operator).

Example

```
// cs_operator_left_shift_assignment.cs
using System;
class Test
{
   public static void Main()
   {
      int a = 1000;
      a <<= 4;
      Console.WriteLine(a);
   }
}
```

Output

```
16000
```

See Also

C# Operators | 7.13 Assignment operators

>>= Operator

The right-shift assignment operator.

```
lhs >>= expr
```

Where:

lhs
A storage location, property, or indexer.

expr
An expression.

Remarks

An expression of the form

```
x >>= y
```

is evaluated as

```
x = x >> y
```

except that `x` is only evaluated once. The `>>` operator shifts `x` right by an amount specified by `y`.

The `>>=` operator cannot be overloaded directly, but user-defined types can overload the `>>` operator (see operator).

Example

```
// cs_operator_right_shift_assignment.cs
using System;
class Test
{
   public static void Main()
   {
      int a = 1000;
      a >>= 4;
      Console.WriteLine(a);
   }
}
```

Output

```
62
```

See Also

C# Operators | 7.13 Assignment operators

-> Operator

The -> operator combines pointer dereferencing and member access.

```
expr1 -> expr2
```

Where:

expr1
A pointer expression.
expr2
An expression.

Remarks

An expression of the form

```
x->y
```

(where `x` is a pointer of type `T*` and `y` is a member of `T`) is equivalent to

```
(*x).y
```

The -> operator can be used only in unmanaged code.

The -> operator cannot be overloaded.

Example

```
// cs_operator_dereferencing.cs
// compile with: /unsafe
using System;
struct Point
{
   public int x, y;
}

class Test
{
   public unsafe static void Main()
   {
      Point pt = new Point();
      Point* pp = &pt;
      pp->x = 123;
      pp->y = 456;
      Console.WriteLine ( "{0} {1}", pt.x, pt.y );
   }
}
```

Output

```
123 456
```

See Also

C# Operators | A.2 Pointer types

C# Attributes

This section contains an introduction to attributes and descriptions of the following reserved attributes.

AttributeUsage	Describes how a custom attribute class can be used.
Conditional	Marks a conditional method, a method whose execution depends on a specified preprocessing identifier.
Obsolete	Marks a program entity that should not be used.

See Also

C# Programmer's Reference | 17. Attributes | Attributes Tutorial | Introduction to Attributes | B. Interoperability | Platform Invoke Tutorial

Introduction to Attributes

C# provides a mechanism for defining declarative tags, called attributes, which you can place on certain entities in your source code to specify additional information. The information that attributes contain can be retrieved at run time through reflection. You can use predefined attributes or you can define your own custom attributes. This section provides a general introduction to attributes in C#; for more detailed information, refer to 17. Attributes.

- Using Attributes
- Attribute Targets and Global Attributes
- Creating Custom Attributes
- Retrieving Attribute Information

See Also

C# Attributes | 17. Attributes | Attributes Tutorial | Platform Invoke Tutorial

Using Attributes

Attributes can be placed on most any declaration (though a specific attribute might restrict the types of declarations on which it is valid). Syntactically, an attribute is specified by placing the name of the attribute, enclosed in square brackets, in front of the declaration of the entity to which it applies. For example, a class with the attribute **DllImport** is declared like this:

```
[DllImport] public class MyDllimportClass { ... }
```

For more information, see DllImportAttribute Class.

Many attributes have parameters, which can be either positional (unnamed) or named. Any positional parameters must be specified in a certain order and cannot be omitted; named parameters are optional and can be specified in any order. Positional parameters are specified first. For example, these three attributes are equivalent:

```
[DllImport("user32.dll", SetLastError=false, ExactSpelling=false)]
[DllImport("user32.dll", ExactSpelling=false, SetLastError=false)]
[DllImport("user32.dll")]
```

The first parameter, the DLL name, is positional and always comes first; the others are named. In this case, both named parameters default to false, so they can be omitted (refer to the individual attribute's documentation for information on default parameter values).

More than one attribute can be placed on a declaration, either separately or within the same set of brackets:

```
bool AMethod([In][Out]ref double x);
bool AMethod([Out][In]ref double x);
bool AMethod([In,Out]ref double x);
```

Some attributes can be specified more than once for a given entity. An example of such a multiuse attribute is Conditional:

```
[Conditional("DEBUG"), Conditional("TEST1")] void TraceMethod() {...}
```

> **Note** By convention, all attribute names end with the word "Attribute" to distinguish them from other items in the .NET Framework. However, you do not need to specify the attribute suffix when using attributes in code. For example, `[DllImport]` is equivalent to `[DllImportAttribute]`, but **DllImportAttribute** is the attribute's actual name in the .NET Framework.

See Also

Introduction to Attributes | Attribute Targets | Global Attributes | Creating Custom Attributes | Retrieving Attribute Information

Attribute Targets

In certain situations, the target of an attribute (that is, the entity to which the attribute applies) appears to be ambiguous. For example, in the following method declaration, the `SomeAttr` attribute could apply to the method or to the method's return value:

```
[SomeAttr] int Method1( string s )
```

This sort of situation arises frequently when marshaling. To resolve the ambiguity, C# has a set of default targets for each kind of declaration, which can be overridden by explicitly specifying attribute targets.

```
[method: SomeAttr] int Method1( string s ) // applies to method
[return: SomeAttr] int Method1( string s ) // applies to return value
[SomeAttr] int Method1( string s ) // default: applies to method
```

Note that this is independent of the targets on which `SomeAttr` is defined to be valid (see AttributeUsage); that is, even if `SomeAttr` were defined to apply only to return values, the **return** target would still have to be specified. In other words, the compiler will not use **AttributeUsage** information to resolve ambiguous attribute targets.

The syntax for attribute targets is as follows:

```
[target : attribute-list]
```

where:

target
One of the following: assembly, field, event, method, module, param, property, return, type.
attribute-list
A list of applicable attributes.

The table below lists all declarations where attributes are allowed; for each declaration, the possible targets for attributes on the declaration are listed in the second column. Targets in bold are the defaults.

Declaration	Possible targets
assembly	assembly
module	module
class	**type**
struct	**type**
interface	**type**
enum	**type**
delegate	**type**, return
method	**method**, return
parameter	**param**
field	**field**
property — indexer	**property**
property — get accessor	**method**, return
property — set accessor	**method**, param, return
event — field	**event**, field, method
event — property	**event**, property
event — add	**method**, param
event — remove	**method**, param

Note that assembly- and module-level attributes have no default target. For more information, see Global Attributes.

Example

```
// cs_attribute_targets.cs
// compile with: /target:library
using System.Runtime.InteropServices;
```

```
[Guid("12345678-1234-1234-1234-123456789abc"),
InterfaceType(ComInterfaceType.InterfaceIsIUnknown)]
interface IMyInterface
{
   [DispId(17)]                  // set the DISPID of the method
   // the following attribute sets the marshaling on the return type
   [return : MarshalAs(UnmanagedType.Interface)]
   object MyInterfaceMethod();
}
```

See Also

Introduction to Attributes | 17.2 Attribute specification | Global Attributes | Creating Custom Attributes | Retrieving Attribute Information

Global Attributes

Most attributes are attached to specific language elements, such as classes or methods; however, some attributes are global — they apply to an entire assembly or module. Assembly-level attributes are specified as follows:

```
[assembly: attribute-list]
```

Module-level attributes are specified as follows:

```
[module: attribute-list]
```

where:

attribute-list
A list of applicable global attributes.

Global attributes appear in the source code after any top-level using directives and before any type or namespace declarations. Global attributes can appear in multiple source files in a single compilation.

Of the .NET Framework assembly-level attributes, the following let you modify the information in an assembly:

- System.Reflection.AssemblyConfigurationAttribute
- System.Reflection.AssemblyCultureAttribute
- System.Reflection.AssemblyDescriptionAttribute
- System.Reflection.AssemblyKeyFileAttribute
- System.Reflection.AssemblyKeyNameAttribute
- System.Reflection.AssemblyDelaySignAttribute
- System.Reflection.AssemblyTitleAttribute
- System.Reflection.AssemblyVersionAttribute

Use the file created by Strong Name Utility (SN.exe) as the parameter to **AssemblyKeyFileAttribute**, **AssemblyKeyNameAttribute**, or **AssemblyDelaySignAttribute** to sign (or give a strong name to) your assembly.

See the file AssemblyInfo.cs, which is added to all C# projects created in Visual Studio, to see those attributes in use.

Use /target:module to create an output file that does not contain an Assembly.

> **Note** Assembly-level attributes are ignored if you are not creating an assembly.

Example

The CLSCompliant attribute indicates whether all types defined in the assembly comply with the .NET Framework common language specification.

```
[assembly: System.CLSCompliant(true)]
```

See Also

Introduction to Attributes | Using Attributes | Attribute Targets | Creating Custom Attributes | Retrieving Attribute Information

Creating Custom Attributes

You can create your own custom attributes by defining an attribute class, a class that derives directly or indirectly from **System.Attribute** (which makes identifying attribute definitions in metadata fast and easy). Suppose you want to tag classes and structs with the name of the programmer who wrote the class or struct. You might define a custom `Author` attribute class:

```
using System;
[AttributeUsage(AttributeTargets.Class|AttributeTargets.Struct)]
public class Author : Attribute
{
   public Author(string name) { this.name = name; version = 1.0; }
   public double version;
   string name;
}
```

The class name is the attribute's name, `Author`. It is derived from **System.Attribute**, so it is a custom attribute class. The constructor's parameters are the custom attribute's positional parameters (in this case, `name`), and any public read-write fields or properties are named parameters (in this case, `version` is the only named parameter). Note the use of the **AttributeUsage** attribute to make the `Author` attribute valid only on class and struct declarations.

You could use this new attribute as follows:

```
[Author("H. Ackerman", version=1.1)] class SomeClass{...}
```

AttributeUsage has a named parameter, **AllowMultiple**, with which you can make a custom attribute single-use or multiuse.

```
[AttributeUsage(AttributeTargets.Class|AttributeTargets.Struct,
   AllowMultiple=true)] // multiuse attribute
public class Author : Attribute
{
   ...
}
```

See Also

Introduction to Attributes | Using Attributes | Attribute Targets | Global Attributes | Retrieving Attribute Information | AttributeUsage

Retrieving Attribute Information

The fact that you can define custom attributes and place them in your source code would be of little value without some way of retrieving that information and acting on it. Fortunately, C# has a reflection system that allows you to do just that. The key method is **GetCustomAttributes**, which returns an array of objects that are the run-time equivalents of the source code attributes.

An attribute specification such as:

```
[Author("H. Ackerman", version=1.1)] class AClass {...}
```

is conceptually equivalent to this:

```
Author anonymousAuthorObject = new Author("H. Ackerman");
anonymousAuthorObject.version = 1.1;
```

However, the code is not executed until `AClass` is queried for attributes. Calling **GetCustomAttributes** on `AClass` causes an `Author` object to be constructed and initialized as above. If the class has other attributes, other attribute objects are constructed similarly. **GetCustomAttributes** then returns the `Author` object and any other attribute objects in an array. You can then iterate over this array, determine what attributes were applied based on the type of each array element, and extract information from the attribute objects.

Example

Here is a complete example. A custom attribute is defined, applied to several entities, and retrieved via reflection.

```
// cs_attributes_retr.cs
using System;
[AttributeUsage(AttributeTargets.Class|AttributeTargets.Struct,
   AllowMultiple=true)]
public class Author : Attribute
{
   public Author(string name)
   {
      this.name = name; version = 1.0;
   }
   public double version;
   string name;
   public string GetName()
   {
      return name;
   }
}

[Author("H. Ackerman")]
class FirstClass
{
   /*...*/
}

class SecondClass  // no Author attribute
{
   /*...*/
}

[Author("H. Ackerman"), Author("M. Knott", version=1.1)]
class Steerage
{
   /*...*/
}

class AuthorInfo
{
   public static void Main()
```

```
    {
       PrintAuthorInfo(typeof(FirstClass));
       PrintAuthorInfo(typeof(SecondClass));
       PrintAuthorInfo(typeof(Steerage));
    }
    public static void PrintAuthorInfo(Type t)
    {
       Console.WriteLine("Author information for {0}", t);
       Attribute[] attrs = Attribute.GetCustomAttributes(t);
       foreach(Attribute attr in attrs)
       {
          if (attr is Author)
          {
             Author a = (Author)attr;
             Console.WriteLine("   {0}, version {1:f}",
a.GetName(), a.version);
          }
       }
    }
}
```

Output

```
Author information for FirstClass
   H. Ackerman, version 1.00
Author information for SecondClass
Author information for Steerage
   H. Ackerman, version 1.00
   M. Knott, version 1.10
```

See Also

Introduction to Attributes | Using Attributes | Attribute Targets | Global Attributes | Creating Custom Attributes | Discovering Type Information at Runtime

AttributeUsage

Describes how a custom attribute class can be used.

```
[AttributeUsage(
   validon,
   AllowMultiple=allowmultiple,
   Inherited=inherited
)]
```

Parameters

validon
Specifies the language elements on which the attribute can be placed; a combination of AttributeTargets values. Default value is **AttributeTargets.All**.

allowmultiple (optional)
A bool; if **true**, the attribute is multiuse. Default is **false** (single-use).

inherited (optional)
A **bool**; if **true**, the attribute is inherited by derived classes. Default is **false** (not inherited).

Applies To

Class declarations.

Remarks

The **AttributeUsage** attribute is a single-use attribute. **AttributeUsage** is an alias for **System.AttributeUsageAttribute**.

Example

See the example in Retrieving Attribute Information.

See Also

C# Attributes | Attributes Tutorial

Conditional

Marks a conditional method whose execution depends on a specified preprocessing identifier.

```
[Conditional(
   conditionalSymbol
)]
```

Parameters

conditionalSymbol
A string representing the name of a preprocessing identifier.

Applies To

Method declarations.

Remarks

The **Conditional** attribute is a multiuse attribute. **Conditional** is an alias for **System.Diagnostics.ConditionalAttribute**.

Wherever a conditional method is called, the presence or absence of the preprocessing symbol specified by *conditionalSymbol* at the point of the call determines whether the call is included or omitted. If the symbol is defined, the call is included; otherwise, the call is omitted. Conditional methods provide a cleaner, more elegant alternative to enclosing the method call in #if *conditionalSymbol*...#endif preprocessor directives.

A conditional method must be a method in a class or struct declaration and must have a return type of void (for other restrictions, see 17.4.2 The Conditional attribute).

If a method has multiple **Conditional** attributes, a call to the method is included if at least one of the conditional symbols is defined (in other words, the symbols are logically ORed together).

```
[Conditional("A"), Conditional("B")] public static void IfAorB( )
{
   ...
}
```

To achieve the effect of logically ANDing symbols, you can define serial conditional methods:

```
[Conditional("A")] public static void IfAandB( )
{
   AandBPrivate( );
}
[Conditional("B")] static void AandBPrivate( )
{
   /* Code to execute when both A and B are defined... */
}
```

Call `IfAandB`; if both `A` and `B` are defined, `AandBPrivate` will execute.

Example

```
// cs_attribute_conditional.cs
#define DEBUG
using System;
using System.Diagnostics;
public class Trace
{
   [Conditional("DEBUG")] public static void Msg(string msg)
   {
      Console.WriteLine(msg);
```

```
    }
}
class Test
{
    static void A( )
    {
        Trace.Msg("Now in A.");
        B( );
    }
    static void B( )
    {
        Trace.Msg("Now in B.");
    }
    public static void Main( )
    {
        Trace.Msg("Now in Main.");
        A( );
        Console.WriteLine("Done.");
    }
}
```

Output

```
Now in Main.
Now in A.
Now in B.
Done.
```

If you compile the sample with first line omitted (or changed to `#undef DEBUG`), the output will be: `Done`..

See Also

C# Attributes | Conditional Methods Tutorial | C# Preprocessor Directives

Obsolete

Marks a program entity that should not be used.

```
[Obsolete(
   message
)]
[Obsolete(
   message,
   iserror
)]
```

Parameters

message
A string; ideally, a human-readable explanation of why the item is obsolete and what to use instead.
iserror
A bool; if **true**, the compiler should treat the use of the item as an error. Default value is **false** (compiler generates a warning).

Applies To

Any declaration that allows attributes.

Remarks

The **Obsolete** attribute is a single-use attribute. **Obsolete** is an alias for **System.ObsoleteAttribute**.

When an entity marked **Obsolete** is used in a program, the compiler issues either an error or a warning (depending on *iserror*) and prints out *message*.

Example

```
// cs_attribute_obsolete.cs
// CS0619 expected
using System;
public class MyClass
{
   [Obsolete("Don't use OldWay; use NewWay instead", true)]
   static void OldWay( ) { Console.WriteLine("Silly me!"); }
   static void NewWay( ) { Console.WriteLine("D'oh!"); }
   public static void Main( )
   {
      OldWay( );
   }
}
```

See Also

C# Attributes

C# Preprocessor Directives

This section discusses the C# language's preprocessor directives:

- #if
- #else
- #elif
- #endif
- #define
- #undef
- #warning
- #error
- #line
- #region
- #endregion

While the compiler does not have a separate preprocessor, the directives described in this section are processed as if there was one; these directives are used to aid in conditional compilation. Unlike C and C++ directives, you cannot use these directives to create macros.

A preprocessor directive must be the only instruction on a line.

For more information, see 2.5 Pre-processing directives.

See Also

C# Programmer's Reference | Conditional Methods Tutorial

#if

#if lets you begin a conditional directive, testing a symbol or symbols to see if they evaluate to **true**. If they do evaluate to **true**, the compiler evaluates all the code between the **#if** and the next directive.

```
#if symbol [operator symbol]...
```

where:

symbol
The name of the symbol you want to test. You can also use **true** and **false**. *symbol* can be prefaced with the negation operator. For example, **!true** will evaluate to **false**.

operator (optional)
You can use the following operators to evaluate multiple symbols:

== (equality)

!= (inequality)

&& (and)

|| (or)

You can group symbols and operators with parentheses.

Remarks

#if, along with the **#else**, **#elif**, **#endif**, **#define**, and **#undef** directives, lets you include or exclude code based on the condition of one or more symbols. This can be most useful when compiling code for a debug build or when compiling for a specific configuration.

A conditional directive beginning with a **#if** directive must explicitly be terminated with a **#endif** directive.

Example

```
// preprocessor_if.cs
#define DEBUG
#define VC_V7
using System;
public class MyClass
{
   public static void Main()
   {

      #if (DEBUG && !VC_V7)
         Console.WriteLine("DEBUG is defined");
      #elif (!DEBUG && VC_V7)
         Console.WriteLine("VC_V7 is defined");
      #elif (DEBUG && VC_V7)
         Console.WriteLine("DEBUG and VC_V7 are defined");
      #else
         Console.WriteLine("DEBUG and VC_V7 are not defined");
      #endif
   }
}
```

Output

```
DEBUG and VC_V7 are defined
```

See Also

#else

#else lets you create a compound conditional directive, such that, if none of the expressions in the preceding #if or (optional) #elif directives did not evaluate to **true**, the compiler will evaluate all code between **#else** and the subsequent **#endif**.

```
#else
```

Remarks

#endif must be the next preprocessor directive after **#else**.

Example

See #if for an example of how to use **#else**.

See Also

C# Preprocessor Directives

#elif

#elif lets you create a compound conditional directive. The **#elif** expression will be evaluated if neither the preceding #if nor any preceding (optional) #elif directive expressions evaluate to **true**. If a **#elif** expression evaluates to **true**, the compiler evaluates all the code between the **#elif** and the next directive.

```
#elif symbol [operator symbol]...
```

where:

symbol
The name of the symbol you want to test. You can also use **true** and **false**. *symbol* can be prefaced with the negation operator. For example, **!true** will evaluate to **false**.

operator (optional)
You can use the following operators to evaluate multiple symbols:

== (equality)

!= (inequality)

&& (and)

|| (or)

You can group symbols and operators with parentheses.

Remarks

#elif is equivalent to using:

```
#else
#if
```

But using **#elif** is simpler because each **#if** requires a #endif, whereas a **#elif** can be used without a matching **#endif**.

Example

See #if for an example of how to use **#elif**.

See Also

C# Preprocessor Directives

#endif

#endif specifies the end of a conditional directive, which began with the #if directive.

```
#endif
```

Remarks

A conditional directive, beginning with a **#if** directive, must explicitly be terminated with a **#endif** directive.

Example

See #if for an example of how to use **#endif**.

See Also

C# Preprocessor Directives

#define

#define lets you define a symbol, such that, by using the symbol as the expression passed to the #if directive, the expression will evaluate to **true**.

```
#define symbol
```

where:

symbol
The name of the symbol to define.

Remarks

Symbols can be used to specify conditions for compilation. You can test for the symbol with either #if or #elif. You can also use the **conditional** attribute to perform conditional compilation.

You can define a symbol, but you cannot assign a value to a symbol. The **#define** directive must appear in the file before you use any instructions that are not also directives.

You can also define a symbol with the /define compiler option. You can undefine a symbol with #undef.

A symbol that you define with /define or with **#define** does not conflict with a variable of the same name. That is, a variable name should not be passed to a preprocessor directive and a symbol can only be evaluated by a preprocessor directive.

The scope of a symbol created with **#define** is the file in which it was defined.

Example

See #if for an example of how to use **#define**.

See Also

C# Preprocessor Directives | Conditional Methods Tutorial

#undef

#undef lets you undefine a symbol, such that, by using the symbol as the expression in a **#if** directive, the expression will evaluate to **false**.

```
#undef symbol
```

where:

symbol
The name of the symbol you want to undefine.

Remarks

A symbol can be defined either with the #define directive or the /define compiler option. The **#undef** directive must appear in the file before you use any statements that are not also directives.

Example

```
// preprocessor_undef.cs
// compile with: /d:DEBUG
#undef DEBUG
using System;
public class MyClass
{
   public static void Main()
   {
      #if DEBUG
         Console.WriteLine("DEBUG is defined");
      #else
         Console.WriteLine("DEBUG is not defined");
      #endif
   }
}
```

Output

```
DEBUG is not defined
```

See Also

C# Preprocessor Directives

#warning

#warning lets you generate a level one warning from a specific location in your code.

```
#warning text
```

where:

text
The text of the warning that should appear in the compiler's output.

Remarks

A common use of **#warning** is in a conditional directive. It is also possible to generate a user-defined error with #error.

Example

```
// preprocessor_warning.cs
// CS1030 expected
#define DEBUG
public class MyClass
{
   public static void Main()
   {
      #if DEBUG
      #warning DEBUG is defined
      #endif
   }
}
```

See Also

C# Preprocessor Directives

#error

#error lets you generate an error from a specific location in your code.

```
#error text
```

where:

text
The text of the error that should appear in the compiler's output.

Remarks

A common use of **#error** is in a conditional directive. It is also possible to generate a user-defined warning with #warning.

Example

```
// preprocessor_error.cs
// CS1029 expected
#define DEBUG
public class MyClass
{
   public static void Main()
   {
      #if DEBUG
      #error DEBUG is defined
      #endif
   }
}
```

See Also

C# Preprocessor Directives

#line

#line lets you modify the compiler's line number and (optionally) the file name output for errors and warnings.

```
#line [ number ["file_name"] | hidden | default ]
```

where:

number
The number you want to specify for the following line in a source code file.

"*file_name*" (optional)
The file name you want to appear in the compiler output. By default, the actual name of the source code file is used. The file name must be in double quotation marks ("").

hidden
Hides the successive lines from the debugger until another **#line** directive is encountered.

default
Resets the line numbering in a file.

Remarks

The **#line** directive might be used in an automated, intermediate step in the build process. For example, if lines were removed from the original source code file, but you still wanted the compiler to generate output based on the original line numbering in the file, you could remove lines and then simulate the original line numbering with **#line**.

The **#line hidden** directive hides the successive lines from the debugger, such that when the developer steps through the code, any lines between a **#line hidden** and the next **#line** directive (assuming that it is not another **#line hidden** directive) will be stepped over. This option can also be used to allow ASP.NET to differentiate between user-defined and machine-generated code. Although ASP.NET is the primary consumer of this feature, it is likely that more source generators will make use of it.

A **#line hidden** directive does not affect file names or line numbers in error reporting. That is, if an error is encountered in a hidden block, the compiler will report the current file name and line number of the error.

A source code file can have any number of **#line** directives.

Example 1

The following example shows how to report two warnings, associated with line numbers. The `#line 200` directive forced the line number to be 200 (although the default is #7). The other line (#9) follows the usual sequence as a result of the default `#line` directive.

```
// preprocessor_line.cs
public class MyClass2
{
   public static void Main()
   {
      #line 200
      int i;   // CS0168 on line 200
      #line default
      char c;   // CS0168 on line 9
   }
}
```

Example 2

The following example shows how the debugger ignores the hidden lines in the code. When you run the example, it will display three lines of text. However, when you set a break point, as shown in the example, and hit F10 to step through the code, you will notice that the debugger ignores the hidden line. Notice also that even if you set a break point at the hidden line, the debugger will still ignore it.

```
// preprocessor_linehidden.cs

using System;
```

```
class MyClass
{
   public static void Main()
   {
      Console.WriteLine("Normal line #1.");   // Set a break point here.
      #line hidden
      Console.WriteLine("Hidden line.");
      #line default
      Console.WriteLine("Normal line #2.");
   }
}
```

See Also

C# Preprocessor Directives

#region

#region lets you specify a block of code that you can expand or collapse when using the outlining feature of the Visual Studio Code Editor.

```
#region name
```

where:

name
The name you want to give to the region, which will appear in the Visual Studio Code Editor.

Remarks

A **#region** block must be terminated with a #endregion directive.

A **#region** block cannot overlap with a #if block. However, a **#region** block can be nested in a **#if** block, and a **#if** block can be nested in a **#region** block.

Example

```
// preprocessor_region.cs
#region MyClass definition
public class MyClass
{
   public static void Main()
   {
   }
}
#endregion
```

See Also

C# Preprocessor Directives

#endregion

#endregion marks the end of a #region block.

```
#endregion name
```

where:

name
A comment you want to pass to the `#endregion` directive.

Example

See #region for an example of how to use **#endregion**.

See Also

C# Preprocessor Directives

C# Language Features

This section of the *C# Programmer's Reference* provides information about the following language features:

- Arrays
- Constructors and Destructors
- Indexers
- Main
- Properties
- Passing Parameters
- XML Documentation

See Also

C# Programmer's Reference | C# Tutorials | Visual C# Samples

Arrays

An array is a data structure that contains a number of variables called the elements of the array. The array elements are accessed through computed indexes. C# arrays are zero indexed; that is, the array indexes start at zero. All of the array elements must be of the same type, which is called the element type of the array. Array elements can be of any type, including an array type. An array can be a single-dimensional array, or a multidimensional array. Array types are reference types derived from the abstract base type **System.Array**.

The following topics are discussed in this section:

- Single-Dimensional Arrays
- Multidimensional Arrays
- Jagged Arrays
- Passing Arrays Using ref and out

See Also

C# Language Features | Reference Types | 12. Arrays | Arrays Tutorial

Single-Dimensional Arrays

You can declare an array of five integers as in the following example:

```
int[] myArray = new int [5];
```

This array contains the elements from `myArray[0]` to `myArray[4]`. The **new** operator is used to create the array and initialize the array elements to their default values. In this example, all the array elements are initialized to zero.

An array that stores string elements can be declared in the same way. For example:

```
string[] myStringArray = new string[6];
```

Array Initialization

It is possible to initialize an array upon declaration, in which case, the rank specifier is not needed because it is already supplied by the number of elements in the initialization list. For example:

```
int[] myArray = new int[] {1, 3, 5, 7, 9};
```

A string array can be initialized in the same way. The following is a declaration of a string array where each array element is initialized by a name of a day:

```
string[] weekDays = new string[]
               {"Sun","Sat","Mon","Tue","Wed","Thu","Fri"};
```

When you initialize an array upon declaration, it is possible to use the following shortcuts:

```
int[] myArray = {1, 3, 5, 7, 9};
string[] weekDays = {"Sun","Sat","Mon","Tue","Wed","Thu","Fri"};
```

It is possible to declare an array variable without initialization, but you must use the **new** operator when you assign an array to this variable. For example:

```
int[] myArray;
myArray = new int[] {1, 3, 5, 7, 9};   // OK
myArray = {1, 3, 5, 7, 9};   // Error
```

Value Type and Reference Type Arrays

Consider the following array declaration:

```
MyType[] myArray = new MyType[10];
```

The result of this statement depends on whether `MyType` is a value type or a reference type. If it is a value type, the statement results in creating an array of 10 instances of the type `MyType`. If `MyType` is a reference type, the statement creates an array of 10 elements, each of which is initialized to a null reference.

For more information on value types and reference types, see Types.

Passing Arrays as Parameters

You can pass an initialized array to a method. For example:

```
PrintArray(myArray);
```

You can also initialize and pass a new array in one step. For example:

```
PrintArray(new int[] {1, 3, 5, 7, 9});
```

Example

In the following example, a string array is initialized and passed as a parameter to the `PrintArray` method, where its elements are displayed:

```
// cs_sd_arrays.cs
using System;
public class ArrayClass
{
   static void PrintArray(string[] w)
   {
      for (int i = 0 ; i < w.Length ; i++)
         Console.Write(w[i] + "{0}", i < w.Length - 1 ? " " : "");
      Console.WriteLine();
   }

   public static void Main()
   {
      // Declare and initialize an array:
      string[] WeekDays = new string []
         {"Sun","Sat","Mon","Tue","Wed","Thu","Fri"};

      // Pass the array as a parameter:
      PrintArray(WeekDays);
   }
}
```

Output

```
Sun Sat Mon Tue Wed Thu Fri
```

See Also

Arrays | Multidimensional Arrays | Jagged Arrays

Multidimensional Arrays

Arrays can have more than one dimension. For example, the following declaration creates a two-dimensional array of four rows and two columns:

```
int[,] myArray = new int[4,2];
```

Also, the following declaration creates an array of three dimensions, 4, 2, and 3:

```
int[,,] myArray = new int [4,2,3];
```

Array Initialization

You can initialize the array upon declaration as shown in the following example:

```
int[,] myArray = new  int[,] {{1,2}, {3,4}, {5,6}, {7,8}};
```

You can also initialize the array without specifying the rank:

```
int[,] myArray = {{1,2}, {3,4}, {5,6}, {7,8}};
```

If you choose to declare an array variable without initialization, you must use the **new** operator to assign an array to the variable. For example:

```
int[,] myArray;
myArray = new int[,] {{1,2}, {3,4}, {5,6}, {7,8}};   // OK
myArray = {{1,2}, {3,4}, {5,6}, {7,8}};   // Error
```

You can also assign a value to an array element, for example:

```
myArray[2,1] = 25;
```

Passing Arrays as Parameters

You can pass an initialized array to a method. For example:

```
PrintArray(myArray);
```

You can also initialize and pass a new array in one step. For example:

```
PrintArray(new int[,] {{1,2}, {3,4}, {5,6}, {7,8}});
```

Example

In this example, a two-dimensional array is initialized and passed to the `PrintArray` method, where its elements are displayed.

```
// cs_td_arrays.cs
using System;
public class ArrayClass
{
   static void PrintArray(int[,] w)
   {
      // Display the array elements:
      for (int i=0; i < 4; i++)
         for (int j=0; j < 2; j++)
            Console.WriteLine("Element({0},{1})={2}", i, j, w[i,j]);
```

```
    }

    public static void Main()
    {
       // Pass the array as a parameter:
       PrintArray(new int[,] {{1,2}, {3,4}, {5,6}, {7,8}});
    }
}
```

Output

```
Element(0,0)=1
Element(0,1)=2
Element(1,0)=3
Element(1,1)=4
Element(2,0)=5
Element(2,1)=6
Element(3,0)=7
Element(3,1)=8
```

See Also

Arrays | Single-Dimensional Arrays | Jagged Arrays

Jagged Arrays

A jagged array is an array whose elements are arrays. The elements of a jagged array can be of different dimensions and sizes. A jagged array is sometimes called an "array-of-arrays." This topic contains examples of declaring, initializing, and accessing jagged arrays.

The following is a declaration of a single-dimensional array that has three elements, each of which is a single-dimensional array of integers:

```
int[][] myJaggedArray = new int[3][];
```

Before you can use `myJaggedArray`, its elements must be initialized. You can initialize the elements like this example:

```
myJaggedArray[0] = new int[5];
myJaggedArray[1] = new int[4];
myJaggedArray[2] = new int[2];
```

Each of the elements is a single-dimensional array of integers. The first element is an array of 5 integers, the second is an array of 4 integers, and the third is an array of 2 integers.

It is also possible to use initializers to fill the array elements with values, in which case you don't need the array size, for example:

```
myJaggedArray[0] = new int[] {1,3,5,7,9};
myJaggedArray[1] = new int[] {0,2,4,6};
myJaggedArray[2] = new int[] {11,22};
```

You can also initialize the array upon declaration like this:

```
int[][] myJaggedArray = new int [][]
                        {
                           new int[] {1,3,5,7,9},
                           new int[] {0,2,4,6},
                           new int[] {11,22}
                        };
```

You can use the following shortcut (notice that you cannot omit the **new** operator from the elements initialization because there is no default initialization for the elements):

```
int[][] myJaggedArray = {
                           new int[] {1,3,5,7,9},
                           new int[] {0,2,4,6},
                           new int[] {11,22}
                        };
```

You can access individual array elements like these examples:

```
// Assign 33 to the second element of the first array:
myJaggedArray[0][1] = 33;
// Assign 44 to the second element of the third array:
myJaggedArray[2][1] = 44;
```

It is possible to mix jagged and multidimensional arrays. The following is a declaration and initialization of a single-dimensional jagged array that contains two-dimensional array elements of different sizes:

```
int[][,] myJaggedArray = new int [3][,]
                         {
                            new int[,] { {1,3}, {5,7} },
                            new int[,] { {0,2}, {4,6}, {8,10} },
                            new int[,] { {11,22}, {99,88}, {0,9} }
                         };
```

You can access individual elements like this example, which displays the value of the element `[1,0]` of the first array (value `5`):

```
Console.Write("{0}", myJaggedArray[0][1,0]);
```

Example

This example builds an array, `myArray`, whose elements are arrays. Each one of the array elements has a different size.

```
// cs_array_of_arrays.cs
using System;
public class ArrayTest
{
   public static void Main()
   {
      // Declare the array of two elements:
      int[][] myArray = new int[2][];

      // Initialize the elements:
      myArray[0] = new int[5] {1,3,5,7,9};
      myArray[1] = new int[4] {2,4,6,8};

      // Display the array elements:
      for (int i=0; i < myArray.Length; i++)
      {
         Console.Write("Element({0}): ", i);

         for (int j = 0 ; j < myArray[i].Length ; j++)
            Console.Write("{0}{1}", myArray[i][j],
                          j == (myArray[i].Length-1) ? "" : " ");

         Console.WriteLine();
      }
   }
}
```

Output

```
Element(0): 1 3 5 7 9
Element(1): 2 4 6 8
```

See Also

Arrays | Single-Dimensional Arrays | Multidimensional Arrays

Passing Arrays Using ref and out

Like all out parameters, an **out** parameter of an array type must be assigned before it is used; that is, it must be assigned by the callee. For example:

```
public static void MyMethod(out int[] arr)
{
   arr = new int[10];   // definite assignment of arr
}
```

Like all ref parameters, a **ref** parameter of an array type must be definitely assigned by the caller. Therefore, there is no need to be definitely assigned by the callee. A **ref** parameter of an array type may be altered as a result of the call. For example, the array can be assigned the **null** value or can be initialized to a different array. For example:

```
public static void MyMethod(ref int[] arr)
{
   arr = new int[10];   // arr initialized to a different array
}
```

The following two examples demonstrate the difference between **out** and **ref** when used in passing arrays to methods.

Example 1

In this example, the array `myArray` is declared in the caller (the `Main` method), and initialized in the `FillArray` method. Then, the array elements are returned to the caller and displayed.

```
// cs_array_ref_and_out.cs
using System;
class TestOut
{
   static public void FillArray(out int[] myArray)
   {
      // Initialize the array:
      myArray = new int[5] {1, 2, 3, 4, 5};
   }

   static public void Main()
   {
      int[] myArray; // Initialization is not required

      // Pass the array to the callee using out:
      FillArray(out myArray);

      // Display the array elements:
      Console.WriteLine("Array elements are:");
      for (int i=0; i < myArray.Length; i++)
         Console.WriteLine(myArray[i]);
   }
}
```

Output

```
Array elements are:
1
2
3
4
5
```

Example 2

In this example, the array `myArray` is initialized in the caller (the `Main` method), and passed to the `FillArray` method by using the **ref** parameter. Some of the array elements are updated in the `FillArray` method. Then, the array elements are returned to the caller and displayed.

```
// cs_array_ref_and_out2.cs
using System;
class TestRef
{
   public static void FillArray(ref int[] arr)
   {
      // Create the array on demand:
      if (arr == null)
         arr = new int[10];
      // Otherwise fill the array:
      arr[0] = 123;
      arr[4] = 1024;
   }

   static public void Main ()
   {
      // Initialize the array:
      int[] myArray = {1,2,3,4,5};

      // Pass the array using ref:
      FillArray(ref myArray);

      // Display the updated array:
      Console.WriteLine("Array elements are:");
      for (int i = 0; i < myArray.Length; i++)
         Console.WriteLine(myArray[i]);
   }
}
```

Output

```
Array elements are:
123
2
3
4
1024
```

See Also

Arrays

Constructors and Destructors

This section explains the process of creation, initialization, and destruction of objects. It introduces the following topics:

- Class Constructors
- Struct Constructors
- Destructors

See Also

C# Language Features

Class Constructors

In this section, three kinds of class constructors are discussed:

Type of class constructor	Comments
Instance	Used to create and initialize instances of the class.
Private	A special type of instance constructor that is not accessible outside the class. A class c annot be instantiated using a private constructor.
Static	Called automatically to initialize the class before the first instance is created or any st atic members are referenced. The constructor cannot be called directly.

See Also

Constructors and Destructors

Instance Constructors

Instance constructors are used to create and initialize instances. Constructors are declared using the following form:

```
[attributes] [modifiers] identifier([formal-parameter-list])
[initializer] { constructor-body }
```

where:

attributes (Optional)
Additional declarative information. For more information on attributes and attribute classes, see C# Attributes.
modifiers (Optional)
The allowed modifiers are extern and the four access modifiers.
identifier
The *identifier* is the same as the class name.
formal-parameter-list (Optional)
The optional parameters passed to the constructor. The parameters must be as accessible as the constructor itself.
initializer (Optional)
Invoked before the execution of the constructor body. The *initializer* can be one of the following with an optional *argument-list*:
: base (*argument-list*)
: this (*argument-list*)
constructor-body
The block that contains the statements that initialize the class instance.

Remarks

The class constructor is invoked when you create a new object, for example:

```
Point myPoint = new Point();
```

A class can have more than one constructor. For example, you can declare a constructor without arguments, such as `Point()`, and another constructor with arguments, such as `Point(int x, int y)`.

If the class does not have a constructor, a default parameterless constructor is automatically generated for you and the default values are used to initialize the object fields (for example, an **int** is initialized to 0). For more information on default values, see Default Values Table.

The class constructor can invoke the constructor of the base class through the initializer, for example:

```
public Cylinder(double radius, double height): base(radius, height)
{
}
```

In the preceding example, the fields, `radius` and `height`, are initialized through the base class constructor. This is analogous to the C++ initialization list.

The class constructor can also invoke another constructor from the same class by using the keyword **this**, for example:

```
public Point(): this(0,20)
{
}
```

In the preceding example, the parameterless constructor `Point()` is invoking another constructor that has two arguments to initialize the default position at (0, 20).

Example 1

The following example demonstrates a class with two class constructors, one without arguments and one with two arguments.

```
// Constructor1.cs
```

```
using System;

class Point
{
   public int x, y;
   // Default constructor:
   public Point()
   {
      x = 0;
      y = 0;
   }
   // A constructor with two arguments:
   public Point(int x, int y)
   {
      this.x = x;
      this.y = y;
   }

   // Override the ToString method:
   public override string ToString()
   {
      return(String.Format("({0},{1})", x, y));
   }
}

class MainClass
{
   static void Main()
   {
      Point p1 = new Point();
      Point p2 = new Point(5,3);

      // Display the results using the overriden ToString method:
      Console.WriteLine("Point #1 at {0}", p1);
      Console.WriteLine("Point #2 at {0}", p2);
   }
}
```

Output

```
Point #1 at (0,0)
Point #2 at (5,3)
```

Example 2

In this example the class `Person` does not have any constructors, in which case, a default constructor is automatically provided and the fields are initialized to their default values.

```
// Constructor2.cs
using System;

public class Person
{
   public int age;
   public string name;
}

public class MainClass
{
   static void Main()
   {
      Person p = new Person();

      Console.Write("Name: {0}, Age: {1}",p.name, p.age);
   }
```

```
}
```

Output

```
Name: , Age: 0
```

Notice that the default value of `age` is `0` and the default value of `name` is **null**. For more information on default values, see Default Values Table.

Example 3

The following example demonstrates using the base class initializer. The `Circle` class is derived from the general class `Shape`, and the `Cylinder` class is derived from the `Circle` class. The constructor on each derived class is using its base class initializer.

```
// CtorInitializer.cs
using System;

abstract class Shape
{
   public const double pi = Math.PI;
   protected double x, y, z;

   public Shape (double x, double y, double z)
   {
      this.x = x;
      this.y = y;
      this.z = z;
   }

   public abstract double Area();
}

class Circle: Shape
{
   public Circle(double radius): base(radius,0,0)
   {
   }

   public override double Area()
   {
      return pi*x*x;
   }
}

class Cylinder: Circle
{
   public Cylinder(double radius, double height): base(radius)
   {
      y = height;
   }

   public override double Area()
   {
      return 2*(base.Area()) + 2*pi*x*y;
   }
}
class TestClass
{
   public static void Main()
   {
      double radius = 2.5, height = 3.0;
      Circle myCircle = new Circle(radius);
      Cylinder myCylinder = new Cylinder(radius, height);

      Console.WriteLine("Area of the circle   = {0:F2}",
```

```
                                myCircle.Area());
        Console.WriteLine("Area of the cylinder = {0:F2}",
                                myCylinder.Area());
    }
}
```

Output

```
Area of the circle   = 19.63
Area of the cylinder = 86.39
```

For more examples on invoking the base class constructors, see virtual, override, and base.

See Also

10.10 Instance Constructors | Constructors and Destructors | Class Constructors

Private Constructors

A private constructor is a special instance constructor. It is commonly used in classes that contain static members only. If a class has one or more private constructors and no public constructors, then other classes (except nested classes) are not allowed to create instances of this class. For example:

```
class NLog
{
   // Private Constructor:
   private NLog() {}

   public static double e = 2.71828;
}
```

The declaration of the empty constructor prevents the automatic generation of a default constructor. Note that if you don't use an access modifier with the constructor it will still be private by default. However, the **private** modifier is usually used explicitly to make it clear that the class cannot be instantiated.

Private constructors are useful to prevent creation of a class when there are no instance fields or methods, such as the Math class, or when a method is called to obtain an instance of a class.

Example

The following is an example of a class using a private constructor.

```
// PrivateCtor1.cs
using System;

public class MyClass
{
   private MyClass() {}
   public static int counter;
   public static int IncrementCounter()
   {
      return ++counter;
   }
}

class MainClass
{
   static void Main()
   {
      // If you uncomment the following statement, it will generate
      // an error because the constructor is inaccessible:
      // MyClass myObject = new MyClass();   // Error
      MyClass.counter = 100;
      MyClass.IncrementCounter();
      Console.WriteLine("New count: {0}", MyClass.counter);
   }
}
```

Output

```
New count: 101
```

Notice that if you uncomment the following statement from the example, it will generate an error because the constructor is inaccessible due to its protection level:

```
// MyClass myObject = new MyClass();   // error
```

See Also

static | 10.10.5 Private constructors | Constructors and Destructors | Class Constructors

Static Constructors

A static constructor is used to initialize a class. It is called automatically to initialize the class before the first instance is created or any static members are referenced. It is declared using the following form:

```
[attributes] static identifier( ) { constructor-body }
```

where:

attributes (Optional)
Additional declarative information. For more information on attributes and attribute classes, see C# Attributes.
identifier
The *identifier* is the same as the class name.
constructor-body
The block that contains the statements that initialize the class.

Remarks

A static constructor does not take access modifiers or have parameters.

A static constructor is called automatically to initialize the class before the first instance is created or any static members are referenced.

A static constructor cannot be called directly.

The user has no control on when the static constructor is executed in the program.

A typical use of static constructors is when the class is using a log file and the constructor is used to write entries to this file.

Example

In this example, the class `MyClass` has a static constructor and one static member, `MyMethod()`. When `MyMethod()` is called, the static constructor is invoked to initialize the class.

```
// StaticCtor1.cs
using System;
class MyClass
{
   // Static constructor:
   static MyClass()
   {
      Console.WriteLine("The static constructor invoked.");
   }

   public static void MyMethod()
   {
      Console.WriteLine("MyMethod invoked.");
   }
}

class MainClass
{
   static void Main()
   {
      MyClass.MyMethod();
   }
}
```

Output

```
The static constructor invoked.
MyMethod invoked.
```

For more information and examples of static constructors, see 1.7.10 Static constructors and 10.11 Static constructors.

See Also

static | Constructors and Destructors | Class Constructors

Struct Constructors

Struct constructors are similar to class constructors, except for the following differences:

- Structs cannot contain explicit parameterless constructors. Struct members are automatically initialized to their default values.
- A struct cannot have an initializer in the form: **base** (*argument-list*).

Example

The following is an example of a constructor using two arguments for the `Point` struct.

```
// Structctor1.cs
using System;

public struct Point
{
   public int x, y;
   // Constructor:
   public Point(int x, int y)
   {
      this.x = x;
      this.y = y;
   }
   // Override the ToString method:
   public override string ToString()
   {
      return(String.Format("({0},{1})", x, y));
   }
}

class MainClass
{
   static void Main()
   {
      // Initialize two points:
      Point p1 = new Point();
      Point p2 = new Point(10,15);

      // Display the results using the overriden ToString method:
      Console.WriteLine("Point #1 (default initialization): {0}", p1);
      Console.WriteLine("Point #2 (explicit initialization): {0}", p2);
   }
}
```

Output

```
Point #1 (default initialization): (0,0)
Point #2 (explicit initialization): (10,15)
```

For more information on structs, see struct and the Structs Tutorial.

For a complete comparison between structs and classes, see 11.3 Class and struct differences.

See Also

Constructors and Destructors | Class Constructors

Destructors

Destructors are used to destruct instances of classes. A destructor is declared using the following form:

```
[attributes] ~ identifier( ) { destructor-body }
```

where:

attributes (Optional)
Additional declarative information. For more information on attributes and attribute classes, see C# Attributes.
identifier
The *identifier* is the same as the class name.
destructor-body
The block that contains the statements that destruct the instance of the class.

Remarks

Destructors cannot be used with structs. They are only used with classes.

A class can only have one destructor.

Destructors cannot be inherited or overloaded.

Destructors cannot be called. They are invoked automatically.

A destructor does not take modifiers or have parameters. For example, the following is a declaration of a destructor for the class `MyClass`:

```
~ MyClass()
{
   // Cleanup statements.
}
```

The destructor implicitly calls the Object.Finalize method on the object's base class. Therefore, the preceding destructor code is implicitly translated to:

```
protected override void Finalize()
{
   try
   {
      // Cleanup statements.
   }
   finally
   {
      base.Finalize();
   }
}
```

This means that the **Finalize** method is called recursively for all of the instances in the inheritance chain, from the most derived to the least derived.

The programmer has no control on when the destructor is called because this is determined by the garbage collector. The garbage collector checks for objects that are no longer being used by the application. It considers these objects eligible for destruction and reclaims their memory. Destructors are also called when the program exits.

It is possible to force garbage collection by calling the GC.Collect method, but in most cases, this should be avoided because it may result in performance issues. For more information, see Forcing a Garbage Collection.

Example

The following example creates three classes that make a chain of inheritance. The class `First` is the base class, `Second` is derived from `First`, and `Third` is derived from `Second`. All three have destructors. In `Main()`, an instance of the most derived class is created. When the program runs, notice that the destructors for the three classes are called automatically, and in order, from the

most derived to the least derived.

```
// Destructors1.cs
using System;

class First
{
   ~First()
   {
      Console.WriteLine("First's destructor is called");
   }
}

class Second: First
{
   ~Second()
   {
      Console.WriteLine("Second's destructor is called");
   }
}

class Third: Second
{
   ~Third()
   {
      Console.WriteLine("Third's destructor is called");
   }
}

public class MainClass
{
   public static void Main()
   {
      Third myObject = new Third();
   }
}
```

Output

```
Third's destructor is called
Second's destructor is called
First's destructor is called
```

Using Destructors to Release Resources

In general, you should not be as concerned about memory management as you would with C++. This is because the .NET Framework garbage collector implicitly manages the allocation and release of memory for your objects. However, when your application encapsulates unmanaged resources such as windows, files, and network connections, you should use destructors to free those resources. When the object is eligible for destruction, the garbage collector runs the object's **Finalize** method.

Explicit Release of Resources

If your application is using an expensive external resource, it is also recommended that you provide a way to explicitly release the resource before the garbage collector frees the object. You do this by implementing a **Dispose** method (from the IDisposable interface) that performs the necessary cleanup for the object. This can considerably improve the performance of the application. Even with this explicit control over resources, the destructor becomes a safeguard to clean up resources if the call to the **Dispose** method failed.

For more details on cleaning up resources, see the following topics:

- Cleaning Up Unmanaged Resources
- Implementing a Dispose Method
- using Statement

See Also

Programming Essentials for Garbage Collection | 8.13 The using statement | Constructors and Destructors

Indexers

Indexers permit instances of a class or struct to be indexed in the same way as arrays. Indexers are similar to properties except that their accessors take parameters.

In this section, the following topics are introduced:

- Indexer Declaration
- Comparison Between Properties and Indexers
- Interface Indexers

For information on associated arrays, see the .NET Framework Hashtable Class.

See Also

Properties | Indexers Tutorial | C# Language Features

Indexer Declaration

Indexers allow you to index a class or a struct instance in the same way as an array. To declare an indexer, use the following:

```
[attributes] [modifiers] indexer-declarator {accessor-declarations}
```

The *indexer-declarator* takes one of the forms:

```
type this [formal-index-parameter-list]
type interface-type.this [formal-index-parameter-list]
```

The *formal-index-parameter* takes the form:

```
[attributes] type identifier
```

where:

attributes (Optional)
Additional declarative information. For more information on attributes and attribute classes, see C# Attributes.

modifiers (Optional)
Allowed modifiers are **new**, **virtual**, **sealed**, **override**, **abstract**, **extern**, and a valid combination of the four access modifiers. For details, see Access Modifiers.

indexer-declarator
Includes the *type* of the element introduced by the indexer, **this**, and the *formal-index-parameter-list*. If the indexer is an explicit interface member implementation, the *interface-type* is included.

type
A type name.

interface-type
The interface name.

formal-index-parameter-list
Specifies the parameters of the indexer. The parameter includes optional *attributes*, the index *type*, and the index *identifier*. At least one parameter must be specified. The parameter modifiers **out** and **ref** are not allowed.

accessor-declarations
The indexer accessors, which specify the executable statements associated with reading and writing indexer elements.

identifier
The parameter name.

get Accessor

The **get** accessor body of an indexer is similar to a method body. It returns the type of the indexer. The **get** accessor uses the same *formal-index-parameter-list* as the indexer. For example:

```
get
{
   return myArray[index];
}
```

set Accessor

The **set** accessor body of an indexer is similar to a method body. It uses the same *formal-index-parameter-list* as the indexer, in addition to the **value** implicit parameter. For example:

```
set
{
   myArray[index] = value;
}
```

Remarks

The type of an indexer and each of the types referenced in the *formal-index-parameter-list* must be at least as accessible as the indexer itself. For more information on accessibility levels, see Access Modifiers.

The signature of an indexer consists of the number and types of its formal parameters. It does not include the indexer type or the names of the formal parameters.

If you declare more than one indexer in the same class, they must have different signatures.

An indexer value is not classified as a variable; therefore, it is not possible to pass an indexer value as a **ref** or **out** parameter.

To provide the indexer with a name that other languages can use for the default indexed property, use a **name** attribute in the declaration. For example:

```
[System.Runtime.CompilerServices.CSharp.IndexerName("MyItem")]
public int this [int index]   // Indexer declaration
{
}
```

This indexer will have the name `MyItem`. Without providing the **name** attribute, the default name will be `Item`.

Example

The following example shows how to declare a private array field, `myArray`, and an indexer. Using the indexer allows direct access to the instance `b[i]`. The alternative to using the indexer is to declare the array as a **public** member and access its members, `myArray[i]`, directly.

```
// cs_keyword_indexers.cs
using System;
class IndexerClass
{
   private int [] myArray = new int[100];
   public int this [int index]   // Indexer declaration
   {
      get
      {
         // Check the index limits.
         if (index < 0 || index >= 100)
            return 0;
         else
            return myArray[index];
      }
      set
      {
         if (!(index < 0 || index >= 100))
            myArray[index] = value;
      }
   }
}

public class MainClass
{
   public static void Main()
   {
      IndexerClass b = new IndexerClass();
      // Call the indexer to initialize the elements #3 and #5.
      b[3] = 256;
      b[5] = 1024;
      for (int i=0; i<=10; i++)
      {
         Console.WriteLine("Element #{0} = {1}", i, b[i]);
      }
   }
}
```

Output

```
Element #0 = 0
Element #1 = 0
Element #2 = 0
Element #3 = 256
Element #4 = 0
Element #5 = 1024
Element #6 = 0
Element #7 = 0
Element #8 = 0
Element #9 = 0
Element #10 = 0
```

Notice that when an indexer's access is evaluated (for example, in a **Console.Write** statement) the **get** accessor is invoked. Therefore, if no **get** accessor exists, a compile-time error occurs.

For an additional example, see 10.8 Indexers.

See Also

Indexers | Properties | Accessors | Indexers Tutorial

Comparison Between Properties and Indexers

Indexers are similar to properties. Except for the differences shown in the following table, all of the rules defined for property accessors apply to indexer accessors as well.

Property	**Indexer**
Identified by its name.	Identified by its signature.
Accessed through a simple name or a member access.	Accessed through an element access.
Can be a static or an instance member.	Must be an instance member.
A **get** accessor of a property has no parameters.	A **get** accessor of an indexer has the same formal parameter list as the indexer.
A **set** accessor of a property contains the implicit **value** parameter.	A **set** accessor of an indexer has the same formal parameter list as the indexer, in addition to the **value** parameter.

See Also

Indexers | Properties | Accessors

Interface Indexers

Indexers can be declared on interfaces. The declaration takes the following form:

```
[attributes] [new] type this [formal-index-parameter-list] {interface-accessors}
```

where:

attributes (Optional)
Same as indexers on classes.
type
Same as indexers on classes.
formal-index-parameter-list
Same as indexers on classes.
interface-accessors
The indexer accessors.

Remarks

Accessors of interface indexers differ from the accessors of class indexers in the following ways:

- Interface accessors do not use modifiers.
- An interface accessor does not have a body.

Thus, the purpose of the accessor is to indicate whether the indexer is read-write, read-only, or write-only.

The following is an example of an interface indexer accessor:

```
public interface IMyInterface
{
   ...
   // Indexer declaration:
   string this[int index]
   {
      get;
      set;
   }
}
```

The signature of an indexer must differ from the signatures of all other indexers declared in the same interface.

Example

The following example shows how to implement interface indexers.

```
// cs_interface_indexers.cs
using System;
// Indexer on an interface:
public interface IMyInterface
{
   // Indexer declaration:
   int this[int index]
   {
      get;
      set;
   }
}

// Implementing the interface.
class IndexerClass : IMyInterface
{
   private int [] myArray = new int[100];
   public int this [int index]   // indexer declaration
```

```
    {
       get
       {
          // Check the index limits.
          if (index < 0 || index >= 100)
             return 0;
          else
             return myArray[index];
       }
       set
       {
          if (!(index < 0 || index >= 100))
             myArray[index] = value;
       }
    }
}

public class MainClass
{
   public static void Main()
   {
      IndexerClass b = new IndexerClass();
      // Call the indexer to initialize the elements #3 and #5.
      b[2] = 4;
      b[5] = 32;
      for (int i=0; i<=10; i++)
      {
         Console.WriteLine("Element #{0} = {1}", i, b[i]);
      }
   }
}
```

Output

```
Element #0 = 0
Element #1 = 0
Element #2 = 4
Element #3 = 0
Element #4 = 0
Element #5 = 32
Element #6 = 0
Element #7 = 0
Element #8 = 0
Element #9 = 0
Element #10 = 0
```

In the preceding example, you could use the explicit interface member implementation by using the fully qualified name of the interface member. For example:

```
public string IMyInterface.this
{
}
```

However, the fully qualified name is only needed to avoid ambiguity when the class is implementing more than one interface with the same indexer signature. For example, if the `Employee` class is implementing two interfaces, `ICitizen` and `IEmployee`, and both interfaces have the same indexer signature, the explicit interface member implementation is necessary. That is, the following indexer declaration:

```
public string IEmployee.this
{
}
```

implements the indexer on the `IEmployee` interface, while the following declaration:

```
public string ICitizen.this
{
}
```

implements the indexer on the `ICitizen` interface.

For more information, see 13.4.1 Explicit interface member implementations.

See Also

Indexers | interface

Main

The `Main` method is the entry point of your program, where the program control starts and ends. It is declared inside a class or struct. It must be static. It can either have a **void** or **int** return type. The `Main` method is where you create objects and invoke other methods. The `Main` method can be declared without parameters or with parameters. The latter form allows your program to read command-line arguments.

See the following topics for more information and examples on using the `Main` method:

- Return Values
- Command-Line Arguments

See Also

C# Language Features

Return Values

The `Main` method can be of the type **void**:

```
static void Main()
{
}
```

It can also return an **int**:

```
static int Main()
{
   return 0;
}
```

Example

In this example, the program contains two classes, `Factorial` and `MainClass`. The `Main` method, which resides in the `MainClass` class, is used to read a number from the keyboard, invoke the `Fac` method from the `Factorial` class, and calculate and display the factorial of the input number.

```
// cs_main.cs
using System;
public class Factorial
{
   public static long Fac(long i)
   {
      return ((i <= 1) ? 1 : (i * Fac(i-1)));
   }
}

class MainClass
{
   public static void Main()
   {
      // Read a string from the keyboard:
      Console.Write("Enter an integer: ");

      string s = Console.ReadLine();

      // Convert the string to long:

      try
      {
         long num = Int64.Parse(s);
         Console.WriteLine("The Factorial of {0} is {1}.",
                           num, Factorial.Fac(num));
      }
      catch (System.FormatException)
      {
         Console.WriteLine("Invalid input specified");
      }
   }
}
```

Input

```
5
```

Sample Output

```
Enter an integer: 5
The Factorial of 5 is 120.
```

See Also

Command-Line Arguments | Main

Command-Line Arguments

The `Main` method can use arguments, in which case, it takes one of the following forms:

```
static int Main(string[] args)
static void Main(string[] args)
```

The parameter of the `Main` method is a `string` array that represents the command-line arguments. Usually you check for the existence of the arguments by testing the `Length` property, for example:

```
if (args.Length == 0)
{
   Console.WriteLine("Please enter a numeric argument.");
   return 1;
}
```

You can also convert the string arguments to numeric types by using the `Convert` class or the `Parse` method. For example, the following statement converts the string to a long number by using the `Parse` method on the `Int64` class:

```
long num = Int64.Parse(args[0]);
```

It is also possible to use the C# type `long`, which aliases `Int64`:

```
long num = long.Parse(args[0]);
```

You can also use the `Convert` class method `ToInt64` to do the same thing:

```
long num = Convert.ToInt64(s);
```

For more information see Parse Method and Convert Class.

Example

In this example, the program takes one argument at run time, converts the argument to a long number, and calculates the factorial of the number. If no arguments are supplied, the program issues a message that explains the correct usage of the program.

```
// Factorial_main.cs
// arguments: 3
using System;
public class Factorial
{
   public static long Fac(long i)
   {
      return ((i <= 1) ? 1 : (i * Fac(i-1)));
   }
}

class MainClass
{
   public static int Main(string[] args)
   {
      // Test if input arguments were supplied:
      if (args.Length == 0)
      {
         Console.WriteLine("Please enter a numeric argument.");
         Console.WriteLine("Usage: Factorial <num>");
         return 1;
      }
```

```
        // Convert the input arguments to numbers:
        try
        {
             long num = long.Parse(args[0]);
             Console.WriteLine("The Factorial of {0} is {1}.",
                          num, Factorial.Fac(num));
             return 0;
        }
        catch (System.FormatException)
        {
           Console.WriteLine("Please enter a numeric argument.");
           Console.WriteLine("Usage: Factorial <num>");
           return 1;
        }
    }
}
```

Output

```
The Factorial of 3 is 6.
```

The following are two sample runs of the program assuming that the program name is Factorial.exe.

Run #1:

```
Enter the following command line:
Factorial 10
You get the following result:
The Factorial of 10 is 3628800.
```

Run #2:

```
Enter the following command line:
Factorial
You get the following result:
Please enter a numeric argument.
Usage: Factorial <num>
```

For more examples on using command-line arguments, see the example in Creating and Using C# DLLs.

See Also

Main | Return Values | General Structure of a C# Program | Command Line Parameters Tutorial

Properties

Properties are named members of classes, structs, and interfaces. They provide a flexible mechanism to read, write, or compute the values of private fields through accessors.

In this section, the following topics are introduced:

- Property Declaration
- Accessors
- Interface Properties
- value

See Also

Indexers | Properties Tutorial | C# Language Features

Property Declaration

Properties are an extension of fields and are accessed using the same syntax. Unlike fields, properties do not designate storage locations. Instead, properties have accessors that read, write, or compute their values.

Property declaration takes one of the following forms:

```
[attributes] [modifiers] type identifier {accessor-declaration}
[attributes] [modifiers] type interface-type.identifier {accessor-declaration}
```

where:

attributes (Optional)
Additional declarative information. For more information on attributes and attribute classes, see C# Attributes.
modifiers (Optional)
The allowed modifiers are new, static, virtual, abstract, override, and a valid combination of the four access modifiers.
type
The property type, which must be at least as accessible as the property itself. For more information on accessibility levels, see Access Modifiers.
identifier
The property name. For more information on interface member implementation, see interface.
accessor-declaration
Declaration of the property accessors, which are used to read and write the property.
interface-type
The interface in a fully qualified property name. See Interface Properties.

Remarks

Unlike fields, properties are not classified as variables. Therefore, it is not possible to pass a property as a ref or out parameter.

A property declared using the **static** modifier is classified as a static property; otherwise, it is classified as an instance property. Like other static members, a static property is not associated with a specific instance and cannot be referenced through an instance. Instead, it is associated with the type and can only be referenced through the type name. For example, in the following statements:

```
Button okButton = new Button();
// Using an instance property:
string s = okButton.Caption;
```

the `Caption` property is associated with the instance `okButton`. If `Caption` is declared as a static property, the class name (`Button`) must be used instead:

```
// Using a static property:
string s = Button.Caption;
```

An instance of a class can be accessed using this in the accessors of an instance property, but it is an error to use **this** in the accessors of a static property.

It is an error to use a virtual, abstract, or override modifier on an accessor of a static property.

Example

This example demonstrates instance, static, and read-only properties. It accepts the name of the employee from the keyboard, increments `numberOfEmployees` by 1, and displays the Employee name and number.

```
// property.cs
// Properties
using System;
public class Employee
{
   public static int numberOfEmployees;
```

```
    private static int counter;
    private string name;

    // A read-write instance property:
    public string Name
    {
       get
       {
          return name;
       }
       set
       {
          name = value;
       }
    }

    // A read-only static property:
    public static int Counter
    {
       get
       {
          return counter;
       }
    }

    // Constructor:
    public Employee()
    {
       // Calculate the employee's number:
       counter = ++counter + numberOfEmployees;
    }
 }

 public class MainClass
 {
    public static void Main()
    {
       Employee.numberOfEmployees = 100;
       Employee e1 = new Employee();
       e1.Name = "Claude Vige";
       Console.WriteLine("Employee number: {0}", Employee.Counter);
       Console.WriteLine("Employee name: {0}", e1.Name);
    }
 }
```

Output

```
Employee number: 101
Employee name: Claude Vige
```

See Also

Properties | Accessors | Indexers | Properties Tutorial

Accessors

The accessor of a property contains the executable statements associated with getting (reading or computing) or setting (writing) the property. The accessor declarations can contain a **get** accessor, a **set** accessor, or both. The declarations take the following forms:

```
set {accessor-body}
get {accessor-body}
```

where:

accessor-body
The block that contains the statements to be executed when the accessor is invoked.

The get Accessor

The body of the **get** accessor is similar to that of a method. It must return a value of the property type. The execution of the **get** accessor is equivalent to reading the value of the field. The following is a **get** accessor that returns the value of a private field `name`:

```
private string name;   // the name field
public string Name   // the Name property
{
   get
   {
      return name;
   }
}
```

When you reference the property, except as the target of an assignment, the **get** accessor is invoked to read the value of the property. For example:

```
Employee e1 = new Employee();
...
Console.Write(e1.Name);   // The get accessor is invoked here
```

The **get** accessor must terminate in a **return** or **throw** statement, and control cannot flow off the accessor body.

The set Accessor

The **set** accessor is similar to a method that returns **void**. It uses an implicit parameter called **value**, whose type is the type of the property. In the following example, a **set** accessor is added to the `Name` property:

```
public string Name
{
   get
   {
      return name;
   }
   set
   {
      name = value;
   }
}
```

When you assign a value to the property, the **set** accessor is invoked with an argument that provides the new value. For example:

```
e1.Name = "Joe";   // The set accessor is invoked here
```

It is an error to use the implicit parameter name (**value**) for a local variable declaration in a **set** accessor.

Remarks

A property is classified according to the accessors used as follows:

- A property with a **get** accessor only is called a read-only property. You cannot assign a value to a read-only property.
- A property with a **set** accessor only is called a write-only property. You cannot reference a write-only property except as a target of an assignment.
- A property with both **get** and **set** accessors is a read-write property.

In a property declaration, both the **get** and **set** accessors must be declared inside the body of the property.

It is a bad programming style to change the state of the object by using the **get** accessor. For example, the following accessor produces the side effect of changing the state of the object each time the `number` field is accessed.

```
public int Number
{
   get
   {
      return number++;   // Don't do this
   }
}
```

The **get** accessor can either be used to return the field value or to compute it and return it. For example:

```
public string Name
{
   get
   {
      return name != null ? name : "NA";
   }
}
```

In the preceding code segment, if you don't assign a value to the `Name` property, it will return the value `NA`.

Example 1

This example demonstrates how to access a property in a base class that is hidden by another property with the same name in a derived class.

```
// property_hiding.cs
// Property hiding
using System;
public class BaseClass
{
   private string name;
   public string Name
   {
      get
      {
         return name;
      }
      set
      {
         name = value;
      }
   }
}

public class DerivedClass : BaseClass
{
   private string name;
   public new string Name   // Notice the use of the new modifier
   {
      get
```

```
        {
           return name;
        }
        set
        {
           name = value;
        }
    }
}

public class MainClass
{
    public static void Main()
    {
        DerivedClass d1 = new DerivedClass();
        d1.Name = "John";  // Derived class property
        Console.WriteLine("Name in the derived class is: {0}",d1.Name);
        ((BaseClass)d1).Name = "Mary"; // Base class property
        Console.WriteLine("Name in the base class is: {0}",
           ((BaseClass)d1).Name);
    }
}
```

Output

```
Name in the derived class is: John
Name in the base class is: Mary
```

The following are important points shown in the preceding example:

- The property `Name` in the derived class hides the property `Name` in the base class. In such a case, the **new** modifier is used in the declaration of the property in the derived class:

  ```
  public new string Name
  {
  ...
  ```

- The cast `(BaseClass)` is used to access the hidden property in the base class:

  ```
  ((BaseClass)d1).Name = "Mary";
  ```

For more information on hiding members, see the new modifier.

Example 2

In this example, two classes, `Cube` and `Square`, implement an abstract class, `Shape`, and override its abstract `Area` property. Note the use of the override modifier on the properties. The program accepts the side as an input and calculates the areas for the square and cube. It also accepts the area as an input and calculates the corresponding side for the square and cube.

```
// overridding_properties.cs
// Overriding properties
using System;
abstract class Shape
{
    public abstract double Area
    {
        get;
        set;
    }
}

class Square: Shape
{
```

```
      public double side;

      // Constructor:
      public Square(double s)
      {
         side = s;
      }

   // The Area property
   public override double Area
   {
      get
      {
         return side*side ;
      }
      set
      {
         // Given the area, compute the side
         side = Math.Sqrt(value);
      }
   }
}

class Cube: Shape
{
      public double side;

      // Constructor:
      public Cube(double s)
      {
         side = s;
      }

      // The Area property
      public override double Area
      {
         get
         {
            return 6*side*side;
         }
         set
         {
            // Given the area, compute the side
            side = Math.Sqrt(value/6);
         }
      }
}

public class MainClass
{
   public static void Main()
   {
      // Input the side:
      Console.Write("Enter the side: ");
      string sideString = Console.ReadLine();
      double side = double.Parse(sideString);

      // Compute areas:
      Square s = new Square(side);
      Cube c = new Cube(side);

      // Display results:
      Console.WriteLine("Area of a square = {0:F2}",s.Area);
      Console.WriteLine("Area of a cube = {0:F2}", c.Area);

      // Input the area:
      Console.Write("Enter the area: ");
      string areaString = Console.ReadLine();
```

```
        double area = double.Parse(areaString);

        // Compute sides:
        s.Area = area;
        c.Area = area;

        // Display results:
        Console.WriteLine("Side of a square = {0:F2}", s.side);
        Console.WriteLine("Side of a cube = {0:F2}", c.side);
    }
}
```

Input

```
4
24
```

Sample Output

```
Enter the side: 4
Area of a square = 16.00
Area of a cube = 96.00
Enter the area: 24
Side of a square = 4.90
Side of a cube = 2.00
```

See Also

Properties | virtual | override | abstract | base | Indexers | new modifier | Properties Tutorial | Indexers Tutorial

Interface Properties

Properties can be declared on interfaces. The declaration takes the following form:

```
[attributes] [new] type identifier {interface-accessors}
```

where:

attributes (Optional)
  Same as properties on classes.
type
  Same as properties on classes.
identifier
  The property name.
interface-accessors
  The property accessors. See the Remarks section below.

Remarks

The accessor of an interface property does not have a body.

Thus, the purpose of the accessors is to indicate whether the property is read-write, read-only, or write-only.

The following is an example of an interface indexer accessor:

```
public interface IMyInterface
{
   // Property declaration:
   string Name
   {
      get;
      set;
   }
}
```

Example

In this example, the interface `IEmployee` has a read-write property, `Name`, and a read-only property, `Counter`. The class `Employee` implements the `IEmployee` interface and uses these two properties. The program reads the name of a new employee and the current number of employees and displays the employee name and the computed employee number.

```
// cs_interface_properties.cs
// Interface Properties
using System;
interface IEmployee
{
   string Name
   {
      get;
      set;
   }

   int Counter
   {
      get;
   }
}

public class Employee: IEmployee
{
   public static int numberOfEmployees;
   private int counter;
   private string name;
```

```
      // Read-write instance property:
      public string Name
      {
         get
         {
            return name;
         }
         set
         {
            name = value;
         }
      }
      // Read-only instance property:
      public int Counter
      {
         get
         {
            return counter;
         }
      }
      // Constructor:
      public Employee()
      {
         counter = ++counter + numberOfEmployees;
      }
   }

   public class MainClass
   {
      public static void Main()
      {
         Console.Write("Enter number of employees: ");
         string s = Console.ReadLine();
         Employee.numberOfEmployees = int.Parse(s);
         Employee e1 = new Employee();
         Console.Write("Enter the name of the new employee: ");
         e1.Name = Console.ReadLine();
         Console.WriteLine("The employee information:");
         Console.WriteLine("Employee number: {0}", e1.Counter);
         Console.WriteLine("Employee name: {0}", e1.Name);
      }
   }
```

Input

```
210
Hazem Abolrous
```

Sample Output

```
Enter number of employees: 201
Enter the name of the new employee: Hazem Abolrous
The employee information:
Employee number: 202
Employee name: Hazem Abolrous
```

In the preceding example, you could use the fully qualified name of the property, which references the interface in which the member is declared. For example:

```
public string IEmployee.Name
{
}
```

However, the fully qualified name is only needed to avoid ambiguity when the class is implementing more than one interface with the same property signature. For example, if the class `Employee` is implementing two interfaces `ICitizen` and `IEmployee` and both interfaces have the `Name` property, the explicit interface member implementation will be necessary. That is, the following property declaration:

```
public string IEmployee.Name
{
}
```

implements the `Name` property on the `IEmployee` interface, while the following declaration:

```
public string ICitizen.Name
{
}
```

implements the `Name` property on the `ICitizen` interface.

For more information, see 13.4.1 Explicit interface member implementations.

See Also

Properties | Accessors | Comparison Between Properties and Indexers

value

The implicit parameter **value** is used in setting accessors and to add or remove event handlers.

For more information on the use of **value**, see event and Accessors.

See Also

Indexers Tutorial | Properties Tutorial

Passing Parameters

In C#, parameters can be passed either by value or by reference. Passing parameters by reference allows function members (methods, properties, indexers, operators, and constructors) to change the value of the parameters and have that change persist. To pass a parameter by reference, use the **ref** or **out** keyword. For simplicity, only the **ref** keyword is used in the examples of this topic. For information on the difference between **ref** and **out**, see ref, out, and Passing Arrays Using ref and out.

This topic includes the following sections:

- Passing Value-Type Parameters
- Passing Reference-Type Parameters

It also includes the following examples:

Example	Demonstrates	Uses ref or out
1	Passing value types by value	No
2	Passing value types by reference	Yes
3	Swapping value types (two integers)	Yes
4	Passing reference types by value	No
5	Passing reference types by reference	Yes
6	Swapping reference types (two strings)	Yes

Passing Value-Type Parameters

A value-type variable contains its data directly as opposed to a reference-type variable, which contains a reference to its data. Therefore, passing a value-type variable to a method means passing a copy of the variable to the method. Any changes to the parameter that take place inside the method have no affect on the original data stored in the variable. If you want the called method to change the value of the parameter, you have to pass it by reference, using the **ref** or **out** keyword. For simplicity, the following examples use **ref**.

Example 1: Passing Value Types by Value

The following example demonstrates passing value-type parameters by value. The variable `myInt` is passed by value to the method `SquareIt`. Any changes that take place inside the method have no affect on the original value of the variable.

```
// PassingParams1.cs
using System;
class PassingValByVal
{
    static void SquareIt(int x)
    // The parameter x is passed by value.
    // Changes to x will not affect the original value of myInt.
    {
        x *= x;
        Console.WriteLine("The value inside the method: {0}", x);
    }
    public static void Main()
    {
        int myInt = 5;
        Console.WriteLine("The value before calling the method: {0}",
           myInt);
        SquareIt(myInt);   // Passing myInt by value.
        Console.WriteLine("The value after calling the method: {0}",
           myInt);
    }
}
```

Output

```
The value before calling the method: 5
The value inside the method: 25
The value after calling the method: 5
```

Code Discussion

The variable `myInt`, being a value type, contains its data (the value `5`). When `SquareIt` is invoked, the contents of `myInt` are copied into the parameter `x`, which is squared inside the method. In Main, however, the value of `myInt` is the same, before and after calling the `SquareIt` method. In fact, the change that takes place inside the method only affects the local variable `x`.

Example 2: Passing Value Types by Reference

The following example is the same as Example 1, except for passing the parameter using the **ref** keyword. The value of the parameter is changed after calling the method.

```
// PassingParams2.cs
using System;
class PassingValByRef
{
    static void SquareIt(ref int x)
    // The parameter x is passed by reference.
    // Changes to x will affect the original value of myInt.
    {
        x *= x;
        Console.WriteLine("The value inside the method: {0}", x);
    }
    public static void Main()
    {
        int myInt = 5;
        Console.WriteLine("The value before calling the method: {0}",
           myInt);
        SquareIt(ref myInt);   // Passing myInt by reference.
        Console.WriteLine("The value after calling the method: {0}",
           myInt);
    }
}
```

Output

```
The value before calling the method: 5
The value inside the method: 25
The value after calling the method: 25
```

Code Discussion

In this example, it is not the value of `myInt` that is passed; rather, a reference to `myInt` is passed. The parameter `x` is not an **int**; it is a reference to an **int** (in this case, a reference to `myInt`). Therefore, when `x` is squared inside the method, what actually gets squared is what `x` refers to: `myInt`.

Example 3: Swapping Value Types

A common example of changing the values of the passed parameters is the `Swap` method, where you pass two variables, `x` and `y`, and have the method swap their contents. You must pass the parameters to the `Swap` method by reference; otherwise, you will be dealing with a local copy of the parameters inside the method. The following is an example of the `Swap` method that uses reference parameters:

```
static void SwapByRef(ref int x, ref int y)
{
    int temp = x;
    x = y;
    y = temp;
}
```

When you call this method, use the **ref** keyword in the call, like this:

```
SwapByRef (ref i, ref j);
```

Passing Reference-Type Parameters

A variable of a reference type does not contain its data directly; it contains a reference to its data. When you pass a reference-type parameter by value, it is possible to change the data pointed to by the reference, such as the value of a class member. However, you cannot change the value of the reference itself; that is, you cannot use the same reference to allocate memory for a new class and have it persist outside the block. To do that, pass the parameter using the **ref** (or **out**) keyword. For simplicity, the following examples use **ref**.

Example 4: Passing Reference Types by Value

The following example demonstrates passing a reference-type parameter, `myArray`, by value, to a method, `Change`. Because the parameter is a reference to `myArray`, it is possible to change the values of the array elements. However, the attempt to reassign the parameter to a different memory location only works inside the method and does not affect the original variable, `myArray`.

```
// PassingParams4.cs
// Passing an array to a method without the ref keyword.
// Compare the results to those of Example 5.
using System;
class PassingRefByVal
{
   static void Change(int[] arr)
   {
      arr[0]=888;   // This change affects the original element.
      arr = new int[5] {-3, -1, -2, -3, -4};   // This change is local.
      Console.WriteLine("Inside the method, the first element is: {0}", arr[0]);
   }

   public static void Main()
   {
      int[] myArray = {1,4,5};
      Console.WriteLine("Inside Main, before calling the method, the first element is: {0}",
myArray [0]);
      Change(myArray);
      Console.WriteLine("Inside Main, after calling the method, the first element is: {0}", m
yArray [0]);
   }
}
```

Output

```
Inside Main, before calling the method, the first element is: 1
Inside the method, the first element is: -3
Inside Main, after calling the method, the first element is: 888
```

Code Discussion

In the preceding example, the array, `myArray`, which is a reference type, is passed to the method without the **ref** parameter. In such a case, a copy of the reference, which points to `myArray`, is passed to the method. The output shows that it is possible for the method to change the contents of an array element (from `1` to `888`). However, allocating a new portion of memory by using the **new** operator inside the `Change` method makes the variable `arr` reference a new array. Thus, any changes after that will not affect the original array, `myArray`, which is created inside `Main`. In fact, two arrays are created in this example, one inside `Main` and one inside the `Change` method.

Example 5: Passing Reference Types by Reference

This example is the same as Example 4, except for using the **ref** keyword in the method header and call. Any changes that take place in the method will affect the original variables in the calling program.

```
// PassingParams5.cs
// Passing an array to a method with the ref keyword.
// Compare the results to those of Example 4.
using System;
class PassingRefByRef
{
   static void Change(ref int[] arr)
   {
      // Both of the following changes will affect the original variables:
      arr[0]=888;
      arr = new int[5] {-3, -1, -2, -3, -4};
      Console.WriteLine("Inside the method, the first element is: {0}", arr[0]);
   }

   public static void Main()
   {
      int[] myArray = {1,4,5};
      Console.WriteLine("Inside Main, before calling the method, the first element is: {0}",
myArray [0]);
      Change(ref myArray);
      Console.WriteLine("Inside Main, after calling the method, the first element is: {0}", m
yArray [0]);
   }
}
```

Output

```
Inside Main, before calling the method, the first element is: 1
Inside the method, the first element is: -3
Inside Main, after calling the method, the first element is: -3
```

Code Discussion

All of the changes that take place inside the method affect the original array in Main. In fact, the original array is reallocated using the **new** operator. Thus, after calling the `Change` method, any reference to `myArray` points to the five-element array, which is created in the `Change` method.

Example 6: Swapping Two Strings

Swapping strings is a good example of passing reference-type parameters by reference. In the example, two strings, `str1` and `str2`, are initialized in Main and passed to the `SwapStrings` method as parameters modified by the **ref** keyword. The two strings are swapped inside the method and inside Main as well.

```
// PassingParams6.cs
using System;
class SwappinStrings
{
    static void SwapStrings(ref string s1, ref string s2)
    // The string parameter x is passed by reference.
    // Any changes on parameters will affect the original variables.
    {
        string temp = s1;
        s1 = s2;
        s2 = temp;
        Console.WriteLine("Inside the method: {0}, {1}", s1, s2);
    }
    public static void Main()
    {
        string str1 = "John";
        string str2 = "Smith";
        Console.WriteLine("Inside Main, before swapping: {0} {1}",
           str1, str2);
        SwapStrings(ref str1, ref str2);   // Passing strings by reference
```

```
        Console.WriteLine("Inside Main, after swapping: {0}, {1}",
           str1, str2);
    }
}
```

Output

```
Inside Main, before swapping: John Smith
Inside the method: Smith, John
Inside Main, after swapping: Smith, John
```

Code Discussion

In this example, the parameters need to be passed by reference to affect the variables in the calling program. If you remove the **ref** keyword from both the method header and the method call, no changes will take place in the calling program.

For more information on strings, see the string keyword.

See Also

Passing Arrays Using ref and out | C# Language Features

XML Documentation

In Visual C# you can document the code you write using XML. C# is the only programming language in Visual Studio .NET with this feature. For details on creating an XML file with documentation comments, see the following topics.

For information about	See
The C# compiler option to use	/doc (Process Documentation Comments)
Tags you can use to provide commonly used functionality in documentation	Recommended Tags for Documentation Comments
The ID strings that the compiler produces to identify the constructs in your code	Processing the XML File
How to delimit documentation tags	Delimiters for Documentation Tags
How to document a type	XML Documentation Tutorial
Links to information about XML as it relates to Visual Studio feature areas	XML in Visual Studio

See Commenting Code for information on how Visual Studio will display a report on the structure of your code, to include XML Documentation comments.

See Also

C# Language Features

Recommended Tags for Documentation Comments

The C# compiler will process documentation comments in your code to an XML file. Processing the XML file to create documentation is a detail that needs to be implemented at your site.

Tags are processed on code constructs such as types and type members.

> **Note** Tags are not processed on namespaces.

The compiler will process any tag that is valid XML. The following tags provide commonly used functionality in user documentation:

<c>	<para>	<see>[1]
<code>	<param>[1]	<seealso>[1]
<example>	<paramref>[1]	<summary>
<exception>[1]	<permission>[1]	<value>
<include>[1]	<remarks>	
<list>	<returns>	

1. Compiler verifies syntax.

See Also

/doc (Process Documentation Comments) | XML Documentation

<c>

```
<c>text</c>
```

where:

text
The text you would like to indicate as code.

Remarks

The <c> tag gives you a way to indicate that text within a description should be marked as code. Use <code> to indicate multiple lines as code.

Compile with /doc to process documentation comments to a file.

Example

```
// xml_c_tag.cs
// compile with: /doc:xml_c_tag.xml

/// text for class MyClass
public class MyClass
{
   /// <summary><c>MyMethod</c> is a method in the <c>MyClass</c> class.
   /// </summary>
   public static void MyMethod(int Int1)
   {
   }
   /// text for Main
   public static void Main ()
   {
   }
}
```

See Also

Recommended Tags for Documentation Comments

<code>

```
<code>content</code>
```

where:

content
The text you want marked as code.

Remarks

The <code> tag gives you a way to indicate multiple lines as code. Use <c> to indicate that text within a description should be marked as code.

Compile with /doc to process documentation comments to a file.

Example

See the <example> topic for an example of how to use the <code> tag.

See Also

Recommended Tags for Documentation Comments

<example>

```
<example>description</example>
```

where:

description
A description of the code sample.

Remarks

The <example> tag lets you specify an example of how to use a method or other library member. Commonly, this would involve use of the <code> tag.

Compile with /doc to process documentation comments to a file.

Example

```
// xml_example_tag.cs
// compile with: /doc:xml_ctag.xml

/// text for class MyClass
public class MyClass
{
   /// <summary>
   /// The GetZero method.
   /// </summary>
   /// <example> This sample shows how to call the GetZero method.
   /// <code>
   ///   class MyClass
   ///   {
   ///      public static int Main()
   ///      {
   ///         return GetZero();
   ///      }
   ///   }
   /// </code>
   /// </example>
   public static int GetZero()
   {
      return 0;
   }
   /// text for Main
   public static void Main ()
   {
   }
}
```

See Also

Recommended Tags for Documentation Comments

<exception>

```
<exception cref="member">description</exception>
```

where:

cref = "*member*"
A reference to an exception that is available from the current compilation environment. The compiler checks that the given exception exists and translates *member* to the canonical element name in the output XML. *member* must appear within double quotation marks (" ").

description
A description.

Remarks

The <exception> tag lets you specify which exceptions can be thrown. This tag is applied to a method definition.

Compile with /doc to process documentation comments to a file.

Example

```
// xml_exception_tag.cs
// compile with: /doc:xml_exception_tag.xml
using System;

/// comment for class
public class EClass : Exception
{
   // class definition ...
}

/// <exception cref="System.Exception">Thrown when... .</exception>
class TestClass
{
   public static void Main()
   {
      try
      {
      }
      catch(EClass)
      {
      }
   }
}
```

See Also

Recommended Tags for Documentation Comments

<include>

```
<include file='filename' path='tagpath[@name="id"]' />
```

where:

filename
The name of the file containing the documentation. The file name can be qualified with a path. Enclose *filename* in single quotation marks (' ').

tagpath
The path of the tags in *filename* that leads to the tag *name*. Enclose the path in single quotation marks (' ').

name
The name specifier in the tag that precedes the comments; *name* will have an *id*.

id
The ID for the tag that precedes the comments. Enclose the ID in double quotation marks (" ").

Remarks

The <include> tag lets you refer to comments in another file that describe the types and members in your source code. This is an alternative to placing documentation comments directly in your source code file.

The <include> tag uses the XML XPath syntax. Refer to XPath documentation for ways to customize your <include> use.

Example

This is a multifile example. The first file, which uses <include>, is listed below:

```
// xml_include_tag.cs
// compile with: /doc:xml_include_tag.xml
/// <include file='xml_include_tag.doc' path='MyDocs/MyMembers[@name="test"]/*' />
class Test
{
   public static void Main()
   {
   }
}

/// <include file='xml_include_tag.doc' path='MyDocs/MyMembers[@name="test2"]/*' />
class Test2
{
   public void Test()
   {
   }
}
```

The second file, xml_include_tag.doc, contains the following documentation comments:

```
<MyDocs>

<MyMembers name="test">
<summary>
The summary for this type.
</summary>
</MyMembers>

<MyMembers name="test2">
<summary>
The summary for this other type.
</summary>
</MyMembers>

</MyDocs>
```

Program Output

```
<?xml version="1.0"?>
<doc>
    <assembly>
        <name>t2</name>
    </assembly>
    <members>
        <member name="T:Test">
            <summary>
The summary for this type.
</summary>
        </member>
        <member name="T:Test2">
            <summary>
The summary for this other type.
</summary>
        </member>
    </members>
</doc>
```

See Also

Recommended Tags for Documentation Comments

<list>

```
<list type="bullet" | "number" | "table">
   <listheader>
      <term>term</term>
      <description>description</description>
   </listheader>
   <item>
      <term>term</term>
      <description>description</description>
   </item>
</list>
```

where:

term
A term to define, which will be defined in *text*.

description
Either an item in a bullet or numbered list or the definition of a *term*.

Remarks

The <listheader> block is used to define the heading row of either a table or definition list. When defining a table, you only need to supply an entry for term in the heading.

Each item in the list is specified with an <item> block. When creating a definition list, you will need to specify both *term* and *text*. However, for a table, bulleted list, or numbered list, you only need to supply an entry for *text*.

A list or table can have as many <item> blocks as needed.

Compile with /doc to process documentation comments to a file.

Example

```
// xml_list_tag.cs
// compile with: /doc:xml_list_tag.xml

/// text for class MyClass
public class MyClass
{
   /// <remarks>Here is an example of a bulleted list:
   /// <list type="bullet">
   /// <item>
   /// <description>Item 1.</description>
   /// </item>
   /// <item>
   /// <description>Item 2.</description>
   /// </item>
   /// </list>
   /// </remarks>
   public static void Main ()
   {
   }
}
```

See Also

Recommended Tags for Documentation Comments

<para>

```
<para>content</para>
```

where:

content
The text of the paragraph.

Remarks

The <para> tag is for use inside a tag, such as <summary>, <remarks>, or <returns>, and lets you add structure to the text.

Compile with /doc to process documentation comments to a file.

Example

See <summary> for an example of using <para>.

See Also

Recommended Tags for Documentation Comments

<param>

```
<param name='name'>description</param>
```

where:

name
The name of a method parameter. Enclose the name in single quotation marks (' ').
description
A description for the parameter.

Remarks

The <param> tag should be used in the comment for a method declaration to describe one of the parameters for the method.

The text for the <param> tag will be displayed in IntelliSense, the Object Browser, and in the Code Comment Web Report.

Compile with /doc to process documentation comments to a file.

Example

```
// xml_param_tag.cs
// compile with: /doc:xml_param_tag.xml

/// text for class MyClass
public class MyClass
{
   /// <param name="Int1">Used to indicate status.</param>
   public static void MyMethod(int Int1)
   {
   }
   /// text for Main
   public static void Main ()
   {
   }
}
```

See Also

Recommended Tags for Documentation Comments

<paramref>

```
<paramref name="name"/>
```

where:

name
The name of the parameter to refer to. Enclose the name in double quotation marks (" ").

Remarks

The <paramref> tag gives you a way to indicate that a word is a parameter. The XML file can be processed to format this parameter in some distinct way.

Compile with /doc to process documentation comments to a file.

Example

```
// xml_paramref_tag.cs
// compile with: /doc:xml_paramref_tag.xml
/// text for class MyClass
public class MyClass
{
   /// <remarks>MyMethod is a method in the MyClass class.
   /// The <paramref name="Int1"/> parameter takes a number.
   /// </remarks>
   public static void MyMethod(int Int1)
   {
   }

   /// text for Main
   public static void Main ()
   {
   }
}
```

See Also

Recommended Tags for Documentation Comments

<permission>

```
<permission cref="member">description</permission>
```

where:

cref = "*member*"
A reference to a member or field that is available to be called from the current compilation environment. The compiler checks that the given code element exists and translates *member* to the canonical element name in the output XML. *member* must appear within double quotation marks (" ").

description
A description of the access to the member.

Remarks

The <permission> tag lets you document the access of a member. The **System.Security.PermissionSet** lets you specify access to a member.

Compile with /doc to process documentation comments to a file.

Example

```
// xml_permission_tag.cs
// compile with: /doc:xml_permission_tag.xml
using System;
class TestClass
{
   /// <permission cref="System.Security.PermissionSet">Everyone can access this method.</per
mission>
   public static void Test()
   {
   }
   public static void Main()
   {
   }
}
```

See Also

Recommended Tags for Documentation Comments

<remarks>

```
<remarks>description</remarks>
```

where:

description
A description of the member.

Remarks

The <remarks> tag is used to add information about a type, supplementing the information specified with <summary>. This information is displayed in the Object Browser and in the Code Comment Web Report.

Compile with /doc to process documentation comments to a file.

Example

```
// xml_remarks_tag.cs
// compile with: /doc:xml_remarks_tag.xml
/// <summary>
/// You may have some primary information about this class.
/// </summary>
/// <remarks>
/// You may have some additional information about this class.
/// </remarks>
public class MyClass
{
   /// text for Main
   public static void Main ()
   {
   }
}
```

See Also

Recommended Tags for Documentation Comments

<returns>

```
<returns>description</returns>
```

where:

description
A description of the return value.

Remarks

The <returns> tag should be used in the comment for a method declaration to describe the return value.

Compile with /doc to process documentation comments to a file.

Example

```
// xml_returns_tag.cs
// compile with: /doc:xml_returns_tag.xml

/// text for class MyClass
public class MyClass
{
   /// <returns>Returns zero.</returns>
   public static int GetZero()
   {
      return 0;
   }

   /// text for Main
   public static void Main ()
   {
   }
}
```

See Also

Recommended Tags for Documentation Comments

<see>

```
<see cref="member"/>
```

where:

cref = "*member*"
A reference to a member or field that is available to be called from the current compilation environment. The compiler checks that the given code element exists and passes *member* to the element name in the output XML. *member* must appear within double quotation marks (" ").

Remarks

The <see> tag lets you specify a link from within text. Use <seealso> to indicate text that you might want to appear in a See Also section.

Compile with /doc to process documentation comments to a file.

Example

See <summary> for an example of using <see>.

See Also

Recommended Tags for Documentation Comments

<seealso>

```
<seealso cref="member"/>
```

where:

cref = "*member*"
A reference to a member or field that is available to be called from the current compilation environment. The compiler checks that the given code element exists and passes *member* to the element name in the output XML. *member* must appear within double quotation marks (" ").

Remarks

The <seealso> tag lets you specify the text that you might want to appear in a See Also section. Use <see> to specify a link from within text.

Compile with /doc to process documentation comments to a file.

Example

See <summary> for an example of using <seealso>.

See Also

Recommended Tags for Documentation Comments

<summary>

```
<summary>description</summary>
```

where:

description
A summary of the object.

Remarks

The <summary> tag should be used to describe a type or a type member. Use <remarks> to add supplemental information to a type description.

The text for the <summary> tag is the only source of information about the type in IntelliSense, and is also displayed in the Object Browser and in the Code Comment Web Report.

Compile with /doc to process documentation comments to a file.

Example

```
// xml_summary_tag.cs
// compile with: /doc:xml_summary_tag.xml

/// text for class MyClass
public class MyClass
{
   /// <summary>MyMethod is a method in the MyClass class.
   /// <para>Here's how you could make a second paragraph in a description. <see cref="System
.Console.WriteLine"/> for information about output statements.</para>
   /// <seealso cref="MyClass.Main"/>
   /// </summary>
   public static void MyMethod(int Int1)
   {
   }

   /// text for Main
   public static void Main ()
   {
   }
}
```

See Also

Recommended Tags for Documentation Comments

<value>

```
<value>property-description</value>
```

where:

property-description
A description for the property.

Remarks

The <value> tag lets you describe a property. Note that when you add a property via code wizard in the Visual Studio .NET development environment, it will add a <summary> tag for the new property. You should then manually add a <value> tag to describe the value that the property represents.

Compile with /doc to process documentation comments to a file.

Example

```
// xml_value_tag.cs
// compile with: /doc:xml_value_tag.xml
using System;
/// text for class Employee
public class Employee
{
   private string name;
   /// <value>Name accesses the value of the name data member</value>
   public string Name
   {
      get
      {
         return name;
      }
      set
      {
         name = value;
      }
   }
}

/// text for class MainClass
public class MainClass
{
   /// text for Main
   public static void Main()
   {
   }
}
```

See Also

Recommended Tags for Documentation Comments

Processing the XML File

The compiler generates an ID string for each construct in your code that is tagged to generate documentation. (For information on how to tag your code, see Recommended Tags for Documentation Comments.) The ID string uniquely identifies the construct. Programs that process the XML file can use the ID string to identify the corresponding .NET Framework metadata/reflection item that the documentation applies to.

The XML file is not a hierarchical representation of your code, it is a flat list with a generated ID for each element.

The compiler observes the following rules when it generates the ID strings:

- No whitespace is placed in the string.
- The first part of the ID string identifies the kind of member being identified, via a single character followed by a colon. The following member types are used:

Character	Description
N	namespace You cannot add documentation comments to a namespace, but you can make cref references to them, where supported.
T	type: class, interface, struct, enum, delegate
F	field
P	property (including indexers or other indexed properties)
M	method (including such special methods as constructors, operators, and so forth)
E	event
!	error string The rest of the string provides information about the error. The C# compiler generates error informati on for links that cannot be resolved.

- The second part of the string is the fully qualified name of the item, starting at the root of the namespace. The name of the item, its enclosing type(s), and namespace are separated by periods. If the name of the item itself has periods, they are replaced by the hash-sign ('#'). It is assumed that no item has an hash-sign directly in its name. For example, the fully qualified name of the String constructor would be "System.String.#ctor".
- For properties and methods, if there are arguments to the method, the argument list enclosed in parentheses follows. If there are no arguments, no parentheses are present. The arguments are separated by commas. The encoding of each argument follows directly how it is encoded in a .NET Framework signature:
    - Base types. Regular types (ELEMENT_TYPE_CLASS or ELEMENT_TYPE_VALUETYPE) are represented as the fully qualified name of the type.
    - Intrinsic types (for example, ELEMENT_TYPE_I4, ELEMENT_TYPE_OBJECT, ELEMENT_TYPE_STRING, ELEMENT_TYPE_TYPEDBYREF. and ELEMENT_TYPE_VOID) are represented as the fully qualified name of the corresponding full type. For example, System.Int32 or System.TypedReference.
    - ELEMENT_TYPE_PTR is represented as a '*' following the modified type.
    - ELEMENT_TYPE_BYREF is represented as a '@' following the modified type.
    - ELEMENT_TYPE_PINNED is represented as a '^' following the modified type. The C# compiler never generates this.
    - ELEMENT_TYPE_CMOD_REQ is represented as a '|' and the fully qualified name of the modifier class, following the modified type. The C# compiler never generates this.
    - ELEMENT_TYPE_CMOD_OPT is represented as a '!' and the fully qualified name of the modifier class, following the modified type.
    - ELEMENT_TYPE_SZARRAY is represented as "[]" following the element type of the array.
    - ELEMENT_TYPE_GENERICARRAY is represented as "[?]" following the element type of the array. The C# compiler never generates this.
    - ELEMENT_TYPE_ARRAY is represented as [*lowerbound*:*size*,*lowerbound*:*size*] where the number of commas is the rank - 1, and the lower bounds and size of each dimension, if known, are represented in decimal. If a lower bound or size is not specified, it is simply omitted. If the lower bound and size for a particular dimension are omitted, the ':' is omitted as well. For example, a 2-dimensional array with 1 as the lower bounds and unspecified sizes is [1:,1:].
    - ELEMENT_TYPE_FNPTR is represented as "=FUNC:*type*(*signature*)", where *type* is the return type, and *signature* is the arguments of the method. If there are no arguments, the parentheses are omitted. The C# compiler never generates this.

The following signature components are not represented because they are never used for differentiating overloaded methods:

- calling convention
- return type
- ELEMENT_TYPE_SENTINEL

- For conversion operators only (op_Implicit and op_Explicit), the return value of the method is encoded as a '~' followed by the return type, as encoded above.

Examples

The following examples show how the ID strings for a class and its members would be generated:

```
///
///
namespace N  // "N:N"
{
   ///
   ///
   public unsafe class X    // "T:N.X"
   {
      /// <summary>
      ///
      /// </summary>
      public X(){}   // "M:N.X.#ctor"
      // public X(){} // "M:N.X.#cctor", a class constructor
      /// <summary>
      ///
      /// </summary>
      /// <param name="i"></param>
      public X(int i){}  // "M:N.X.#ctor(System.Int32)"
      /// <summary>
      ///
      /// </summary>
      ~X(){}  // "M:N.X.Finalize", destructor's representation in metadata

      /// <summary>
      ///
      /// </summary>
      public string q;  // "F:N.X.q"
      /// <summary>
      ///
      /// </summary>
      /// <returns></returns>
      public const double PI = 3.14;  // "F:N.X.PI"
      /// <summary>
      ///
      /// </summary>
      /// <param name="s"></param>
      /// <param name="y"></param>
      /// <param name="z"></param>
      /// <returns></returns>
      public int f(){return 1;}  // "M:N.X.f"
      /// <summary>
      ///
      /// </summary>
      /// <param name="array1"></param>
      /// <param name="array"></param>
      /// <returns></returns>
      public int bb(string s, ref int y, void * z){return 1;}
      // "M:N.X.bb(System.String,System.Int32@,=System.Void*)"
      /// <summary>
      ///
      /// </summary>
      /// <param name="x"></param>
      /// <param name="xx"></param>
```

```
        /// <returns></returns>
        public int gg(short[] array1, int[,] array){return 0;}
        // "M:N.X.gg(System.Int16[], System.Int32[0:,0:])"
        /// <summary>
        ///
        /// </summary>
        public static X operator+(X x, X xx){return x;} // "M:N.X.op_Addition(N.X,N.X)"
        /// <summary>
        ///
        /// </summary>
        public int prop {get{return 1;} set{}} // "P:N.X.prop"
        /// <summary>
        ///
        /// </summary>
        ///
        public event D d; // "E:N.X.d"
        /// <summary>
        ///
        /// </summary>
        public int this[string s]{get{return 1;}} // "P:N.X.Item(System.String)"
        /// <summary>
        ///
        /// </summary>
        public class Nested{} // "T:N.X.Nested"
        /// <summary>
        ///
        /// </summary>
        public delegate void D(int i); // "T:N.X.D"
        /// <summary>
        ///
        /// </summary>
        /// <param name="x"></param>
        /// <returns></returns>
        public static explicit operator int(X x){return 1;}
        // "M:N.X.op_Explicit(N.X)~System.Int32"
    }
}
```

See Also

/doc (Process Documentation Comments) | XML Documentation

Delimiters for Documentation Tags

The use of documentation tags requires delimiters, which indicate to the compiler where a documentation comment begins and ends. You can use the following kinds of delimiters with the XML documentation tags:

`///`
This is the form that is shown in documentation examples and used by the Visual C# project templates.

> **Note** The Visual Studio IDE has a feature called Smart Comment Editing that automatically inserts the <summary> and </summary> tags and moves your cursor within these tags after you type the `///` delimiter in the Code Editor. Access this feature from the Formatting, C#, Text Editor, Options Dialog Box in your project property pages.

`/** */`
Added after the Visual Studio .NET release. These are multiline delimiters.

There are some formatting rules when using the `/** */` delimiters:

- For the line that contains the `/**` delimiter, if the remainder of the line is white space, the line is not processed for comments. If the first character is white space, that white space character is ignored and the rest of the line is processed. Otherwise, the entire text of the line after the `/**` delimiter is processed as part of the comment.
- For the line that contains the `*/` delimiter, if there is only white space up to the `*/` delimiter, that line is ignored. Otherwise, the text on the line up to the `*/` delimiter is processed as part of the comment, subject to the pattern-matching rules described in the following bullet.
- For the lines after the one that begins with the `/**` delimiter, the compiler looks for a common pattern at the beginning of each line that consists of optional white space and an asterisk (`*`), followed by more optional white space. If the compiler finds a common set of characters at the beginning of each line, it will ignore that pattern for all lines after the `/**` delimiter, up to and possibly including the line that contains the `*/` delimiter.

Some examples:

- The only part of the following comment that will be processed is the line that begins with `<summary>`. The following two tag formats will produce the same comments:

```
/**
<summary>text</summary>
*/
```

```
/** <summary>text</summary> */
```

- The compiler applies a pattern of " * " to ignore at the beginning of the second and third lines.

```
/**
 * <summary>
 *  text </summary>*/
```

- The compiler finds no pattern in this comment because there is no asterisk on the second line. Therefore, all text on the second and third lines, up till the `*/`, will be processed as part of the comment.

```
/**
 * <summary>
   text </summary>*/
```

- The compiler finds no pattern in this comment for two reasons. First, there is no line that begins with a consistent number of spaces before the asterisk. Second, the fifth line begins with a tab, which does not match spaces. Therefore, all text from the second line until the `*/` will be processed as part of the comment.

```
/**
  * <summary>
  * text
```

```
 *  text2
   *  </summary>
*/
```

See Also

/doc (Process Documentation Comments) | XML Documentation

.NET Framework Reference Information

This topic links you to information on working with the .NET Framework class library.

In the Visual Basic and Visual C# Documentation

Introduction to the .NET Framework Class Library in Visual Studio
Provides links to Visual Studio topics that support the development of features based on classes in the .NET Framework.

In the .NET Framework Documentation

Reference (.NET Framework)
Contains a complete set of reference topics for the .NET Framework, including the class library and ASP.NET syntax.
Class Library (.NET Framework)
Contains a complete set of reference topics for the .NET Framework.
Introduction to the .NET Framework Class Library
Introduces the classes, interfaces, and value types that help expedite and optimize the development process and provide access to system functionality.
Inside the .NET Framework
Contains description of key features of the .NET Framework, including the common language runtime.
Samples (.NET Framework)
Contains a list of sample programs that demonstrate how to write applications and components that use the .NET Framework technologies.

Upgrading Reference for Visual Basic and Visual C#

This section links you to topics on upgrading your applications from other languages to Visual Basic .NET and Visual C# .NET.

In This Section

Visual Basic 6.0 Upgrading Reference
Contains topics about errors, warnings, and issues generated while upgrading a Visual Basic 6.0 project to the new version of Visual Basic.

Java Language Conversion Assistant
Describes the tool for converting Visual J++ projects to Visual C#. Includes diagnostic messages for the conversion process.

Related Sections

Reference (Visual Basic and Visual C#)
Links you to reference information on various aspects of Visual Basic and Visual C# programming.

Visual Basic 6.0 Upgrading Reference

The topics in the following section of the Table of Contents relate to errors, warnings, and issues generated while upgrading a Visual Basic 6.0 project to Visual Basic .NET. These topics should only be accessed from links in the upgraded project in order to provide the proper context.

See Also

Upgrading Applications Created in Previous Versions of Visual Basic

Upgrade Wizard

The Upgrade Wizard is used to upgrade Visual Basic 6.0 projects to Visual Basic .NET. The wizard creates a new project, copies each file from the original project into the new project and modifies it as necessary, and generates a report detailing what was done and what you need to do to finish the upgrade.

To access the Upgrade Wizard, open any Visual Basic 6.0 project file (.vbp) in Visual Basic .NET.

The following options are available in the Upgrade Wizard:

What type of project would you like this to be upgraded to?
If the Visual Basic 6.0 project was an ActiveX EXE or ActiveX Document EXE project, you can choose to upgrade it to either an .exe or a .dll file. For all other project types, the correct option is already selected and cannot be changed.

Where do you want your new project created?
Specifies the location of the new project. By default, the project will be placed in a directory directly beneath the Visual Basic 6.0 project. The new project will be named *ProjectName*.NET, where *ProjectName* is the name of the project being upgraded.

See Also

Upgrading Applications Created in Previous Versions of Visual Basic | Visual Basic Upgrade Wizard

A string cannot be used to index a control collection

In Visual Basic 6.0 the variable representing the name of a control could be used as the index value in the controls collection, as follows:

```
Form1.Controls (stringVariable)
```

In Visual Basic .NET, the index must be numeric.

What to do next

- Replace the control name with its index value from the **Controls** collection. The index value indicates the order in which the controls were added to the collection, with the first control being 0.

See Also

Controls Collection Changes in Visual Basic .NET

Add a delegate for AddressOf operator

In Visual Basic 6.0, the **AddressOf** operator was used to pass the address in memory of a function to an API procedure that takes a function pointer as an argument.

In Visual Basic .NET, the **AddressOf** operator must be used with a **Delegate** type. Delegates allow Visual Basic .NET to maintain a pointer to a function even if the function itself has already been garbage-collected.

The following example illustrates how a callback function that uses **AddressOf** is upgraded:

```
' Visual Basic 6.0
Declare Function SetWindowLong Lib "USER32.DLL" Alias "SetWindowLongA" (ByVal hwnd As Long, B
yVal attr As Long, ByVal lVal As Long) As Long
Sub SubClassWindow(ByVal hwnd As Long)
   If PrevProcPtr = 0 Then
      PrevProcPtr = SetWindowLong(hwnd, GWL_WNDPROC, AddressOf SubClassProc)
   End If
End Sub

' After upgrade to Visual Basic .NET
Declare Function SetWindowLong Lib "USER32.DLL"  Alias "SetWindowLongA"(ByVal hwnd As Integer
, ByVal attr As Integer, ByVal lVal As Integer) As Integer
Sub SubClassWindow(ByVal hwnd As Integer)
   If PrevProcPtr = 0 Then
      ' UPGRADE_WARNING: Add a delegate for AddressOf SubClassProc.
      PrevProcPtr = SetWindowLong(hwnd, GWL_WNDPROC, AddressOf SubClassProc)
   End If
End Sub
```

What to do next

- Add a delegate for the **AddressOf** operator, and change the parameter of the function declaration to the **Delegate** type:

```
Delegate Function SubClassProcDelegate(ByVal hwnd As Integer, ByVal msg As Integer, ByVal
wParam As Integer, ByVal lParam As Integer) As Integer
Declare Function SetWindowLong Lib "USER32.DLL" Alias "SetWindowLongA" (ByVal hwnd As Int
eger, ByVal attr As Integer, ByVal lVal As SubClassProcDelegate) As Integer
Sub SubClassWindow(ByVal hwnd As Integer)
   If PrevProcPtr = 0 Then
      PrevProcPtr = SetWindowLong(hwnd, GWL_WNDPROC, AddressOf SubClassProc)
   End If
End Sub
```

See Also

Delegates and the AddressOf Operator | AddressOf Operator | Delegate Statement

Application will terminate when Sub Main() finishes

In Visual Basic 6.0, an application would terminate only after every object that has been created was destroyed. Many applications set the startup object to **Sub_Main** in order to execute initialization code.

In Visual Basic .NET, the application terminates when the startup object is destroyed. If **Sub_Main** is set as the startup object, the application will terminate immediately after running the code in **Sub_Main**.

What to do next

- Set a form to be the startup object.
- Either move the code from **Sub_Main** to the form's **Load** event, or call **Sub_Main** from the **Load** event.

See Also

Choosing the Startup Form in Your Windows Application

Array may need to have individual elements initialized

In Visual Basic 6.0, all members of a user-defined type (UDT) were initialized when they were declared inside the **Type** statement block.

In Visual Basic .NET, user-defined types are replaced by structures. Members that require parameterless constructors (such as fixed-length strings, fixed-size arrays, and variables declared with the **New** keyword) require initialization. During upgrade, an Initialize() subroutine is added to the structure. Calls to the Initialize subroutine must be added whenever the structure is declared.

When an array of structures is declared or redimensioned, the Initialize() subroutine has to be called for each element of the array, or some members of the structures will not be initialized as they were in Visual Basic 6.0.

What to do next

- Add a call to the Initialize() subroutine wherever the array is declared or redimensioned:

```
' After upgrade to Visual Basic .NET
Private Structure MyStruct
   Dim MyClass1 As Class1

   ' UPGRADE_TODO: "Initialize" must be called to initialize instances of this structure.
   Public Sub Initialize()
      MyClass1 = New Class1
   End Sub
End Structure

Private Structure StructArray
   Dim MyArray As MyStruct

   ' UPGRADE_TODO: "Initialize" must be called to initialize instances of this structure.
   Public Sub Initialize()
      MyArray.Initialize
   End Sub
End Structure

Private Sub Form1_Load(ByVal eventSender As System.Object, ByVal eventArgs As System.Even
tArgs) Handles MyBase.Load
   ' UPGRADE_WARNING: Array MyStructArray may need to have individual elements initialize
d.
   Dim MyStructArray(1) as StructArray
   MyStructArray(0).MyArray.Initialize()
   MyStructArray(1).MyArray.Initialize()
End Sub
```

See Also

Initialize must be called to initialize instances of this structure

Array variables cannot be declared with New

In Visual Basic 6.0, array variables could be declared with the **New** keyword; this would automatically initialize all members of the array.

In Visual Basic .NET the **New** keyword is no longer supported for arrays; members within an array must be explicitly initialized.

The following example shows an array declared with the **New** keyword:

```
' Visual Basic 6.0
Dim MyArray(1) As New Class1
MyArray(0).Text = "Hello"
MyArray(1).Text = "world"

' After upgrade to Visual Basic .NET
' UPGRADE_WARNING: Arrays can't be declared with New.
Dim MyArray(1) As New Class1
MyArray(0).Text = "Hello"
MyArray(1).Text = "world"
```

What to do next

- Remove the **New** keyword.
- Initialize all members of the array when the array is declared:

  ```
  ' Modified code
  Dim MyArray(1) As Class1
  MyArray(0) = New Class1
  MyArray(1) = New Class1
  MyArray(0).Text = "Hello"
  MyArray(1).Text = "world"
  ```

- Alternatively, you can initialize the members of the array when they are used; however, if you do so, you should test to ensure that they are not already initialized:

  ```
  If IsNothing(MyArray(1)) Then
     MyArray(1) = New Class1
  End If
  MyArray(1).Text = "world"
  ```

See Also

Array Size Declaration Changes in Visual Basic

Arrays in a structure may need to be initialized

In Visual Basic .NET, fixed-size arrays in structures are not supported. Fixed-size arrays that are members of structures defined in COM interfaces will need to be initialized before they can be used.

The following example illustrates a structure in a COM object named TypeLibTest that contains a fixed-size array:

```
' Visual Basic 6.0
Dim xx As tlStructArrays
xx.m_tlUSHORT(3) = 55

' After upgrade to Visual Basic .NET
' UPGRADE_WARNING: Arrays in structure xx may need to be initialized before they can be used.

Dim xx As TypeLibTest.tlStructArrays
xx.m_tlUSHORT(3) = System.Convert.ToUInt16(55)
```

What to do next

- Add a line of code to initialize the array:

```
Dim xx As TypeLibTest.tlStructArrays
xx.m_tlUSHORT = New Uint16(4) {}
xx.m_tlUSHORT(3) = System.Convert.ToUInt16(55)
```

See Also

Array Size Declaration Changes in Visual Basic

As clause removed from variant ReDim

In Visual Basic 6.0, a variable of the **Variant** data type could be changed to an array of a different data type using the **ReDim** keyword with an **As** clause.

When a Visual Basic 6.0 project is upgraded to Visual Basic .NET, any **Variant** variables are converted to the **Object** data type; you cannot define an object as a different data type using the **As** clause.

The following example shows the usage of the **ReDim** keyword:

```
' Visual Basic 6.0
Dim V As Variant
ReDim V(10) As Integer

' After upgrade to Visual Basic .NET
Dim V As Object
' UPGRADE_WARNING: As Integer was removed from ReDim V statement.
ReDim V(10)
```

What to do next

- Modify your code so that the variable is initially declared as an array of the correct data type:

  ```
  ' Modified code
  Dim V() As Integer
  ReDim V(10)
  ```

- Search your code to see if the variable is used anywhere else, and make sure that the data type is correct for each occurrence. Assign new variables if necessary for any additional data types.

See Also

ReDim Statement Changes in Visual Basic

Assignment not supported

In Visual Basic 6.0, an integer or variant value could be assigned to the **KeyAscii** parameter of a **KeyPress** event in order to override the **KeyAscii** value passed to the event. The most common usage was to assign a value of 0 to cancel the keystroke.

In Visual Basic .NET, you cannot assign any value other than 0 to **KeyAscii**. The following example shows the effect of upgrading the **KeyPress** event:

```
' Visual Basic 6.0
Private Sub Text1_KeyPress(KeyAscii As Integer)
   Dim MyValue As Variant
   KeyAscii = MyValue
   KeyAscii = 0
   KeyAscii = 1
End Sub

' After upgrade to Visual Basic .NET
Private Sub Text1_KeyPress(ByVal eventSender As System.Object, ByVal eventArgs As System.Wind
ows.Forms.KeyPressEventArgs)
   Dim KeyAscii As Short = Asc(eventArgs.KeyChar)
   ' UPGRADE_WARNING: Assignment not supported: KeyAscii to a non-zero value.
   KeyAscii = MyValue
   KeyAscii = 0
   ' UPGRADE_WARNING: Assignment not supported: KeyAscii to a non-zero value.
   KeyAscii = 1
   If KeyAscii = 0 Then
      eventArgs.Handled = True
   End If
End Sub
```

What to do next

- You cannot assign key values in the **KeyPress** event. Instead, create your own function to assign the key value outside of the **KeyPress** event.

See Also

Control.KeyPress Event

Behavior of Null has changed

In Visual Basic 6.0, the **Null** keyword indicated that a variable contained no valid data, and the **IsNull** function was used to test for **Null**. In Visual Basic .NET, the **Null** keyword is still a reserved word, but it has no syntactical value; the **IsNull** function is no longer supported. In addition, Visual Basic 6.0 supported **Null** propagation — when **Null** was used in an expression, the result of the expression would also be **Null**. **Null** propagation is no longer supported in Visual Basic .NET.

During upgrade, **Null** is converted to **DBNull**, and **IsNull** is converted to **IsDBNull**. The behavior of **DBNull** is slightly different than that of **Null**. **Null** could be used in functions and assignments; **DBNull** cannot.

Where **Null** was used with a **Variant** data type, the **Variant** is converted to **Object** during upgrade; it may be more appropriate to use the **Nothing** keyword or the **IsNothing** function.

What to do next

- Review the use of **Null** in your code. In some cases, functions that worked with **Null** will cause an exception with **DBNull**; you will need to test for **DBNull** in these cases:

```
' Visual Basic 6.0
' Assumes that rs!Field1 is null.
x = Left(rs!Field1, 1)

' After upgrade to Visual Basic .NET
x = VB.Left(rs.Fields("Field1").Value, 1)

' Modified code
If Not IsDbNull rs.Fields("Field1").Value Then
   x = VB.Left(rs.Fields("Field1").Value, 1)
End If
```

  In other cases, code can be simplified:

```
' Visual Basic 6.0 code
If Not IsNull (rs!Field1, 1) Then
   x = rs!Field1
End If

' After upgrade to Visual Basic .NET
x = VB.Left(rs.Fields("Field1").Value, 1)

' Modified code.
' UPGRADE_WARNING: Use of Null/IsNull() detected.
If Not IsDbNull(rs.Fields("Field1").Value) Then
   x = rs.Field1("Field1").Value
End If

' Modified code - If statement no longer needed.
x = rs.Field1("Field1").Value & " "
```

See Also

IsDbNull Function | DBNull Class

Cannot resolve the name of a control

The index of an **HScrollBar** or **VScrollBar** control in a control array could not be resolved because it is not a constant value. The actual index of the control is required in order to resolve the control in the **Change** or **Scroll** events.

What to do next

- Replace the variable with the actual index of the control if possible.
- If the index is not known at design time, remove the controls from the control array and add code to handle the **Change** and **Scroll** events for each individual control.

See Also

Introduction to the Windows Forms HScrollBar and VScrollBar Controls

Cannot set a custom MousePointer

In Visual Basic 6.0, you could specify a custom mouse pointer by assigning an image to the **MouseIcon** property of a form or control and setting the **MousePointer** property to **vbCustom** or **99**. Custom mouse pointers are not supported at design time in Visual Basic .NET; the **MousePointer** property is replaced by the **Cursor** property, and the **MouseIcon** property no longer exists.

> **Note** The **Cursor** property can be set at run time to any value of type **System.Windows.Forms.Cursor**.

The following code set a custom mouse pointer in Visual Basic 6.0:

```
' Visual Basic 6.0
PictureBox1.Mousepointer = vbCustom
```

What to do next

- If the custom mouse pointer was set at design time, choose a new mouse pointer by selecting the **Cursor** property in the Properties window for the form or control.
- If the custom mouse pointer was set in code, modify your code to use the **Cursor** property. You can view a list of available cursors in the Properties window by selecting the **Cursor** property for any form or control.

  For example, to use the Hand cursor in a **PictureBox** control, enter the following code:

  ```
  ' Visual Basic .NET
  PictureBox1.Cursor = System.Windows.Forms.Cursors.Hand
  ```

See Also

Control Changes in Visual Basic .NET | UserControl Changes in Visual Basic .NET

Class instancing was changed to Public

In Visual Basic 6.0, instance creation was controlled by the **Instancing** property of a class, which set both the access level (**Public**, **Private**, **Friend**) of a class and the way an object could be created.

In Visual Basic .NET, the **Instancing** property is no longer supported; you control the way that an object can be created by setting the access level of the constructor (**Sub New**). For more information, see Component Instancing Changes in Visual Basic.

There is no direct equivalent for **SingleUse**, **GlobalSingleUse**, or **GlobalMultiUse** instancing in Visual Basic .NET; when the **Instancing** property is set to one of these values, the access level of both the class and the constructor are set to **Public** during upgrade.

What to do next

- Review your code. The behavior may be slightly different than it was in Visual Basic 6.0, and you may need to change the access level of the class and/or the constructor.

See Also

Component Instancing Changes in Visual Basic

Code could not be parsed

In some cases, code that was syntactically incorrect did not cause compilation errors in Visual Basic 6.0. For example, the following code would not cause a compilation error:

```
Private Sub Form_Click()
   ' Note misspelling of DoEvents.
   DoEventss
End Sub
```

Visual Basic 6.0 performed incremental compilation; because the code in the **Form_Click** event would not be compiled until the event was triggered, the syntax error would only be encountered at run time.

In Visual Basic .NET, all code is compiled before the application is run, so syntax errors will always cause a compilation error.

What to do next

- Modify the code to use the correct syntax.

  Note If you run your application in Visual Basic 6.0 prior to upgrading using the **Start With Full Compile** option, you will get compilation errors for any syntax problems.

See Also

Preparing a Visual Basic 6.0 Application for Upgrading

Code was changed from an event to a procedure

During upgrade some events are changed to procedures in the code with an extra parameter and called from a different event. This is usually necessary because the sequence of events in Visual Basic .NET is different than in Visual Basic 6.0.

For example, in Visual Basic 6.0, when the **Scroll** or **Change** event of a **VScrollBar** control was raised, the current scroll value was assigned to the **Value** property. When the **Scroll** or **Change** event is raised in Visual Basic .NET, the control's value has not yet been updated with the new scroll value. During upgrade, the event is changed to a procedure, and the signature of the original subroutine is changed to include a parameter named **NewScrollValue**. When the **Scroll** event is raised, the procedure is called and the value is passed to the **NewScrollValue** parameter.

The following example shows how the code is upgraded.

```
' Visual Basic 6.0
Private Sub VScroll1_Change()
   Debug.Print "change " & VScroll1.Value
End Sub

Private Sub VScroll1_Scroll()
   Debug.Print "scroll " & VScroll1.Value
End Sub
```

```
' After upgrade to Visual Basic .NET
' UPGRADE NOTE: Vscroll1.Change was changed from an event to a procedure.
Private Sub VScroll1_Change(ByVal NewScrollValue As Integer)
   System.Diagnostics.Debug.WriteLine("change " & NewScrollValue)
End Sub

' UPGRADE NOTE: Vscroll1.Scroll was changed from an event to a procedure.
Private Sub VScroll1_Scroll_Renamed(ByVal NewScrollValue As Integer)
   System.Diagnostics.Debug.WriteLine("scroll " & NewScrollValue)
End Sub

Private Sub VScroll1_Scroll(ByVal eventSender As System.Object, _
ByVal eventArgs As System.Windows.Forms.ScrollEventArgs) Handles _
Vscroll1.Scroll
   Select Case eventArgs.type
      Case System.Windows.Forms.ScrollEventType.ThumbTrack
         VScroll1_Scroll_Renamed(eventArgs.newValue)
      Case System.Windows.Forms.ScrollEventType.EndScroll
         VScroll1_Change(eventArgs.newValue)
   End Select
End Sub
```

What to do next

- No further action is necessary — this note was added to make you aware of the changes made to your code.

See Also

Introduction to the Windows Forms HScrollBar and VScrollBar Controls | Event Changes in Visual Basic .NET

Code was upgraded to use a function which may not have the same behavior

During upgrade, some Visual Basic 6.0 functions are upgraded to different Visual Basic .NET functions with different behavior. In many cases, this will not have an effect on your code; however, you should be aware of these differences and review your code to make sure that your usage of these functions is still appropriate.

Assigning a Byte array to a string

In Visual Basic 6.0, you could assign a **Byte** array to a string with direct assignment. When upgraded to Visual Basic .NET, this assignment is replaced by a function in the **System.Text.UnicodeEncoding** namespace. Because it is Unicode based, it expects an even number of bytes; an odd number of bytes will cause an exception.

The following example illustrates how this is upgraded:

```
' Visual Basic 6.0
Dim B() As Byte
Dim S As String
S = B()

' After upgrade to Visual Basic .NET
Dim B() As Byte
Dim S As String
' UPGRADE_TODO: Code was upgraded to use GetString, which may not have the same behavior.
S = System.Text.UnicodeEncoding.Unicode.GetString(B)
```

What to do next

- Review the documentation for the Visual Basic .NET version of the function, and modify your code as necessary. In the above example, you would need to make sure that an even number of bytes are being passed when you assign to B().

See Also

Preparing a Visual Basic 6.0 Application for Upgrading

COM expression not supported

In Visual Basic 6.0, all methods of a COM object were supported whether they were exposed as methods on a class or methods on a module. Most COM objects only expose class methods; however, some objects such as the DirectX library also expose module methods.

In Visual Basic .NET, only the methods on a class are exposed to COM interop; Visual Basic 6.0 code that accesses methods on a module cannot be upgraded.

What to do next

- Review your code and determine how this method was used. In some cases, there may be objects in the .NET Framework that provide the same functionality. If your code absolutely depends on module methods of a COM object, you will need to keep the application in Visual Basic 6.0.

See Also

Preparing a Visual Basic 6.0 Application for Upgrading

Conditional block not upgraded because an expression evaluates to False

Visual Basic 6.0 supported conditional compilation using the **#If**...**#EndIf** statement syntax and conditional compilation constants. During compilation, only the code block that evaluated to **true** would be compiled; other code blocks within the **#If**...**#EndIf** construct would be ignored.

When a project is upgraded to Visual Basic .NET, only those blocks that evaluate to **true** will be upgraded; other code blocks within the **#If**...**#EndIf** construct will not be upgraded.

The following example shows a conditional block:

```
' Visual Basic 6.0
#If True Then
   Dim x As TextBox
#Else
   Dim x As Label
#End If

' After upgrade to Visual Basic .NET
#If True Then
   Dim x As System.Windows.Forms.TextBox
#Else
   ' UPGRADE_NOTE: #If #EndIf block was not upgraded because the expression Else did not eval
uate to True.
   Dim x As Label
#End If
```

What to do next

- Evaluate your project in Visual Basic 6.0 and make sure that the correct conditional compilation constants are being set. If you want to upgrade the code within a conditional block, remove it from the conditional block in Visual Basic 6.0, then add the conditional statements back after upgrading.

See Also

Conditional Compilation Overview

Constant was not upgraded

In some cases, constants used in Visual Basic 6.0 have no equivalent in Visual Basic .NET. During upgrade, constants are converted to their .NET equivalent where possible. If the property, method, or function to which the constant applies was not upgraded, the constant cannot be upgraded. For example, **Drag** constants are not upgraded because drag-and-drop code is not upgraded.

What to do next

- Check your code to determine how the constant is being used. If it applies to a property, method, or function that could not be upgraded, remove the line of code.
- If the context for the constant is still valid, check the value for the constant in your Visual Basic 6.0 application, and replace the constant with the value.

See Also

Constant Changes in Visual Basic .NET

Control was not upgraded

Due to differences between Visual Basic 6.0 and Visual Basic .NET, some controls cannot be upgraded, or some controls can only be upgraded under certain conditions. For example, the **Data** control cannot be upgraded because it depends on DAO, which is not supported in Visual Basic .NET.

What to do next

Select a control from the list below to see a topic listing changes and issues for that control.

Data Control Changes in Visual Basic .NET

Line Control Changes in Visual Basic .NET (diagonal lines only)

OLE Container Control Changes in Visual Basic .NET

RDO Data Control Changes in Visual Basic .NET

Shape Control Changes in Visual Basic .NET (circles or ovals only)

UpDown Control

The **UpDown** control that was included in the Microsoft Windows Common Controls 2 (Mscomct2.ocx) cannot be upgraded. There is no direct equivalent in Visual Basic .NET; however, the **DomainUpDown** control or the **NumericUpDown** control can be used to achieve similar functionality. For more information, see Introduction to the Windows Forms DomainUpDown Control or Introduction to the Windows Forms NumericUpDown Control.

UserControl Changes in Visual Basic .NET

See Also

Things to Consider Before Upgrading

Could not find a project file

The referenced file or files could not be found. The file(s) may have been moved, the media containing the file(s) may have been removed, or a network connection may have failed.

What to do next

- Restore the missing file(s) to the proper directories before upgrading. If the file(s) were not in the same directory as the project file (.vbp), you can open the project file in a text editor to determine the path to the file.

See Also

Upgrading a Project with the Upgrade Wizard

Could not load referenced component

A component referenced in the Visual Basic 6.0 project could not be loaded during the upgrade process. In most cases this occurs because the component does not exist on the upgrade computer, is not registered, or is referenced with an invalid path.

What to do next

Make sure that the project can be successfully compiled and run using Visual Basic 6.0 on the upgrade computer. If the project cannot be run successfully, check the component references in the Visual Basic 6.0 project and correct the path.

Alternatively, you can make sure that the correct version of the component is installed on the target computer in the location specified in the Visual Basic 6.0 .vbp file, then run the upgrade tool again.

See Also

Preparing a Visual Basic 6.0 Application for Upgrading | Working with Both Visual Basic 6.0 and Visual Basic .NET

Data binding for ComboBox or ListBox is read-only

In Visual Basic 6.0, a **ComboBox** or **ListBox** control could be bound to a database using ADO; when a value in the control was changed, the database was updated. In Visual Basic .NET, data binding for the ComboBox Control (Windows Forms) and ListBox Control (Windows Forms) controls is read-only when a user changes the value by selecting an item from the list. When the value in the database changes, the control is updated, but when the value in the control is changed, the database is not updated.

> **Note** For the **ComboBox** control, changing the value by typing in the **TextBox** portion of the control does update the database.

What to do next

Add code to the ListControl.SelectedValueChanged Event event for the **ComboBox** or **ListBox** controls to update the database.

See Also

ComboBox Control Changes in Visual Basic .NET | ListBox Control Changes in Visual Basic .NET | Data Changes in Visual Basic .NET

Declaration changed from Constant to Variable

In Visual Basic 6.0, constants were primitive types. In Visual Basic .NET, some values that were formerly represented as constants are now reference types within the .NET Framework.

For example, in Visual Basic 6.0 color constants such as `vbBlack` were used to refer to colors. In Visual Basic .NET, colors are reference types within the System.Drawing namespace: `System.Drawing.Color.Black`.

The following example shows a constant that is upgraded to a variable:

```
' Visual Basic 6.0
Const ConstColor = vbBlack

' After upgrade to Visual Basic .NET
' UPGRADE_NOTE: ConstColor was changed from a Constant to a Variable.
Dim ConstColor As System.Drawing.Color = System.Drawing.Color.Black
```

What to do next

- No further action is necessary. This upgrade note was added to make you aware of the new reference type that replaces your constant.

See Also

Declaration type not supported | NewEnum property was commented out

Declaration type not supported

Some variable declarations that were allowed in Visual Basic 6.0 are no longer allowed in Visual Basic .NET.

Arrays of fixed-length strings

In Visual Basic 6.0, you could declare an array of fixed-length strings as follows:

```
Dim x(10) As String * 10
```

Arrays of fixed-length strings are not supported in Visual Basic .NET.

What to do next

- Modify the declaration to use variable length strings:

  ```
  Dim x(10) As String
  ```

- If fixed-length strings are necessary, you can manage the length of strings before assigning them to the array using the **Len** function, **Left** function and the **PadRight** method:

  ```
  Public Function FixedLength10(ByVal S As String) As String
     Dim l As Long
     L = Len(s)
     If l > 10 Then
        FixedLength10 = Microsoft.VisualBasic.Left(s, 10)
     ElseIf l < 10 Then
        FixedLength10 = s.PadRight(10)
     End If
  End Function
  ```

Arrays with more than 32 dimensions

In Visual Basic 6.0, the maximum number of dimensions for an array was 60. In Visual Basic .NET, the maximum number of dimensions is 32.

What to do next

- Rethink the design of your application. Arrays with large number of dimensions are very memory-intensive and are likely to produce an out-of-memory error on most computers.

Arrays with a negative upper bounds

In Visual Basic 6.0, arrays could be declared with negative upper bounds. For example, the declaration `Dim MyArray(-15 To -10)` was valid.

What to do next

- In Visual Basic 6.0, change the bounds of arrays so that upper bounds are not negative and modify any code that uses the arrays prior to upgrading.

References to a Private Enum

In Visual Basic 6.0, it was possible to reference a Private Enum from a Public Enum within the same module. In Visual Basic .NET, this is no longer allowed.

What to do next

- Change the Enum to Public, or use the actual value of the Private Enum instead.

Color constants in an Enum

In Visual Basic 6.0, it was possible to define enumerations for colors using the VB Color constants:

```
' Visual Basic 6.0
Enum MyColors
   Red = vbRed
End Enum
```

In Visual Basic .NET, colors are reference types in the System.Drawing namespace; color constants are unnecessary.

What to do next

- Find any code that uses the Enum and replace it with code that sets the color using System.Drawing.Color:

```
' After upgrade to Visual Basic .NET
Me.BackColor = MyColors.Red

' Modified code
Me.BackColor = System.Drawing.Color.Red
```

See Also

Left Function | Len Function | String.PadRight Method | System.Drawing Namespace |
Declaration changed from Constant to Variable

Declare statement does not support parameters of type As Any

In Visual Basic 6.0, you could declare a reference to an external procedure with the **Declare** statement and specify **As Any** for the data type of an argument or return type. The **As Any** keyword disabled type checking and allowed any data type to be passed in or returned.

In Visual Basic .NET, the **As Any** keyword is no longer supported. To ensure type safety, you must specifically declare the data type of all arguments and return values.

For example, the following declaration accepted either a string or a long data type as the second argument:

```
' Visual Basic 6.0
Declare Function LoadCursor Lib "user32" Alias "LoadCursorA"(ByVal hInstance As Long, ByVal l
pCursorName As Any) As Long

' After upgrade to Visual Basic .NET
' UPGRADE_ISSUE: Declaring a parameter 'As Any' is not supported.
Declare Function LoadCursor Lib "user32" Alias "LoadCursorA"(ByVal hInstance As Integer, ByVa
l lpCursorName As Any) As Integer
```

What to do next

- Determine where the declared function is being used in your code and what data type is being passed. If all occurrences pass the same data type, change the declaration to include that data type instead of **As Any**:

  ```
  Declare Function LoadCursor Lib "user32" Alias "LoadCursorA"(ByVal hInstance As Integer,
  ByVal lpCursorName As String) As Integer
  ```

- If you need to support multiple data types, create an overloaded version of the declaration for each data type:

  ```
  Declare Function LoadCursor Lib "user32" Alias "LoadCursorA"(ByVal hInstance As Integer,
  ByVal lpCursorName As String) As Integer
  Declare Function LoadCursor Lib "user32" Alias "LoadCursorA"(ByVal hInstance As Integer,
  ByVal lpCursorName As Integer) As Long
  ```

See Also

Declaration Syntax Changes in Visual Basic

DoEvents no longer returns a value

In previous versions of Visual Basic, the **DoEvents** function returned an **Integer** value representing the number of open forms. In Visual Basic .NET, the **DoEvents** function is replaced by the **DoEvents** method; consequently, it has no return value. Attempting to call **DoEvents** as a function will cause the compiler error "No accessible overloaded method 'DoEvents' produces a value".

> **Note** The more common use of **DoEvents** was to yield execution to allow the system to process messages. In this case it was treated as a method with no return value. Code using **DoEvents** in this manner will migrate successfully to Visual Basic .NET.

```
' Visual Basic 6.0
Dim i As Integer
' Returns the number of open forms.
i = DoEvents
' Yields processing; no return value.
DoEvents

' After upgrade to Visual Basic .NET
Dim i as Short
' Causes a compiler error.
i = System.Windows.Forms.Application.DoEvents()
' Compiles successfully.
System.Windows.Forms.Application.DoEvents()
```

What to do next

- Change the **DoEvents** function to a method as in the above example.
- Review your code to determine how the return value of the **DoEvents** function was being used and modify your code as necessary.

> **Note** There is no equivalent function in Visual Basic .NET that returns the number of open forms.

See Also

Programming Element Support Changes Summary

Event is not supported

Some events that were supported in Visual Basic 6.0 are no longer supported in Visual Basic .NET or do not map directly to a Visual Basic .NET event. In these cases, code in the Visual Basic 6.0 event procedure will be preserved as is; however, the code will not execute in Visual Basic .NET and must be replaced. For example, the **DblClick** event for a **ListBox** control has no equivalent; depending on how it was used you might replace it with either the **SelectedIndexChanged** or **TextChanged** event.

What to do next

Select an object or control from the list below to see a topic listing changes and issues for that object.

CheckBox Control Changes in Visual Basic .NET

ComboBox Control Changes in Visual Basic .NET

CommandButton Control Changes in Visual Basic .NET

DirListBox Control Changes in Visual Basic .NET

DriveListBox Control Changes in Visual Basic .NET

FileListBox Control Changes in Visual Basic .NET

Form Object Changes in Visual Basic .NET

Frame Control Changes in Visual Basic .NET

HScrollBar Control Changes in Visual Basic .NET

Image Control Changes in Visual Basic .NET

Label Control Changes in Visual Basic .NET

ListBox Control Changes in Visual Basic .NET

MDIForm Object Changes in Visual Basic .NET

Menu Object Changes in Visual Basic .NET

OptionButton Control Changes in Visual Basic .NET

PictureBox Control Changes in Visual Basic .NET

TextBox Control Changes in Visual Basic .NET

Timer Control Changes in Visual Basic .NET

UserControl Changes in Visual Basic .NET

VScrollBar Control Changes in Visual Basic .NET

See Also

Event Changes in Visual Basic .NET | Form Event Changes in Visual Basic .NET | Change Event Changes in Visual Basic .NET | TopIndex Property and Scroll Event Changes in Visual Basic .NET

Event may fire when form is initialized

In Visual Basic 6.0, events for a form or control were not raised until the form was finished loading. In Visual Basic .NET, the **InitializeComponent** method handles the initialization of the form and its controls; the order in which objects are initialized cannot be modified. In some cases, this may cause events to be raised during initialization. If the event handler references other components that have not yet been initialized, this could cause a run-time error.

For example, if the **Click** event handler for a **CheckBox** control sets a value in a **TextBox** control, an error may occur if the **TextBox** control is not yet initialized:

```
' Visual Basic 6.0
Private Sub Check1_Click()
   TextBox1.Text = "Check1 was clicked"
End Sub

' After upgrade to Visual Basic .NET
' UPGRADE_WARNING: Event CheckStateChanged may fire when form is initialized.
Private Sub Check1_CheckStateChanged(ByVal sender As System.Object, ByVal e As System.EventAr
gs) Handles Check1.CheckStateChanged
   ' The following line will cause an error if TextBox1 is not
   ' initialized.
   TextBox1.Text = "Check1 was clicked"
End Sub
```

What to do next

1. Add an **IsInitializing** property to the form.
2. Set the property to **true** in the form's constructor just prior to the **InitializeComponent** call; set it to **false** immediately following the call.
3. Add the following logic to the event procedure to check the value of the property before running the code:

```
Private Sub Check1_CheckStateChanged(ByVal sender As System.Object, ByVal e As System.Eve
ntArgs) Handles Check1.CheckStateChanged
   If Form1.DefInstance.IsInitializing = True Then
      Exit Sub
   Else
      TextBox1.Text = "Check1 was clicked"
   End If
End Sub
```

See Also

Property Statement | Forms Changes in Visual Basic .NET

Event parameter was not upgraded

Some event parameters that were supported in Visual Basic 6.0 are no longer supported in Visual Basic .NET.

- The **Cancel** parameter of the **Form_Unload** event is no longer supported. The **Unload** event is upgraded to the **Closed** event; when this event raises, garbage collection has already been initiated and it is too late to cancel.
- The **UnloadMode** parameter of the **Form_QueryUnload** event is no longer supported. The **QueryUnload** event is upgraded to the **Closing** event; this event has no mechanism for determining why it was called.

What to do next

- Move any code that is dependent on the **Cancel** parameter into the **Closing** event.
- If it is necessary to determine why the form is closing, you will need to create an overloaded version of the **Closing** event with custom code to determine the caller.

See Also

Form.Closed Event | Form.Closing Event

Function has a new behavior

Several functions in Visual Basic .NET behave differently than their equivalent Visual Basic 6.0 functions. In most cases, this will not have an effect on your code; however, you should be aware of these differences and review your code to make sure that your usage of these functions is still appropriate.

Array function

In Visual Basic 6.0, the **Array** statement was used to return a variant containing an array, assigning values to array elements in a single step. The **Array** function is no longer supported in Visual Basic .NET.

When a Visual Basic 6.0 project is upgraded to Visual Basic .NET, any **Variant** variables are converted to the **Object** data type, so the upgraded code will return an object containing an array.

What to do next

- Review your code. In most cases, you will probably want to create an array to replace the upgraded object and assign the values as you would with any other array.

Dir function

In Visual Basic 6.0, the **Dir** function was used to return the name of a file, folder, or directory. The "." and ".." strings could be used in the path argument to represent the current directory or the encompassing directory, respectively.

In Visual Basic .NET, the **Dir** function is essentially the same, but the "." and ".." syntax has a different behavior.

What to do next

- Code that returned files or directories was often written with the assumption that the "." and ".." would be returned as the first two entries; this is no longer true in Visual Basic .NET. Because of the difference in behavior, the upgraded code may not return the first two files or directories.

Get#, Put# Functions

In Visual Basic 6.0, the **Get#** and **Put#** functions were used to read and write data to a file.

In Visual Basic .NET, these are upgraded to the **FilePut** and **FileGet** functions. When these functions are used to pass dynamic arrays or strings to a file opened in **Random** mode, a two-byte length descriptor is added, resulting in a different file length.

What to do next

- To achieve the same results as in Visual Basic 6.0, use the optional **ArrayIsDynamic** argument to the **FilePut** and **FileGet** functions.
- When opening a file in Random mode, use the **StringIsFixedLength** argument.

IsObject function

In Visual Basic 6.0, the **IsObject** function was used to determine if a **Variant** variable was of type vbObject. If the variable was empty, the **IsObject** function would return **true** for an object, but **false** for a variant. Likewise, if the variable contained a string, it would return **false**.

When a Visual Basic 6.0 project is upgraded to Visual Basic .NET, any **Variant** variables are converted to the **Object** data type. The Visual Basic .NET **IsObject** function is used to determine if a variable contains an object. If an object is empty, the function will still return **true** because it is an object. In addition, strings in Visual Basic .NET are objects and will return **true**.

What to do next

- Find the variable(s) being evaluated by the **IsObject** function and make sure that the logic is still valid.

MOD operator

In Visual Basic 6.0, the MOD operator accepted any numeric expression and the result was always returned as an integer.

In Visual Basic .NET, the MOD operator no longer accepts variants, and if either operand is a floating-point number, the result will be a floating-point number.

What to do next

- If a variant has been converted to an object during upgrade, convert it to the appropriate numeric data type.
- If either of the operands is a floating-point number, modify the data type of the result, or modify your code to convert the operand(s) using the **CShort** function before using the **MOD** operator.

Return statement

In Visual Basic 6.0, the **Return** statement was used to branch back to the code following a **GoSub** statement.

In Visual Basic .NET, the **GoSub** statement is not supported. The **Return** statement is used to return control to a calling procedure from a **Function** or **Sub** procedure.

What to do next

- Remove any lines of code that use **Return** in conjunction with a **GoSub** statement.

SavePicture statement

In Visual Basic 6.0, the **SavePicture** statement was used to save a graphic from the **Picture** or **Image** property of an object to a file. If the format of the graphic for the **Picture** property was either GIF or JPEG, it was saved as a bitmap file (BMP). Graphics for the **Image** property were always saved as bitmaps regardless of their original format.

In Visual Basic .NET, the default behavior of the **Image.Save** method saves the files in different formats than in Visual Basic 6.0. **Image.Save** takes an **ImageFormat** class as an optional parameter that allows you to specify the format to use.

What to do next

- Modify the upgraded code to use the **ImageFormat** method, as in the following example:

```
' Default, will save as a PNG (Portable Network Graphics) file
Picture1.Image.Save("MyPicture.bmp")
' Using the optional parameter, will save as a bitmap
Picture1.Image.Save("MyPicture.bmp", System.Drawing.Imaging.ImageFormat.Bmp)
```

Str function

In Visual Basic 6.0, the **Str** function took a **Long** data type containing any valid numeric expression as an argument; date variables or literals were treated as valid numeric expressions.

In Visual Basic .NET, the **Str** function takes an object containing any valid numeric expression; date variables and literals are no longer treated as valid numeric expressions.

What to do next

- If the **Str** function uses a date variable or date literal as an argument, use the **Format** function instead. For more information, see Format Function.

TypeName function

In Visual Basic 6.0, the **TypeName** function was used to return a string describing the underlying data type of a variable.

The data types returned by the **TypeName** function may have different values in Visual Basic .NET. For example, any **Variant** variables are converted to the **Object** data type, strings are objects in Visual Basic .NET, and controls return a different type (**TreeView** is now **AXTreeView**). For more information, see TypeName Function.

What to do next

- Compare the result of the **TypeName** function with the result in the Visual Basic 6.0 to determine if they are the same, and modify your code if necessary.

TypeOf function

In Visual Basic 6.0, the **TypeOf** function was used in an **If...Then...Else** statement to determine whether an object reference was of a specified object type. In this context, a user-defined type was considered to be an object type.

In Visual Basic .NET, user-defined types (now known as structures) are not object types and cannot be evaluated by the **TypeOf** function.

The following example demonstrates the use of **TypeOf** with a user-defined type:

```
' Visual Basic 6.0
Private Type MyType
   a As Integer
End Type
Private Sub Form_Load()
   Dim m As MyType
   If TypeOf m Is MyType Then
      MsgBox "MyType"
   End If
End Sub

' After upgrade to Visual Basic .NET
Private Structure MyType
   Dim a As Short
End Structure
Private Sub Form_Load(ByVal eventSender As System.Object, ByVal eventArgs As System.EventArgs
) Handles MyBase.Load
   Dim m As MyType
   ' UPGRADE_WARNING: TypeOf has a new behavior.
   If TypeOf m Is MyType Then
      MsgBox "MyType"
   End If
End Sub
```

What to do next

- Modify your code to do a type comparison without the **TypeOf** function:

```
' Modified code
Dim m As MyType
Dim mTest As MyType
If m.GetType Is mTest.GetType Then...
```

VarType function

In Visual Basic 6.0, the **VarType** function was used to return an enumeration representing the underlying data type of a variable declared as a variant.

When a Visual Basic 6.0 project is upgraded to Visual Basic .NET, any **Variant** variables are converted to the **Object** data type. In Visual Basic 6.0, an unassigned variant returned 0 (Empty); in Visual Basic .NET, an unassigned object returns 9 (object).

What to do next

- Determine if the **VarType** function is returning 9 and, if so, modify your code to correctly interpret the new result.

See Also

Programming Element Support Changes Summary | Language Changes in Visual Basic

Function is not supported

Some Visual Basic 6.0 functions are no longer supported in Visual Basic .NET. The functionality that these functions provided can be accomplished using new functions in the .NET Framework; however, the differences in implementation make it too difficult to upgrade automatically.

The following functions are not upgraded:

CVErr function

The **CVErr** function in Visual Basic 6.0 returned a Variant of the subtype Error that contained an error number. Variants are no longer supported in Visual Basic .NET, so this function is no longer useful.

Error handling in general has changed significantly in Visual Studio .NET. For more information, see Event Handlers in Visual Basic and Visual C#.

IMEStatus function

The **IMEStatus** function in Visual Basic 6.0 was used to return the status of the Input Method Editor (IME) mode, available only on Chinese, Japanese, and Korean versions of Windows.

In Visual Basic .NET, this functionality is replaced by the **IMEMode** property of most controls and by the **IMEMode** enumeration, both in the System.Windows.Forms namespace. For more information, see Control.DefaultImeMode Property or ImeMode Enumeration.

AscB, ChrB, InstrB, LeftB, LenB, MidB, RightB functions

The **AscB**, **ChrB**, **InstrB**, **LeftB**, **LenB**, **MidB**, and **RightB** functions in Visual Basic 6.0 were string-handling functions that returned their results in bytes. They were used primarily for converting strings for use by double-byte character set languages.

In Visual Basic .NET, encoding and decoding functions in the System.Text namespace replace this functionality. For more information, see Globalizing and Localizing Applications.

VarPtr, ObjPtr, StrPtr, VarPtrArray, and VarPtrStringArray functions

The **VarPtr**, **ObjPtr**, **StrPtr**, **VarPtrArray**, and **VarPtrStringArray** functions in Visual Basic 6.0 were undocumented and unsupported functions that could be used to return a pointer to a memory address. Although unsupported, these functions were sometimes used when working with Windows API calls that would have otherwise been inaccessible from Visual Basic.

These functions are not supported in Visual Studio .NET; however, most of the scenarios where they were used can be addressed through new functionality in the .NET Framework. Many of the Windows API calls that were necessary in Visual Basic 6.0 are now encapsulated in the framework; pointer references are no longer necessary.

What to do next

- Review your code to determine what was being accomplished with this function, search the .NET Framework documentation for suitable replacements, and modify your code as necessary.

See Also

What's New in Visual Basic and Visual C# | Programming with the .NET Framework

Function parameter is not supported in Visual Basic .NET

Some function signatures are different in Visual Basic .NET than they were in Visual Basic 6.0, and some parameters are no longer supported.

In Visual Basic 6.0 the **MsgBox** and **InputBox** functions supported optional arguments to specify a Help topic that would be displayed when the user pressed F1. Because the mechanism for displaying Help topics has changed significantly in Visual Basic .NET, the **HelpFile** and **Context** arguments have been eliminated.

> **Note** Visual Basic .NET does not support calling Help from a message box or input box; you can duplicate this functionality by using a modal form instead and adding a Help button.

In Visual Basic 6.0, the **AppActivate** statements supported an optional **Wait** parameter that specified whether a specified application should be activated immediately, or wait until the calling application has focus. There is no equivalent functionality in Visual Basic .NET; the specified application is always activated immediately.

What to do next

- If Help support is required, replace the message box or input box with a modal form.
- Review your application's run-time behavior to make sure that removing the **Wait** parameter does not cause unexpected behavior. If necessary, modify your code so that **AppActivate** is only called when your application has focus.

See Also

Providing Help in a Windows Application

Initialize must be called to initialize instances of this structure

In Visual Basic 6.0, all members of a user-defined type (UDT) were initialized when they were declared inside the **Type** statement block.

In Visual Basic .NET, user-defined types are replaced by structures. Members that require parameterless constructors (such as fixed-length strings, fixed-size arrays, and variables declared with the **New** keyword) require initialization. During upgrade, an Initialize() subroutine is added to the structure. Calls to the Initialize subroutine must be added whenever the structure is declared.

The following code illustrates the upgrade of code that contains a fixed-size array in a user-defined type:

```
' Visual Basic 6.0
Public Type DriveInfo
   DriveNumber(25) As Integer
   DriveType As String
End Type

Function AddDrive(DriveNumber As Integer, DriveLabel As String)
   Dim Drives As DriveInfo
   Drives.DriveNumber(0) = 123
   Drives.DriveType = "Fixed"
End Function

' After upgrade to Visual Basic .NET
Public Structure DriveInfo
   ' Note that the fixed size was removed.
   Dim DriveNumber() As Short
   Dim DriveType As String
   ' UPGRADE_TODO: "Initialize" must be called to initialize instances of this structure.
   Public Sub Initialize()
      ReDim DriveNumber(25)
   End Sub
End Structure

Function AddDrive(ByRef Number As Short, ByRef DriveLabel As String) As Object
   Dim Drives As DriveInfo
   ' This will fail; the array has not been initialized.
   Drives.DriveNumber(0) = 123
   Drives.DriveType = "Fixed"
End Function
```

What to do next

- Find each occurrence in your code where an instance of the user-defined type is declared.
- Add a call to the Initialize() subroutine. In the example above, the **AddDrive** function would look like the following.

```
Function AddDrive(ByRef Number As Short, ByRef DriveLabel As String) As Object
   Dim Drives As DriveInfo
   Drives.Initialize()
   Drives.DriveNumber(0) = 123
   Drives.DriveType = "Fixed"
End Function
```

See Also

Array Size Declaration Changes in Visual Basic

IsMissing function was changed to IsNothing

In previous versions of Visual Basic the **IsMissing** function was used to test for optional arguments passed to user-defined function procedures as variants. Default values were not required for optional arguments; if no value was passed then **IsMissing** returned **true**. It was also possible to specify a default value of a primitive type to pass to the **Variant** argument.

In Visual Basic .NET, the **IsMissing** function has been removed from the language, and default values are now required for all optional arguments. During upgrade, variants are replaced with objects, and an object's default value of **Nothing** is added to all optional arguments.

The following code illustrates the upgrade of code that uses the **IsMissing** function:

```
' Visual Basic 6.0
Function DoThis(Optional X As Variant)
   If IsMissing(X) Then
      ' Do something here.
   End If
End Function

' After upgrade to Visual Basic .NET
Function DoThis(Optional ByRef X As Object = Nothing) As Object
   ' UPGRADE_WARNING: IsMissing function was changed to IsNothing.
   If IsNothing(X) Then
      ' Do something here.
   End If
End Function
```

In most cases this code should execute properly; however, in some cases it may not give the expected result. If X in the above example is an object being passed as a variant and the object has been set to **Nothing**, in Visual Basic 6.0 **IsMissing** would have returned **false**. In Visual Basic .NET, the value of X would be **Nothing** and **IsNothing** would return **true**.

What to do next

- Search for any code that calls this function. Step through the function call and inspect the value of the optional arguments that are being passed.
- If a value is being passed and **IsNothing** returns **true**, modify your code to add an additional test for **IsNothing** in the calling procedure:

```
' Visual Basic 6.0
Dim X As Object
' Do something with the object.
Set X = Nothing
' IsMissing in the DoThis function returns false.
DoThis(X)

' Visual Basic .NET
Dim X As Object
' Do something with the object.
X = Nothing
If Not IsNothing(X) Then
   DoThis(X)
End If
```

See Also

Procedure Declaration Changes in Visual Basic

ItemData property cannot be upgraded

In Visual Basic 6.0 the **ItemData** property for a **ListBox** or **ComboBox** control could be set at design time in the Properties window. In Visual Basic .NET the **ItemData** property no longer exists; list items are of type **Object** rather than **String** and can contain any type of data.

When a Visual Basic 6.0 project is upgraded, any values in the **ItemData** property are discarded.

What to do next

1. In your Visual Basic 6.0 project, check the **List** and **ItemData** properties for the **ListBox** or **ComboBox** controls in the Properties window to see the values that were entered at design time.
2. In the upgraded project, add code to the Sub New() procedure to assign the **ItemData** property at run time using the **SetItemData** function from the Visual Basic Compatibility library.

```
' Visual Basic .NET
' In the original project, values of 123, 456, and 789 were set at     ' design time for
the first three items in a ListBox.
VB6.SetItemData(List1, 0, 123)
VB6.SetItemData(List1, 1, 456)
VB6.SetItemData(List1, 2, 789)
```

See Also

ComboBox Control Changes in Visual Basic .NET | ListBox Control Changes in Visual Basic .NET

Late-bound default property reference could not be resolved

In Visual Basic 6.0, objects had a default property that could be used as a shortcut in code. For example, the default property of a **TextBox** control was the **Text** property; you could type `TextBox1 = "Hello"` instead of `TextBox1.Text = "Hello"`.

In Visual Basic .NET, default properties are no longer supported; all property references must be fully qualified. During upgrade, default properties for early-bound objects are resolved to the property name; however, late-bound objects cannot be resolved because it is impossible to determine a default property without knowing what the object is.

In addition, Visual Basic 6.0 allowed late-bound calls to implemented interfaces of classes; this is no longer allowed in Visual Basic .NET.

The following example shows late-bound code that cannot be resolved:

```
' Visual Basic 6.0
   Dim x As String
   Dim y As Object
   x = "Hello"
   Set y = Form1.Controls(0)
   y = x
   Debug.Print y

' After upgrade to Visual Basic .NET
   Dim x As String
   Dim y As Object
   x = "Hello"
   'UPGRADE_WARNING: VB.Controls method Controls.Item has a new behavior.
   y = Form1.DefInstance.Controls(0)
   ' UPGRADE_WARNING: Couldn't resolve default property of object 'y'.
   y = x
   System.Diagnostics.Debug.WriteLine(y)
```

What to do next

- In many cases, nothing needs to be done. If the warning is given for a primitive type rather than an object, nothing needs to be done.
- If possible, modify your code to use early binding before upgrading to Visual Basic .NET.
- If late binding is necessary, determine the default property for the object and add it to your code. The following example shows how to modify the above code, assuming that `Controls(0)` refers to a **TextBox** control:

  ```
  Dim x As String
  Dim y As Object
  x = "Hello"
  y = Me.Controls(0).Text
  y = x
  Console.WriteLine y
  ```

See Also

Preparing a Visual Basic 6.0 Application for Upgrading

LoadResStrings method may need to be replaced

In Visual Basic 6.0, applications created with the **VB Application Wizard** could optionally store string resources in an external resource file. If the **Would you like to use a Resource file for the strings in your application?** option was selected, a function named **LoadResStrings** was created in the .bas module for the project. This function used the Visual Basic 6.0 **LoadResString** function to extract string resources at run time.

In Visual Basic .NET, resources are handled differently than they were in Visual Basic 6.0. The Visual Basic Compatibility library contains a **VB6.LoadResString** function to duplicate Visual Basic 6.0 functionality.

During upgrade, if the **LoadResStrings** function is detected, the code within the function is not upgraded. This is because if the code generated by the wizard was subsequently modified, the upgrade tool cannot determine the correct replacement code. You will need to manually modify the code within this function in order to correctly load the string resources in Visual Basic .NET.

What to do next

- If the code generated by the **VB Application Wizard** was not modified, you can perform a cut-and-paste operation on the following code and place it into your Visual Basic .NET application to replace the **LoadResStrings** function:

```
' Visual Basic .NET
Sub LoadResStrings(ByRef frm As System.Windows.Forms.Form)
   On Error Resume Next
   Dim ctl As System.Windows.Forms.Control
   Dim obj As Object
   Dim fnt As System.Drawing.Font
   Dim sCtlType As String
   Dim nVal As Short

   ' Set the form's caption.
   frm.Text = VB6.LoadResString(CShort(frm.Tag))

   ' Set the font.
   Dim FontName As String, FontSize As Double
   FontName = VB6.LoadResString(20)
   FontSize = CShort(VB6.LoadResString(21))
   frm.Font = New System.Drawing.Font(FontName, FontSize)

   ' Set the controls' captions using the caption
   ' property for menu items and the Tag property
   ' for all other controls.
   For Each ctl In frm.Controls
      ctl.Font = fnt
      sCtlType = TypeName(ctl)
      If sCtlType = "Label" Then
         ctl.Text = VB6.LoadResString(CShort(ctl.Tag))
      ElseIf sCtlType = "AxTabStrip" Then
         For Each obj In CObj(ctl).Tabs
            obj.Caption = VB6.LoadResString(CShort(obj.Tag))
            obj.ToolTipText = VB6.LoadResString(CShort(obj.ToolTipText))
         Next obj
      ElseIf sCtlType = "AxToolbar" Then
         For Each obj In CObj(ctl).Buttons
            obj.ToolTipText = VB6.LoadResString(CShort(obj.ToolTipText))
         Next obj
      ElseIf sCtlType = "AxListView" Then
         For Each obj In CObj(ctl).ColumnHeaders
            obj.Text = VB6.LoadResString(CShort(obj.Tag))
         Next obj
      Else
         nVal = 0
```

```
            nVal = Val(ctl.Tag)
            If nVal > 0 Then ctl.Text = VB6.LoadResString(nVal)
            nVal = 0
            nVal = Val(CObj(frm).ToolTip1.GetToolTip(ctl))
            If nVal > 0 Then
               CObj(frm).ToolTip1.SetToolTip(ctl, VB6.LoadResString(nVal))
            End If
         End If
      Next ctl

      Dim mnu As System.Windows.Forms.MainMenu = CObj(frm).MainMenu1
      If Not mnu Is Nothing Then
         LoadMenuResStrings(mnu)
      End If
   End Sub

   Public Sub LoadMenuResStrings(ByVal mnu As System.Windows.Forms.Menu)
      Dim mnuItem As System.Windows.Forms.MenuItem
      For Each mnuItem In mnu.MenuItems
         On Error Resume Next
         mnuItem.Text = VB6.LoadResString(CInt(mnuItem.Text))
         On Error Goto 0
         If mnuItem.MenuItems.Count > 0 Then
            LoadMenuResStrings(mnuItem)
         End If
      Next
   End Sub
```

- If the code generated by the **VB Application Wizard** was modified, modify the above code to accommodate the changes, then perform a cut-and-paste operation on the code and place it into your Visual Basic .NET application to replace the **LoadResStrings** function. You can use the functions in the above code as an example of how to modify your own code.

See Also

Visual Basic 6.0 Compatibility Library

Lower bound of array was changed to 0

Previous versions of Visual Basic allowed you to set the lower bounds of an array to 1 using the **Option Base** statement, or to any valid number using the syntax MyArray(*lowerbounds* to *upperbounds*) where *upperbounds* and *lowerbounds* are numbers. Arrays in Visual Basic .NET are required to have lower bounds of 0.

During upgrade, the lower bounds of all arrays are changed to 0, as in the following example:

```
' Visual Basic 6.0
Dim x(1 To 10) As Integer
Dim z() As Integer
ReDim z(5 To 15) As Integer

' After upgrade to Visual Basic .NET
' UPGRADE_WARNING: Lower bound of array x was changed from 1 to 0.
Dim x(10) As Short
Dim z() As Short
' UPGRADE_WARNING: Lower bound of array z was changed from 5 to 0.
ReDim z(15)
```

What to do next

Review your code to see where and how the array is being used.

- If the original array had a lower bound greater than 0, the elements of the array remain the same:

```
' Visual Basic 6.0
Dim x(1 To 10) As Integer
' Assign a value to the first element.
x(1) = 100

' After upgrade to Visual Basic .NET
' UPGRADE_WARNING: Lower bound of array x was changed from 1 to 0.
Dim x(10) As Integer
' Assign a value to the first element (now the second element).
x(1) = 100
```

- If any code relies on the size of the array (for example, passing the array to an API function, writing to a file, or assigning to another array), it will be incorrect since the size of the array has changed:

```
' Visual Basic 6.0
Dim x(10 To 15) As Integer
Dim y As Integer
y = UBound(x) - LBound(x)
' Returns 4.
Debug.Print y

' After upgrade to Visual Basic .NET
' UPGRADE_WARNING: Lower bound of array x was changed from 10 to 0.
Dim x(15) As Short
Dim y As Short
y = UBound(x) - LBound(x)
' Returns 15.
System.Diagnostics.Debug.WriteLine(y)
```

See Also

Array Bound Changes in Visual Basic | Array Size Declaration Changes in Visual Basic

LSet may not be used to assign a UDT from one type to another

In Visual Basic 6.0, the **LSet** statement could optionally be used to copy a variable of one user-defined type into a variable of another user-defined type. The primary use for the **LSet** statement was to left align a string within a string variable; the use of **LSet** to copy user-defined types was not recommended.

Visual Basic .NET does not allow the use of **LSet** to assign one user-defined type (now structure) to another. When a Visual Basic 6.0 application is upgraded, code that uses **LSet** for assignment will be flagged as an upgrade issue.

The **LSet** method in Visual Basic .NET can still be used to left align strings; the new **PadLeft** method provides additional capabilities and may be a more suitable replacement.

The following code illustrates the use of the **LSet** statement:

```
' Visual Basic 6.0
Private Type typTypeA
   x As Integer
End Type
Private Type typTypeB
   x As Integer
End Type

Sub AssignType()
   Dim aType As typTypeA
   Dim bType As typTypeB
   LSet aType = bType
End Sub

Sub StringAlign()
   Dim strString1 As String
   Dim strString2 As String
   LSet strString1 = strString2
End Sub

' After upgrade to Visual Basic .NET
Private Structure typTypeA
   Dim x As Short
End Type
Private Structure typTypeB
   Dim x As Short
End Type

Sub AssignType()
   Dim aType As typTypeA
   Dim bType As typTypeB
   ' UPGRADE_ISSUE: LSet cannot assign one type to another.
   aType = LSet(bType)
End Sub

Sub StringAlign()
   Dim strString1 As String
   Dim strString2 As String
   ' The statement is replaced by a method.
   strString1 = LSet( strString2, Len(strString1) )
End Sub
```

What to do next

- Review your code to determine if the user-defined types contain the exact same types. Visual Basic 6.0 sometimes used type coercion to automatically convert a variable of one type to another; in Visual Basic .NET, type mismatches will cause a compilation error.
- If the user-defined types are an exact match, you can modify the above code as follows:

```
Private Structure typTypeA
   Dim x As Integer
```

```
End Type
Private Structure typTypeB
   Dim x As Integer
End Type

Sub AssignType()
   Dim aType As typTypeA
   Dim bType As typTypeB
   AType.x = bType.x
End Sub
```

For structures that contain multiple elements, you will need to assign each element individually.

See Also

String.PadLeft Method

MDI form Show behavior has changed

In Visual Basic 6.0 the **AutoShowChildren** property of an MDI form was used to automatically show an MDI child form when it was loaded. Standard forms did not have this behavior; calling the **Load** method did not show the form.

In Visual Basic .NET, the **AutoShowChildren** property no longer exists. To emulate the Visual Basic 6.0 behavior, a line of code is added during upgrade to show each child form if **AutoShowChildren** was set to **true** in the original application.

What to do next

- If you do not want to show the child form when the MDI form is loaded, remove the line of code containing the **Show** method.

See Also

MDIForm Object Changes in Visual Basic .NET | MDI Changes in Visual Basic .NET

MTS or COM+ objects were not upgraded

The project that you are attempting to upgrade has a reference to the COM+ Services Type Library (Comsvcs.dll) that cannot be upgraded. In Visual Basic 6.0, this library was used to access Microsoft Transaction Services (MTS). In Visual Basic .NET, MTS is replaced by transaction processing functionality in the .NET Framework.

What to do next

- Because of differences in the transaction processing models, existing code will need to be deleted and replaced by code that uses .NET transaction processing.

See Also

Processing Transactions

NewEnum property was commented out

In Visual Basic 6.0, collection classes created using the Class Builder add-in included a **NewEnum** property that allowed you to enumerate the collection using **For...Each** syntax.

Collections in Visual Basic .NET support enumeration through the **System.Collections.IEnumerator** class; the **NewEnum** property is no longer needed.

What to do next

- Remove the code for the **NewEnum** property that was commented out.
- Review the code that was added by the upgrade tool.

See Also

Uncomment and change the following line to return the collection enumerator | System.Collections Namespace

NewIndex property cannot be upgraded

In Visual Basic 6.0 the **NewIndex** property was used to retrieve the index of the item most recently added to a **ListBox** or **ComboBox** control. During upgrade, the value of **NewIndex** cannot be determined in some cases. The return value from the **Items.Add** method can be used instead of **NewIndex**.

What to do next

- Review the usage of the **NewIndex** property and modify your code as in the following example.

```
' Visual Basic 6.0
List1.AddItem "Hello"
List1.ListIndex = List1.NewIndex
```

```
' After upgrade to Visual Basic .NET
List1.Items.Add("Hello")
' UPGRADE_ISSUE: ListBox property Form1.List1.NewIndex was not upgraded.
List1.SelectedIndex = List1.NewIndex    ' Causes a compilation error.
```

The Visual Basic .NET code should then be modified as follows:

```
Dim x As Integer
x = List1.Items.Add("Hello")
List1.SelectedIndex = x
```

See Also

ComboBox Control Changes in Visual Basic .NET | ListBox Control Changes in Visual Basic .NET

Object could not be resolved because it was within a generic namespace

In Visual Basic 6.0 you could use the **ActiveForm** property of a **Forms** collection to refer to controls on the currently selected form; controls were actually treated as properties of the form. In Visual Basic .NET, **ActiveForm** is a generic type that has no controls. For example, the following code works in Visual Basic 6.0 but cannot compile after being upgraded to Visual Basic .NET:

```
' Visual Basic 6.0
MDIForm1.ActiveForm!Label1.Caption = "Active"

' After upgrade to Visual Basic .NET
' Causes a compilation error.
MDIForm1.DefInstance.ActiveMDIChild.Label1.Caption = "Active"
```

What to do next

- Modify your code to refer to the underlying form object:

```
' Modified Visual Basic .NET code
CType(MDIForm1.DefInstance.ActiveMDIChild, Form1).Label1.Text = "Active"
```

> **Note** Using the **CType** function makes this late bound. The original code in this example is problematic because it assumes that the **ActiveForm** property will always have a label named Label1 — if Label1 does not exist, a run-time error will occur.

See Also

Screen Object Changes in Visual Basic .NET | Forms Collection Changes in Visual Basic .NET

Object or collection was not upgraded

Due to differences between Visual Basic 6.0 and Visual Basic .NET, some objects or collections cannot be upgraded.

What to do next

Select an object from the list below to see a topic listing changes and issues for that object.

Forms Collection Changes in Visual Basic .NET

Printer Object Changes in Visual Basic .NET

Printers Collection Changes in Visual Basic .NET

Clipboard Object Changes in Visual Basic .NET

App Object Changes in Visual Basic .NET

WebClass Changes in Visual Basic .NET

See Also

Things to Consider Before Upgrading

Object was upgraded to a panel, and cannot be coerced to a PictureBox

When an application is upgraded from Visual Basic 6.0, if a **PictureBox** control is used as a container (that is, it contains child controls), it is upgraded to **Panel** control; otherwise, it is upgraded to a **PictureBox** control.

If a **PictureBox** control that has been upgraded to a **Panel** control is assigned to a variable of type **PictureBox**, the assignment will fail.

If a **PictureBox** control that was upgraded to a **Panel** control is passed as a parameter to a function, a run-time error will occur because the parameter is expecting a **PictureBox** control.

The following example shows code that would cause these errors:

```
' Visual Basic 6.0
Function PictureOrPanel()
   Dim p As PictureBox
   ' Not used as a container.
   Set p = Me.PictureBoxWithoutControls
   MyFunction Me.PictureBoxWithoutControls
   ' Used as a container.
   Set p = Me.PictureBoxWithControls
   MyFunction Me.PictureBoxWithControls
End Function

Function MyFunction( p As PictureBox )
   ' Do something here.
End Function

' After upgrade to Visual Basic .NET
Function PictureOrPanel() As Object
   Dim p As System.Windows.Forms.PictureBox
   ' Not used as a container.
   Set p = Me.PictureBoxWithoutControls
   MyFunction Me.PictureBoxWithoutControls
   ' Used as a container.
   ' UPGRADE_ISSUE: Form1.PictureBoxWithControls was upgraded to a Panel and cannot be coerce
d to a PictureBox.
   Set p = Me.PictureBoxWithControls
   ' UPGRADE_ISSUE: Form1.PictureBoxWithControls was upgraded to a Panel and cannot be coerce
d to a PictureBox.
   MyFunction Me.PictureBoxWithControls
End Function

Function MyFunction( p As PictureBox )
   ' Do something here.
End Function
```

What to do next

- If the **Panel** control is being assigned to a variable of type **PictureBox**, create a new variable of type **Panel** and assign to that. If the **Panel** control is being passed as a parameter to a function, create a second version of the function with a parameter of type **Panel**.

```
Function PictureOrPanel() As Object
   Dim p As System.Windows.Forms.PictureBox
   Dim q As System.Windows.Forms.Panel
   ' Not used as a container.
   Set p = Me.PictureBoxWithoutControls
   MyFunction Me.PictureBoxWithoutControls
   ' Used as a container.
   Set q = Me.PictureBoxWithControls
   MyFunction2 Me.PictureBoxWithControls
```

```
End Function

Function MyFunction( p As PictureBox )
   ' Do something here.
End Function

Function MyFunction2( p As Panel )
   ' Do something here.
End Function
```

See Also

PictureBox Control Changes in Visual Basic .NET

Objects of type vbDataObject are not supported

In Visual Basic 6.0, objects created in C++ that did not support the IDispatch interface were treated as a **Variant** with a type of 13 (vbDataObject). In Visual Basic .NET, objects that do not support IDispatch are no longer allowed.

During upgrade, any reference in code to `VarType() = vbDataObject` or `VarType() = 13` will be flagged with a warning.

What to do next

- Remove the line of code, or modify it to use a different object.
- If you have access to the source code for the object, modify it to support IDispatch.

See Also

Preparing a Visual Basic 6.0 Application for Upgrading

Only OpenType and TrueType fonts are supported

Visual Basic 6.0 supported standard fonts for backward compatibility; Visual Basic .NET supports only TrueType and OpenType fonts. The font for this control or form has been changed to the default OpenType font Microsoft Sans Serif, and any formatting such as bold or italic has not been preserved.

What to do next

- Review the appearance of the font on your form or control.
  - If you are happy with it, do nothing.
  - If you are not happy with the selected font, change the font by selecting the **Font** property in the Properties window or by setting it in code.

    Note If the Visual Basic 6.0 application used formatting such as bold or italic, it may be easier to change the font in the original application to a TrueType or OpenType font, and then run the upgrade tool again.

See Also

Font Changes in Visual Basic .NET

ParamArray parameter has been changed from ByRef to ByVal

In Visual Basic, the **ParamArray** keyword could be used to specify an arbitrary number of parameters for a declaration, sub, or function. ParamArrays were always passed by reference (**ByRef**). In Visual Basic .NET, ParamArrays are always passed by value; the **ParamArray** keyword must be preceded by the **ByVal** keyword.

When a parameter is passed by value, a copy of the variable is passed; when passed by reference, a pointer to the variable is passed. When passing by value, it is possible that the value of the variable may have changed and that the copy is out of date.

During upgrade, **ParamArray** parameters are modified as follows:

```
' Visual Basic 6.0
Sub HasParameters(ParamArray X())

' After upgrade to Visual Basic .NET
' UPGRADE_WARNING: ParamArray X was changed from ByRef to ByVal.
Sub HasParameters(ParamArray ByVal X() As Object)
```

What to do next

- Review your code to make sure that the variables being passed as parameters are not being changed between the time that they are set and the time that they are used in a procedure. If necessary, modify your code so that the procedure executes before the variables are modified.

See Also

7.1.5.4 Paramarray Parameters

Picture will be tiled

When a Visual Basic 6.0 **PictureBox** control is used as a control container, it is upgraded to a Visual Basic .NET **Panel** control rather than a **PictureBox** control, because the Visual Basic .NET **PictureBox** control cannot be used as a container.

In Visual Basic 6.0, an image specified by the **Picture** property would be displayed in the upper left corner of the **PictureBox** control; the **AutoSize** property determined whether the picture box would be resized to match the dimensions of the image. The Visual Basic .NET **Panel** control does not have an **AutoSize** property, so any image assigned to the **BackgroundImage** property will be tiled and the panel will not be automatically resized.

What to do next

- Run the upgraded application and observe the behavior. If you are happy with the behavior, do nothing.
- If the intended behavior was to resize the picture box to match the size of the image, resize the panel at design time.
- If it is necessary to resize the panel when a new image is assigned at run time, add code similar to the following to the event handler that assigns the new image.

```
' Visual Basic .NET
Picture1.BackgroundImage = _ System.Drawing.Image.FromFile("C:\SomePicture.bmp")
Picture1.Height = Picture1.BackgroundImage.Height
Picture1.Width = Picture1.BackgroundImage.Width
```

See Also

PictureBox Control Changes in Visual Basic .NET

Project item has an invalid format

The referenced file is a Visual Basic 5.0 (or earlier) file. The upgrading tools in Visual Basic .NET do not support upgrading Visual Basic 5.0 files. While it is possible to upgrade a Visual Basic 5.0 file, doing so could cause unpredictable results in Visual Basic .NET.

What to do next

- Upgrade the referenced file to Visual basic 6.0 by opening and saving the file in Visual Basic 6.0, and then upgrade the project to Visual Basic .NET.

See Also

Preparing a Visual Basic 6.0 Application for Upgrading

Project item is not a Visual Basic 6.0 file

The referenced item does not match any known Visual Basic 6.0 file format and cannot be upgraded. It may not be a Visual Basic 6.0 file, it may be from an older version of Visual Basic, or it may have the wrong file extension (for example, a .bas module with an .frm extension).

What to do next

Make sure that the project can be successfully compiled and run using Visual Basic 6.0 on the upgrade computer.

See Also

Preparing a Visual Basic 6.0 Application for Upgrading | Working with Both Visual Basic 6.0 and Visual Basic .NET

Project item was not upgraded

An item in the Visual Basic 6.0 project could not be upgraded because it is of a type not supported by Visual Basic .NET. The item was copied into the new project directory; however, it cannot be opened in Visual Studio .NET.

What to do next

Review the help topic Things to Consider Before Upgrading, which contains a list of unsupported features.

- In some cases unsupported features can be used in Visual Basic .NET after some modification in Visual Basic 6.0.
- In most cases you should be able to duplicate the functionality of the project item using the new features of Visual Basic .NET.
- If the referenced item is critical to the project and cannot easily be duplicated, you may need to reconsider your upgrade decision.

See Also

Things to Consider Before Upgrading | Preparing a Visual Basic 6.0 Application for Upgrading | Working with Both Visual Basic 6.0 and Visual Basic .NET | What's New in Visual Basic and Visual C#

Property is not supported at run time

Some properties that could be set at run time in Visual Basic 6.0 can only be set at design time in Visual Basic .NET, or cannot be mapped properly at run time.

What to do next

Select an object or control from the list below to see a topic listing changes and issues for that object.

CheckBox Control Changes in Visual Basic .NET

ComboBox Control Changes in Visual Basic .NET

Form Object Changes in Visual Basic .NET

Image Control Changes in Visual Basic .NET

Label Control Changes in Visual Basic .NET

Line Control Changes in Visual Basic .NET

ListBox Control Changes in Visual Basic .NET

MDIForm Object Changes in Visual Basic .NET

PictureBox Control Changes in Visual Basic .NET

Shape Control Changes in Visual Basic .NET

TextBox Control Changes in Visual Basic .NET

UserControl Changes in Visual Basic .NET

See Also

Control Changes in Visual Basic .NET

Property, Method, or Event was upgraded to a different name and has a new behavior

Some Visual Basic 6.0 properties, methods, and events are upgraded to Visual Basic .NET equivalents that have different names and can have subtle differences in behavior. For example, in Visual Basic 6.0 the **Change** event for a **ComboBox** control was only raised when the text was changed in the text entry portion of the control; in Visual Basic .NET the **TextChanged** event will also be raised when an item is selected in the list portion of the control.

What to do next

Select an object or control from the list below to see a topic listing changes and issues for that object.

ComboBox Control Changes in Visual Basic .NET

Controls Collection Changes in Visual Basic .NET

See Also

Control Changes in Visual Basic .NET | UserControl Changes in Visual Basic .NET

Property or method has a new behavior

Some Visual Basic 6.0 properties, methods, and events can have subtle differences in behavior after being upgraded to Visual Basic .NET. For example, in Visual Basic 6.0 the **Change** event for a **ComboBox** control was only raised when the text was changed in the text entry portion of the control; in Visual Basic .NET the event will also be raised when an item is selected in the list portion of the control.

What to do next

Select an object or control from the list below to see a topic listing changes and issues for that object.

CheckBox Control Changes in Visual Basic .NET

ComboBox Control Changes in Visual Basic .NET

CommandButton Control Changes in Visual Basic .NET

Controls Collection Changes in Visual Basic .NET

DirListBox Control Changes in Visual Basic .NET

DriveListBox Control Changes in Visual Basic .NET

FileListBox Control Changes in Visual Basic .NET

Form Object Changes in Visual Basic .NET

Frame Control Changes in Visual Basic .NET

HScrollBar Control Changes in Visual Basic .NET

Image Control Changes in Visual Basic .NET

Label Control Changes in Visual Basic .NET

Line Control Changes in Visual Basic .NET

ListBox Control Changes in Visual Basic .NET

MDIForm Object Changes in Visual Basic .NET

MousePointer Changes in Visual Basic .NET

OptionButton Control Changes in Visual Basic .NET

PictureBox Control Changes in Visual Basic .NET

Shape Control Changes in Visual Basic .NET

TextBox Control Changes in Visual Basic .NET

Timer Control Changes in Visual Basic .NET

UserControl Changes in Visual Basic .NET

VScrollBar Control Changes in Visual Basic .NET

WebClass Changes in Visual Basic .NET

See Also

Control Changes in Visual Basic .NET

Property or method was not upgraded

Due to differences between Visual Basic 6.0 and Visual Basic .NET, some objects cannot be upgraded. References to properties or methods of those objects have been copied into the upgraded code, but they will not compile because the underlying objects were not upgraded.

What to do next

Select an object from the list below to see a topic listing changes and issues for that object.

App Object Changes in Visual Basic .NET

Clipboard Object Changes in Visual Basic .NET

Controls Collection Changes in Visual Basic .NET

Data Control Changes in Visual Basic .NET

Line Control Changes in Visual Basic .NET

OLE Container Control Changes in Visual Basic .NET

Printer Object Changes in Visual Basic .NET

Printers Collection Changes in Visual Basic .NET

RDO Data Control Changes in Visual Basic .NET

Shape Control Changes in Visual Basic .NET

UserControl Changes in Visual Basic .NET

WebClass Changes in Visual Basic .NET

See Also

Things to Consider Before Upgrading

Property or method was not upgraded

Due to differences between Visual Basic 6.0 and Visual Basic .NET, some properties and methods cannot be upgraded. For example, some legacy features such as Dynamic Data Exchange are no longer supported; consequently any code that sets DDE-related properties or methods would cause compilation errors in Visual Basic .NET

What to do next

Select an object or control from the list below to see a topic listing changes and issues for that object.

App Object Changes in Visual Basic .NET

CheckBox Control Changes in Visual Basic .NET

Clipboard Object Changes in Visual Basic .NET

ComboBox Control Changes in Visual Basic .NET

CommandButton Control Changes in Visual Basic .NET

Controls Collection Changes in Visual Basic .NET

DirListBox Control Changes in Visual Basic .NET

DriveListBox Control Changes in Visual Basic .NET

FileListBox Control Changes in Visual Basic .NET

Font Object Changes in Visual Basic .NET

Form Object Changes in Visual Basic .NET

Frame Control Changes in Visual Basic .NET

HScrollBar Control Changes in Visual Basic .NET

Image Control Changes in Visual Basic .NET

Label Control Changes in Visual Basic .NET

Line Control Changes in Visual Basic .NET

ListBox Control Changes in Visual Basic .NET

MDIForm Object Changes in Visual Basic .NET

Menu Object Changes in Visual Basic .NET

OptionButton Control Changes in Visual Basic .NET

PictureBox Control Changes in Visual Basic .NET

Screen Object Changes in Visual Basic .NET

Shape Control Changes in Visual Basic .NET

TextBox Control Changes in Visual Basic .NET

Timer Control Changes in Visual Basic .NET

UserControl Changes in Visual Basic .NET

VBControlExtender Object Changes in Visual Basic .NET

VScrollBar Control Changes in Visual Basic .NET

WebClass Changes in Visual Basic .NET

See Also

Control Changes in Visual Basic .NET

Property or statement is not supported

In Visual Basic 6.0, the **Calendar** property was a stand-alone property that determined the calendar (Gregorian or Hijri) to be used for date-related functions. In Visual Basic .NET, this property is no longer supported; the **Calendar** class in the System.Globalization namespace that supports several additional calendar types replaces it.

In Visual Basic 6.0, the **Load** statement could be used to load a form without displaying it. A form could also be loaded without displaying it by referencing any of its properties.

The **Load** statement is no longer supported in Visual Basic .NET. When an instance of a form is created using the **New** keyword, it is loaded but not displayed until the **Show** method is called.

The following code shows the usage of the **Load** statement:

```
' Visual Basic 6.0
Load Form2
Form2.Caption = "Hello"
MsgBox Form2.Caption

' After upgrade to Visual Basic .NET
'UPGRADE_ISSUE: Load statement is not supported.
Load(Form2)
Form2.DefInstance.Text = "Hello"
MsgBox (Form2.DefInstance.Text)
```

What to do next

- Remove the line of code that sets the **Calendar** property and add a reference to one of the following classes:

```
' If the Calendar property was 0 or vbCalGreg.
Public Class GregorianCalendar
   Inherits Calendar
' If the Calendar property was 1 or vbCalHijri.
Public Class Hijri Calendar
   Inherits Calendar
```

- Remove the line of code that contains the **Load** statement.
- If you need to load the form without showing it, use the **New** keyword, as in the following example:

```
' Modified code
Dim Form2 As New Form2
Form2.Text = "Hello"
MsgBox (Form2.Text)
```

See Also

Calendar Class | Programming Element Support Changes Summary

ScaleMode is not supported

In Visual Basic 6.0, the **ScaleMode** property could be used to change the coordinate system for a form or **PictureBox** control from the default scale of twips.

Visual Basic .NET does not support multiple coordinate systems; only pixels are supported. During upgrade, coordinates are automatically converted from twips to pixels; code that sets the **ScaleMode** property at run time will cause a compilation error and must be modified.

> **Note** The upgrade tool assumes that the design-time setting for the **ScaleMode** property was twips; if this is not the case, the conversion will be incorrect and must be fixed.

What to do next

1. Remove the line of code that sets the **ScaleMode** property.
2. Review any code that was based on the **ScaleMode** property and modify any calculations as necessary. For example, the following code shows how to modify a procedure that used a **ScaleMode** of inches.

```
' Visual Basic 6.0
Form1.ScaleMode = vbInches
Text1.Move 2, 1
```

The above procedure moves the text box 2 inches right and 1 inch down from the upper left corner of the form.

```
' After upgrade to Visual Basic .NET
'UPGRADE ISSUE: Constant vbInches was not upgraded.
'UPGRADE ISSUE: Form property Form1.ScaleMode is not supported.
' The next line must be removed in order to compile.
Form1.ScaleMode = vbInches
' Twips are converted to pixels, but the original inch values are used.
Text1.SetBounds(VB6.TwipsToPixelsX(2), VB6.TwipsToPixelsX(1),...
```

After upgrade, the procedure moves the text box right by 2 pixels and down by 1 pixel — not the desired result.

```
' Modified Visual Basic .NET code
' Removed the ScaleMode line.
' Convert the scale (1 inch = 1440 twips).
Text1.SetBounds(VB6.TwipsToPixelsX(2880), VB6.TwipsToPixelsX(1440),...
```

By multiplying the inches by 1440, the twips to pixel conversion now gives the same result as in the original Visual Basic 6.0 code.

See Also

Form Object Changes in Visual Basic .NET

Setting an object variable to Nothing does not destroy the object

In Visual Basic 6.0, setting an object to **Nothing** immediately destroyed the object. In Visual Basic .NET, setting an object to **Nothing** marks the object for garbage collection, but the object is not immediately destroyed. This could cause problems if other portions of your code depend on the object being destroyed.

For example, if an object is set to **Nothing** inside a procedure and the next line of code creates an object of the same name, the first object may not yet be destroyed and a reference to the new object might incorrectly return the first object.

What to do next

- Review your code and search for additional references to this object.
- If necessary, call the **Dispose** method to destroy the object immediately:

```
' Visual Basic .NET
Dim x As New Button
x.Text = "Goodbye"
x.Dispose()
```

See Also

Lifetime

Some run-time code needs to be commented out to give the same effect as VB6

In Visual Basic 6.0 it was possible to write code that had no effect at run time, but that did not cause a compilation error. For example, the following code attempts to set the read-only **BorderStyle** property for a form at run time.

```
' Visual Basic 6.0
Form1.Borderstyle = 2
Debug.Print Form1.Borderstyle
```

This gave the result of 2, indicating that the **BorderStyle** had been changed, when in fact it had not — the **BorderStyle** was still the default (1).

When this code is upgraded to Visual Basic .NET it may cause unexpected results, since the **BorderStyle** constants are different. In other cases, code that did nothing in Visual Basic 6.0 may cause compilation errors in Visual Basic .NET.

What to do next

- If you want your application to behave as it did in Visual Basic 6.0, comment out (or delete) the line of code.
- If you intended for the line of code to be executed, modify the code. In the above example, if your intent was to actually make the border sizable, modify the code as follows:

  ```
  ' Visual Basic .NET
  Me.Borderstyle = Windows.Forms.FormBorderStyle.Sizable
  System.Diagnostics.Debug.WriteLine(Me.Borderstyle)
  ```

See Also

Control Changes in Visual Basic .NET | UserControl Changes in Visual Basic .NET

Statement is no longer supported

In Visual Basic .NET, the **GoSub...Return** and **On...GoSub** statements are no longer supported. These statements were supported in Visual Basic 6.0 for compatibility with earlier versions of the Basic language; they are not compatible with object-oriented programming practices.

What to do next

- Replace **GoSub** statements with **Call**, **Function**, or **Sub** statements. The **Return** keyword is no longer needed.
- Replace **On...GoSub** statements with **Select Case** statements, as shown in the following example:

```
' Visual Basic 6.0
On x GoSub 1000, 2000
1000:
   MyString = "Hello"
2000:
   MyString = "Goodbye"

' Visual Basic .NET
Select Case x
   Case 1000
      MyString = "Hello"
   Case 2000
      MyString = "Goodbye"
End Select
```

See Also

Control Statement Changes in Visual Basic

Structure may require marshalling attributes to be passed as an argument in this Declare statement

In Visual Basic 6.0, user-defined types could be passed as an argument in a **Declare** statement for a Windows API.

In Visual Basic .NET, a structure (user-defined type) passed as an argument in a **Declare** statement may require additional marshalling attributes in order to be passed correctly to the external function or subroutine. In particular, arrays and fixed-length strings may not function as expected without these attributes.

The following example shows a structure before and after upgrade:

```
' Visual Basic 6.0 code
Type MyType
   s As String * 100
End Type

' After upgrade to Visual Basic .NET
Structure MyType
   <VBFixedString(100), System.Runtime.InteropServices.MarshalAs(System.Runtime.InteropServic
es.UnmanagedType.ByValTStr, SizeConst:=100)> Public s As String
End Structure
```

What to do next

- Add an **Imports** statement to reference the **InteropServices** namespace:

  ```
  Imports System.Runtime.InteropServices
  ```

- Modify the structure and the string declaration to include marshalling attributes, as in the following example:

  ```
  ' Modified code
  <StructLayout( LayoutKind.Sequential, CharSet:=CharSet.ANSI)>Structure MyType
     <MarshalAs(UnmanagedType.ByValTStr, SizeConst:=100)> Public s As String
  End Structure
  ```

See Also

Applying Attributes | Passing Structures

Sub Main will not be called at component startup

In Visual Basic 6.0, any code in a **Sub_Main** procedure for a .dll or .ocx file would be run automatically on startup. In Visual Basic .NET, **Sub_Main** cannot be set as a startup object; any code in the **Sub_Main** procedure will not be run on startup.

What to do next

- Add code to the constructor (**Sub New**) to call the **Sub_Main** procedure whenever an instance of the object is created:

```
Public Sub New()
   Main()
End Sub
```

- Alternatively, move the code from the **Sub_Main** procedure to the **Sub_New** procedure.

See Also

What's New in Components and Component Authoring

The referenced component is missing a design-time license

The application that you are upgrading contains the referenced component, which requires a design-time license. The license file is missing, or may have been moved to a different location.

Some ActiveX controls or components require a license in order to be used in Visual Basic. At design time, when you add the control to a form, a check is made for a license file on the development computer; if not found, the control cannot be used.

> **Tip** You should always test your application in Visual Basic 6.0 on the development computer before upgrading. If it runs in Visual Basic 6.0, this upgrade error will not occur.

To correct this error

- Determine if the referenced component is actually required for your application; if not, you can remove the control and ignore this warning.
- If the component is required, you will need to install the license file on your development computer. The license file is usually installed into the same directory as the component.

> **Note** You may need to check the documentation for the component to determine how to install the license file. Licensing terms for some components restrict installation to a single computer.

See Also

Control Changes in Visual Basic .NET | UserControl Changes in Visual Basic .NET

The referenced project is either not compiled or out-of-date

The project that you are attempting to upgrade has a reference to an ActiveX EXE or ActiveX DLL project. The output of that project must be loaded in order for the upgrade to proceed. Either the referenced project has not been compiled, or the compiled output is missing.

What to do next

- Compile the referenced project in Visual Basic 6.0 by choosing **Make *Projectname*** from the **File** menu.

See Also

Preparing a Visual Basic 6.0 Application for Upgrading

There may be differences in data-binding behavior

When Visual Basic 6.0 applications that use data binding are upgraded to Visual Basic .NET, there are a few behavioral differences that you need to be aware of.

Dispose behavior for BindingCollection object has changed

When using a **BindingCollection** object in a Visual Basic 6.0 project, it was not necessary to set the object to **Nothing**; it would automatically be disposed when the form was closed.

In Visual Basic .NET, the BindingCollection Class must be explicitly disposed in the Form.Closing Event event of the form.

What to do next

- Add the following code to the **Closing** event of the form where the **BindingCollection** object was declared. This example assumes a form named MyDataForm and a **BindingCollection** named ADOBind.

```
' Visual Basic .NET
Private Sub MyDataForm_Closing(ByVal sender as Object, ByVal e As System.ComponentModel.C
ancelEventArgs) Handles MyBase.Closing
   ADOBind.Clear()
   ADOBind.Dispose()
   ADOBind = Nothing
End Sub
```

Data-bound properties cannot be bound to Null values

In Visual Basic 6.0, properties of a control could be bound to a database field using the **BindingCollection** object; if the database value was **Null**, the type was automatically coerced to the type of the property.

In Visual Basic .NET, only properties that are of the **String** data type can be bound to a database value that is **Null**; due to strict typing, binding any other data type will cause a InvalidCastException error.

What to do next

- Determine the data type for each property bound to a database field, and either remove the binding for any property whose data type is not **String**, or modify the database so that it does not allow **Null** values.

See Also

Data Changes in Visual Basic .NET

Timer Interval property behavior has changed

In Visual Basic 6.0 you could disable a **Timer** control by setting the **Interval** property to 0. In Visual Basic .NET, the lower range for the **Interval** property is 1. If you set the interval to 0, it will throw a run-time exception. The Visual Basic .NET Timer component uses the **Enabled** property to allow you to disable or enable it.

The following code illustrates the changes to the **Timer** control.

```
' Visual Basic 6.0
Public Function TimerOn(Interval As Integer)
   If Interval > 0 Then
      Timer1.Interval = Interval   ' Enables the timer.
   Else
      Timer1.Interval = 0         ' Disables the timer.
   End If
End Function
```

```
' After upgrade to Visual Basic .NET
Public Function TimerOn(ByRef Interval As Short) As Object
   If Interval > 0 Then
      ' UPGRADE_WARNING: Timer Property Form1.Timer1.Interval has a new
      ' behavior.
      Timer1.Interval = Interval   ' Enables the timer.
   Else
      ' UPGRADE_WARNING: Timer property Timer1.Interval cannot have
      ' a value of 0.
      Timer1.Interval = 0         ' Disables the timer.
   End If
End Function
```

What to do next

- Find any code that sets the **Interval** property to 0.
- Replace it with code that sets the **Enabled** property to **false**.
- Find any code that sets the **Interval** property to a value greater than 0.
- Add a line of code to set the **Enabled** property to **true**.

> **Note** The **Enabled** property only needs to be set the first time the interval is set. The **Timer** will remain enabled until it is explicitly disabled by setting the **Enabled** property to **false**.

```
' Modified Visual Basic .NET code
Public Function TimerOn(ByRef Interval As Short) As Object
   If Interval > 0 Then
      Timer1.Enabled = True      ' Enables the timer.
   Else
      Timer1.Enabled = False      ' Disables the timer.
   End If
End Function
```

See Also

Timer Control Changes in Visual Basic .NET

Unable to determine which equivalent constant is appropriate for this context

In some cases, constants or enumerations for Visual Basic 6.0 are not upgraded because there are two or more Visual Basic .NET equivalents. For example, the Visual Basic 6.0 constant **vbRightJustify** could be mapped to **System.Windows.Forms.HorizontalAlignment.Right**, **System.Drawing.ContentAlignment.TopRight**, or **System.Drawing.ContentAlignment.MiddleRight** in Visual Basic .NET, depending on the type of control. In cases where the type of control is not known, the constant cannot be mapped.

```
' Visual Basic 6.0
Dim x As Control
Set x = Text1
x.Alignment = vbRightJustify
```

```
' After upgrade to Visual Basic .NET
Dim x As Object
x = Text1
' UPGRADE_ISSUE: Unable to determine which constant to upgrade
' vbRightJustify to.
x.Alignment = vbRightJustify
```

What to do next

- Determine which Visual Basic .NET constant or enumeration is appropriate for the given context and modify the code. You can use the Intellisense feature of the code editor to get a list of available enumerations.

  ```
  ' Modified Visual Basic .NET code
  Dim x As Object
  x = Text1
  x.TextAlign = System.Windows.Forms.HorizontalAlignment.Right
  ```

See Also

Constant Changes in Visual Basic .NET

Uncomment and change the following line to return the collection enumerator

In Visual Basic 6.0, collection classes created using the Class Builder add-in included a **NewEnum** property that allowed you to enumerate the collection using **For...Each** syntax. It was also possible to create your own collection classes and add your own enumerator property.

Collections in Visual Basic .NET support enumeration through the **System.Collections.IEnumerator** class; the **GetEnumerator** function was added to your code to replace the **NewEnum** property.

What to do next

- For collections created using the Class Builder, mCol is the default name for the collection class. If you changed the default name, or if this is a collection that you created on your own, you will need to replace mCol with the name of your collection class:

```
GetEnumerator = MyCollection.GetEnumerator
```

- If you used the default name (mCol), uncomment the line containing the GetEnumerator assignment:

```
GetEnumerator = mCol.GetEnumerator
```

See Also

NewEnum property was commented out | System.Collections Namespace

Unload method cannot be upgraded

In cases where an object is late bound, the **Unload** method cannot be upgraded. For example, the following Visual Basic 6.0 code will cause a compilation error after upgrading to Visual Basic .NET.

```
' Visual Basic 6.0
Dim o As Object
Private Sub Command1_Click()
    Set o = New Form1
    o.Show
End Sub

Private Sub Command2_Click()
    Unload o
End Sub
```

```
' After upgrade to Visual Basic .NET
Dim o As Object
Private Sub Command1_Click(ByVal EventSender As System.Object, ByVal EventArgs As System.Even
tArgs) Handles Command1.Click
    o = New Form1
    o.Show()
End Sub

Private Sub Command2_Click(ByVal EventSender As System.Object, ByVal EventArgs As System.Even
tArgs) Handles Command2.Click
    ' UPGRADE_ISSUE: Unload o was not upgraded.
    Unload o
End Sub
```

What to do next

- Modify the code to be early bound to Form1 instead of Object, and call the **Close** method instead of **Unload**.

```
' Modified Visual Basic .NET code
Dim o As Form1
Private Sub Command1_Click(ByVal EventSender As System.Object, ByVal EventArgs As System.
EventArgs) Handles Command1.Click
    o = New Form1
    o.Show()
End Sub

Private Sub Command2_Click(ByVal EventSender As System.Object, ByVal EventArgs As System.
EventArgs) Handles Command2.Click
    o.Close()
End Sub
```

See Also

Form Object Changes in Visual Basic .NET | Control Changes in Visual Basic .NET

Value for a property could not be resolved

During upgrade, some Visual Basic 6.0 properties are mapped to Visual Basic .NET properties with different values. In these cases, properties that are resolved at run time by evaluating a variable or function return cannot be resolved by the upgrade tool.

For example, the **AutoSize** property of a **PictureBox** control is upgraded to the **SizeMode** property of the Windows Forms **PictureBox** control. If the property was set to **true** or **false** (or their numeric equivalents) in the original code, it will be mapped to **System.Windows.Forms.PictureBoxSizeMode.AutoSize** (for **true**) or **System.Windows.Forms.PictureBoxSizeMode.Normal** (for **false**). If the property was set to the value of a variable or function return, the value is not known and cannot be properly upgraded.

```
' Visual Basic 6.0
Dim x As Boolean
x = False
Picture1.AutoSize = True
Picture2.AutoSize = False
Picture3.AutoSize = x
' MyAutoSize is a function that returns either True or False.
Picture4.AutoSize = MyAutoSize ()

' After upgrade to Visual Basic .NET
Dim x As Boolean
x = False
Picture1.SizeMode = System.Windows.Forms.PictureBoxSizeMode.AutoSize
Picture2.SizeMode = System.Windows.Forms.PictureBoxSizeMode.Normal
' x cannot be resolved.
Picture3.SizeMode = x
' MyAutoSize () cannot be resolved.
Picture4.SizeMode = MyAutoSize ()
```

What to do next

- If the property value was stored in a variable, change the value (and type if necessary) of the variable where it is set.

```
' Modified VisualBasic .NET code
Dim x As String
x = " System.Windows.Forms.PictureBoxSizeMode.Normal"
Picture3.SizeMode = x
```

  Note Before modifying the variable, search your code to make sure it is not referenced elsewhere. If the variable is shared, you may need to create a new variable rather than modifying the existing variable.

- If the property value was the return value of a function, modify the function to return the correct value.

```
' Visual Basic 6.0
Public Function MyAutoSize () As Boolean
    MyAutoSize = True
End Function

' Modified Visual Basic .NET code
Public Function MyAutoSize () As System.Windows.Forms.PictureBoxSizeMode
   MyAutoSize = System.Windows.Forms.PictureBoxSizeMode.AutoSize
End Function
```

See Also

Control Changes in Visual Basic .NET

Variables must be explicitly declared

In Visual Basic 6.0, it was possible to declare a group of variables in a single **Dim** statement. A common usage was to declare a series of letters as integer values: `DefInt A-Z`.

In Visual Basic .NET, each variable must be individually declared; the **DefBool**, **DefByte**, **DefInt**, **DefLng**, **DefCur**, **DefSng**, **DefDbl**, **DefDec**, **DefDate**, **DefStr**, **DefObj**, and **DefVar** statements are no longer supported.

During upgrade, any variables declared using these statements are explicitly declared as the correct type as shown in the following example:

```
' Visual Basic 6.0
DefInt A-C

' After upgrade to Visual Basic .NET
' UPGRADE_NOTE: DefInt A-C statement was removed. Variables were explicitly declared as type
Short.
Dim A As Short
Dim B As Short
Dim C As Short
```

What to do next

- No further action is necessary. This note was added to make you aware of the syntax change and the changes made to your code.

See Also

Data Type Changes in Visual Basic

Visual Basic 6.0 function was upgraded to a different Visual Basic .NET function

When a Visual Basic 6.0 application is upgraded to Visual Basic .NET, some functions are upgraded to functions that, although they behave the same, have a different name.

For example, the **Erase** function for fixed-length arrays upgrades to **System.Array.Clear**:

```
' Visual Basic 6.0
Dim a(10) As Integer
Erase a

' After upgrade to Visual Basic .NET
Dim a(10) As Short
' UPGRADE_NOTE: Erase was upgraded to System.Array.Clear.
System.Array.Clear(a, 0, a.Length)
```

> **Note** In some cases, a variable has to be renamed because the original name conflicts with a reserved word. For example, a variable named Day will be renamed Day_Renamed; Day was not a reserved name in Visual Basic 6.0, but it is in Visual Basic .NET.

What to do next

- No further action is necessary. This note was added to make you aware of the changes to your code, because the code looks significantly different.

See Also

Language Changes in Visual Basic

VisualBasic.Compatibility Namespace Reference

Functions and objects in the **VisualBasic.Compatibility** namespace are provided for use by the Visual Basic 6.0 to Visual Basic .NET upgrade tools. In most cases they duplicate functionality that can be achieved using other parts of the .NET Framework; they are only necessary where the Visual Basic 6.0 code model differs significantly from the .NET implementation.

The following tables list functions and objects in the **VisualBasic.Compatibility** namespace. Additional functions and objects related to the **Microsoft.Compatibility.Data** namespace are not listed; they are the equivalents of the ADO data binding in Visual Basic 6.0.

VisualBasic.Compatibility Functions

Function	Description
CopyArray	Duplicates Visual Basic 6.0 functionality of assigning an array to a Variant.
CursorToIPicture	Used in format conversion.
Eqv	Duplicates the Visual Basic 6.0 **Eqv** (equivalence) operator.
FontChangeBold	Used in converting Visual Basic 6.0 fonts to Visual Basic .NET **Font** objects.
FontChangeGdiCharSet	Used in converting Visual Basic 6.0 fonts to Visual Basic .NET **Font** objects.
FontChangeItalic	Used in converting Visual Basic 6.0 fonts to Visual Basic .NET **Font** objects.
FontChangeName	Used in converting Visual Basic 6.0 fonts to Visual Basic .NET **Font** objects.
FontChangeSize	Used in converting Visual Basic 6.0 fonts to Visual Basic .NET **Font** objects.
FontChangeStrikeout	Used in converting Visual Basic 6.0 fonts to Visual Basic .NET **Font** objects.
FontChangeUnderline	Used in converting Visual Basic 6.0 fonts to Visual Basic .NET **Font** objects.
FontToIFont	Used in format conversion.
Format	Duplicates the Visual Basic 6.0 **Format** function.
FromPixelsX	Used for coordinate conversion.
FromPixelsY	Used for coordinate conversion.
FromPixelsUserHeight	Used for coordinate conversion.
FromPixelsUserWidth	Used for coordinate conversion.
FromPixelsUserX	Used for coordinate conversion.
FromPixelsUserY	Used for coordinate conversion.
GetActiveControl	Duplicates the Visual Basic 6.0 **Screen.ActiveControl** property.
GetCancel	Duplicates the functionality of the **Cancel** property of a Visual Basic 6.0 **CommandButton**.
GetDefault	Duplicates the functionality of the **Default** property of a Visual Basic 6.0 **CommandButton**.
GetEXEName	Duplicates the Visual Basic 6.0 **App.EXEName** property.
GetHInstance	Duplicates the Visual Basic 6.0 **App.HInstance** property.
GetItemData	Duplicates the functionality of the **ItemData** property of a Visual Basic 6.0 **ListBox** or **ComboBox**.
GetItemString	Duplicates the functionality of retrieving a value from a list in a Visual Basic 6.0 **ListBox** or **ComboBox**.
GetPath	Duplicates the Visual Basic 6.0 **App.Path** property.
IconToIpicture	Used in format conversion.
IFontToFont	Used in format conversion.
ImageToIPicture	Used in format conversion.
ImageToIPictureDisp	Used in format conversion.
Imp	Duplicates the Visual Basic 6.0 **Imp** (implication) operator.
IPictureDispToImage	Used in format conversion.
IPictureToImage	Used in format conversion.
LoadResData	Duplicates the Visual Basic 6.0 **LoadResData** function.
LoadResPicture	Duplicates the Visual Basic 6.0 **LoadResPicture** function.
LoadResString	Duplicates the Visual Basic 6.0 **LoadResString** function.
PixelsToTwipsX	Used for coordinate conversion.
PixelsToTwipsY	Used for coordinate conversion.
SendKeys	Duplicates the Visual Basic 6.0 **SendKeys** function.
SetCancel	Duplicates the functionality of the **Cancel** property of a Visual Basic 6.0 **CommandButton**.
SetDefault	Duplicates the functionality of the **Default** property of a Visual Basic 6.0 **CommandButton**.
SetItemData	Duplicates the functionality of the **ItemData** property of a Visual Basic 6.0 **ListBox** or **ComboBox**.
SetItemString	Duplicates the functionality of setting a value in a list in a Visual Basic 6.0 **ListBox** or **ComboBox**.

SetResourceBaseName	Used with **LoadRes** functions when a resource file uses a non-standard naming convention.
ShowForm	Duplicates the functionality of the **Show** method of a Visual Basic 6.0 form.
TabLayout	Used when converting **Debug.Print** statements that contain the **Tab** or **Spc** keywords.
ToPixelsX	Used for coordinate conversion.
ToPixelsY	Used for coordinate conversion.
ToPixelsUserHeight	Used for coordinate conversion.
ToPixelsUserWidth	Used for coordinate conversion.
ToPixelsUserX	Used for coordinate conversion.
ToPixelsUserY	Used for coordinate conversion.
TwipsPerPixelX	Duplicates the Visual Basic **Screen.TwipsPerPixelX** property.
TwipsPerPixelY	Duplicates the Visual Basic **Screen.TwipsPerPixelY** property.
TwipsToPixelsX	Used for coordinate conversion.
TwipsToPixelsY	Used for coordinate conversion.
ValidateControls	Duplicates the functionality of the **ValidateControls** method of a Visual Basic 6.0 form.
WhatsThisMode	Duplicates the functionality of the **WhatsThisMode** method of a Visual Basic 6.0 form.
Zorder	Used for converting the z-order of controls.

VisualBasic.Compatibility Objects

Object	Description
BaseControlArray	Parent class for Visual Basic 6.0 control array emulation.
BaseOcxArray	Parent class for emulated arrays of ActiveX controls.
ButtonArray	Emulates a Visual Basic 6.0 control array of **CommandButton** controls.
CheckBoxArray	Emulates a Visual Basic 6.0 control array of **CheckBox** controls.
CheckedListBoxArray	Emulates a Visual Basic 6.0 control array of **ListBox** controls with the **Style** property set to Checked.
ComboBoxArray	Emulates a Visual Basic 6.0 control array of **ComboBox** controls.
DirListBox	Emulates a Visual Basic 6.0 **DirListBox** control.
DirListBoxArray	Emulates a Visual Basic 6.0 control array of **DirListBox** controls.
DriveListBox	Emulates a Visual Basic 6.0 **DriveListBox** control.
DriveListBoxArray	Emulates a Visual Basic 6.0 control array of **DriveListBox** controls.
FileListBox	Emulates a Visual Basic 6.0 **FileListBox** control.
FileListBoxArray	Emulates a Visual Basic 6.0 control array of **FileListBox** controls.
FixedLengthString	Emulates Visual Basic 6.0 fixed-length strings.
GroupBoxArray	Emulates a Visual Basic 6.0 control array of **Frame** controls.
HScrollBarArray	Emulates a Visual Basic 6.0 control array of **HScrollBar** controls.
LabelArray	Emulates a Visual Basic 6.0 control array of **Label** controls.
ListBoxArray	Emulates a Visual Basic 6.0 control array of **ListBox** controls.
ListBoxItem	Emulates the **ItemData** property of a **ListBox** control.
MenuItemArray	Emulates a Visual Basic 6.0 control array of **Menu** controls.
PanelArray	Emulates a Visual Basic 6.0 control array of **PictureBox** controls that contain child controls.
PictureBoxArray	Emulates a Visual Basic 6.0 control array of **PictureBox** controls.
RadioButtonArray	Emulates a Visual Basic 6.0 control array of **OptionButton** controls.
TabControlArray	Emulates a Visual Basic 6.0 control array of **TabStrip** controls.
TextBoxArray	Emulates a Visual Basic 6.0 control array of **TextBox** controls.
TimerArray	Emulates a Visual Basic 6.0 control array of **Timer** controls.
VscrollBarArray	Emulates a Visual Basic 6.0 control array of **VScrollBar** controls.

See Also

Visual Basic 6.0 Compatibility Library | Introduction to Visual Basic .NET for Visual Basic Veterans

Converting Visual J++ Applications to Visual C#

Microsoft® Visual Studio® .NET provides the ability to convert projects created in Visual J++® version 6.0 or written in the Java language to Visual C#™ so that you can take advantage of the .NET Framework. Java Language Conversion Assistant generates a new Visual C# project from an existing Visual J++ 6.0 project or Java-language files. The Java Language Conversion Assistant wizard makes it easy to convert your existing files.

In This Section

Converting Visual J++ or Java-Language Projects to Visual C#
Describes how to use Java Language Conversion Assistant.
Java Language Conversion Assistant
Provides an overview of upgrading a Visual J++ 6.0 project or Java-language files to Visual C#.
Java Language Conversion Assistant Wizard
Describes how to use the wizard.
Manually Upgrading Unconverted Code
Explains how to upgrade code that could not be converted automatically.

Related Sections

What's New in Java Language Conversion Assistant
Describes the new features available in this version of Java Language Conversion Assistant.
Converting Web Applications
Describes how to adapt Web applications when converting them with Java Language Conversion Assistant.
Conversion of JSP Custom Tag Libraries
Provides a list of Java-language classes for which support classes are created to emulate custom tag runtime behavior.
Compile Converted Servlet Classes Before Opening Them
Explains how to open Web Forms for converted servlet classes.
Matching Code Pages
Describes what to do when you have missing or mismatched code pages.
JLCA Diagnostic Messages Listed by Package
Provides lists of error messages generated by Java Language Conversion Assistant.
Visual C# Language
Provides an overview of the Visual C# language.
C# Tutorials
Provides a link to tutorials designed to introduce a variety of Visual C# topics.

Converting Visual J++ or Java-Language Projects to Visual C#

Use Java Language Conversion Assistant to convert Visual J++ 6.0 projects and Java-language files for further development in Visual C#. Java Language Conversion Assistant does not modify your existing project, but rather creates a new Visual C# project based upon the original project. Any errors, warnings, or issues generated in the course of conversion are displayed in a conversion report after the new project has been generated. These errors, warnings, and issues are also noted in comments in the converted code to help you manually convert the parts of the project that cannot be converted automatically.

> **Note** If you are new to Visual C#, take some time to familiarize yourself with the language before attempting this process. For more information, see C# Language Tour.

Visual J++ Projects

To convert a Visual J++ project

1. Start Visual Studio .NET.
2. On the **File** menu, point to **Open**, and click **Convert**.
3. Select **Java Language Conversion Assistant**, and click **OK**.
4. On the **Source Files** page, select **Visual J++ 6.0 project** .
5. On the **Select a project file** page, click **Browse**.
6. Browse to the correct .vjp file, and select it.

   > **Note** If the CLASSPATH directive of your .vjp file specifies .jar or .class files, they are ignored.

7. On the **Specify a directory for your new project** page, specify the name and directory of the new project to be created.
8. On the **Begin conversion** page, click **Next**.
9. Fix code that could not be converted automatically. For more information, see Manually Upgrading Unconverted Code.

Java-Language Projects

To convert a Java-language project

1. Start Visual Studio .NET
2. On the **File** menu, point to **Open**, and click **Convert**.
3. Select **Java Language Conversion Assistant**, and click **OK**.
4. On the **Source Files** page, click **A directory containing the project's files**.
5. On the **Select source directory** page, click **Browse**.
6. Browse to the correct project, and select it.

   > **Note** You will not see the files in the directory you select, but all .jav and .java files in it will be converted. All other files in the directory are ignored.

7. On the **Configure your new project** page, specify the following:

   - Name of the project to be created.
   - Directory in which the any additional files for your project are located.

     > **Note** You will not see the files in the directory you select, but all .jav and .java files in it will be converted. All other files in the directory are ignored.

   - Output type of the project.

8. On the **Specify a directory for your new project** page, specify the name and directory of the new project to be created.
9. On the **Begin your conversion** page, click **Next**.
10. Fix code that could not be converted automatically. For more information, see Manually Upgrading Unconverted Code.

See Also

Java Language Conversion Assistant | Manually Upgrading Unconverted Code

Java Language Conversion Assistant

Java Language Conversion Assistant is a tool that converts Visual J++ 6.0 projects and Java-language files to Visual C#. By converting these files to Visual C#, you can leverage your existing code base and take advantage of the benefits of the .NET Framework.

The new Visual C# project contains all the new Visual C# code that can be generated automatically from the existing Visual J++ or Java-language code. For more information, see Converting Visual J++ or Java-language Projects to Visual C#.

You can use Java Language Conversion Assistant to convert Visual J++ or Java-language applications and applet projects. They are converted in the following way:

Before Conversion	After Conversion
Applications	Windows Forms applications
Applets	Web user controls

You can host converted Web user controls in a browser just as you would an applet. Hosted controls are declared in HTML pages with the **OBJECT** tag, instead of the **APPLET** tag. Use the **classid** attribute to identify the control by specifying the path to the control and the fully qualified name of the control, separated by the pound sign (#), as shown in the following example:

```
<OBJECT id="myControl" classid="http:ControlLibrary1.dll#ControlLibrary1.myControl" VIEWASTEX
T></OBJECT>
```

For the control to be displayed properly, the .dll file containing the control must either be in the same virtual directory as the Web page that is displaying it or be installed in the global assembly cache.

Support Classes

To convert functionality in the original project that is not available in Visual C#, Java Language Conversion Assistant creates *support classes* (also called *managers*) that duplicate the original functionality. Support classes are sometimes substantially different architecturally from the classes they emulate. Although every effort is made to preserve the original architecture of your application in the converted project, the primary goal of these support classes is to duplicate the original functionality.

Conversion Report

There might be some code in your project that could not be converted automatically. After you run the Java Language Conversion Assistant wizard, you can view the conversion report that details all errors, warnings, and issues encountered during the conversion process. Unconverted code is noted in the code of the new project by comments labeled UPGRADE_TODO. You can view conversion comments on the task list. Each conversion comment contains a link to a Help topic about how to convert that code manually. For more information, see Manually Upgrading Unconverted Code.

See Also

Converting Visual J++ and Java-Language Projects to Visual C# | Manually Upgrading Unconverted Code |
Converting Visual J++ Applications to Visual C#

Java Language Conversion Assistant Wizard

Use the Java Language Conversion Assistant wizard to convert Visual J++ and Java-language projects to Visual C#. With the wizard, you can create a new project and copy files from your original project for conversion from Visual J++ or the Java language to Visual C#. A report detailing any errors, warnings, or issues encountered in the conversion process is generated. Errors, warnings, and issues are also noted as comments in the code of the new project and can be viewed on the task list.

To access the Java Language Conversion Assistant wizard

1. On the **File** menu, point to **Open**, and click **Convert**.
2. Click **Java Language Conversion Assistant**, and click **OK**.

See Also

Java Language Conversion Assistant | Converting Visual J++ or Java-Language Projects to Visual C# | Manually Upgrading Unconverted Code

Manually Upgrading Unconverted Code

After Java Language Conversion Assistant converts your Visual J++ project or Java-language files, the new Visual C# project might contain code that could not be converted automatically. Comments are inserted in the code of the new project to help you convert that code to Visual C# manually.

Comment Type	Description
UPGRADE_TODO	Code that could not be converted automatically
UPGRADE_WARNING	Code that might be problematic
UPGRADE_NOTE	Code that might exhibit a different behavior from the original code

There might also be some compiler errors that must be corrected before the application compiles. Each comment contains a brief description of the issue and a link to a Help topic about how to convert the code.

To upgrade unconverted code manually

1. After running the Java Language Conversion Assistant wizard, open your project in Visual Studio .NET.
2. On the **View** menu, select **Show Tasks**, and then either **Comment** or **All**.

   The conversion notes are displayed on the **Task List**.

3. On the **Task List**, double-click the first conversion note.

   The focus moves to the location of that comment in the code.

4. Examine the note and the code to which it pertains. For more information on correcting that error manually, click the link in the conversion note to display a Help topic for that error.

   Some of the conversion notes in the converted code for JSP pages are not clickable. You can cut and paste the note in the Help **Search** box. For a list of errors, warnings, and issues, see Java Language Errors, Warnings, and Issues. For a list of Help topics for JSP classes and methods that cannot be converted automatically, see Javax.servlet.jsp Error Messages.

5. Upgrade the code manually.
6. Repeat Steps 3 to 5 until all conversion notes have been addressed.

See Also

Java Language Conversion Assistant | Converting Visual J++ or Java-Language Projects to Visual C#

What's New in Java Language Conversion Assistant

The following features are included with this version of Java Language Conversion Assistant:

- Support for JavaServer Pages (JSP)
- Servlet support
- Improved applet support
- Support for different character-encoding systems
- Support for collections
- Performance enhancements

JavaServer Pages

JavaServer Pages can now be converted to ASP.NET applications to develop information-rich Web applications that are easy to maintain. Development time is reduced, and existing business systems can be tapped for database and other support. You can change the page layout without affecting the dynamically generated content. Tag handlers are converted to Web user controls.

Servlet Support

Java Language Conversion Assistant now converts servlet classes that allow you to extend the functionality of your Web server and interact with your existing business systems. These are platform-independent, light-weight applications that are very much like applets, but run on the server side. They can access enterprise databases through ODBC and are ideal for building interactive Web applications.

Improved Applet Support

Java Language Conversion Assistant includes additional support for converting applets to Windows Forms user controls. Applets can now take advantage of additional .NET functionality while maintaining their behavior.

Support for Different Character-Encoding Systems

Encoding support has been expanded to include all the major character-encoding systems. Files that are encoded in the current system ANSI code page keep this encoding after conversion. Any other encoding is converted to UTF-8 encoding.

The following encoding systems are currently supported:

- Windows system ANSI code pages
- UTF-8 Unicode
- UTF-16 Unicode
- ISO Latin alphabet No. 2 (Central European)
- ISO Latin alphabet No. 3
- ISO Latin alphabet No. 4 (Baltic States)
- ISO Latin/Arabic alphabet
- ISO Latin/Cyrillic alphabet
- ISO Latin/Greek alphabet
- ISO Latin/Hebrew alphabet
- ISO Latin alphabet No. 9
- EUC Japanese
- EUC Korean
- EUC Chinese Simplified
- Chinese National Standard

For issues with matching character encodings, see Matching Code Pages.

Support for Collections

Java Language Conversion Assistant now offers support for a unified approach to handling and manipulating collections. The Java 1.2 Collections API can be converted to the classes in the **System.Collections** namespace, which present methods to represent and use collections to reduce development effort and increase productivity.

Performance Enhancements

Overall performance for large projects has been improved dramatically in this version of Java Language Conversion Assistant. Conversion time has been reduced 70 percent for sample projects of about 25 MB (580+ lines of code). Conversion time for medium-sized projects (about 250 KB) has been reduced by 50 percent.

Converting Web Applications

When converting Web applications from JSP to ASP.NET, several additional steps are necessary. Java Language Conversion Assistant establishes the application root directory, moves servlet classes to a different directory, replaces links, and resolves tag library descriptors defined in the Web.xml file. You must set the context path and domain parameters for your Web application.

In This Section

Establishing the Directory Structure for a Web Application
Explains how to establish the directory structure for your Web application.
Link Replacement
Describes how Java Language Conversion Assistant replaces links in your converted Web application.
Resolution of Tag Library Descriptors
Describes the conversion of tag library descriptors included in the Web.xml file.

Establishing the Directory Structure for a Web Application

When converting a Web application, Java Language Conversion Assistant establishes a different directory structure for the new application. You must add two parameters containing the context path and domain on which your application will run.

To establish the directory structure

1. Add a parameter called *ContextPath* at the command line.

   This is the starting point for all URL references in the Web application.

   `jconvert.\projecttype ASP.NET\ContextPath AppName`

2. Add a parameter called *Domain* at the command line to provide the host name in applications with an empty context path.

   This parameter is necessary to avoid replacement of links to servlets or JSP pages outside the current application. If no host name is provided, all absolute links are processed. It is not necessary to include **http://**.

   ```
   jconvert.\projecttype ASP.NET\Domain www.mydomain.com
   ```

Any servlet classes are moved to the output directory (the virtual root). The directory structure created in the process of conversion does not necessarily correspond to the directory in which the servlet is located.

Example

In the following example, the Web application root directory is \website\AppName. The servlet classes are moved to a different directory in the converted directory structure.

Original Directory Structure

```
mt\src
mt\src\com\ais\servlets
mt\src\com\ais\servlets\Servlet1.java
mt\src\com\ais\servlets\Servlet2.java
mt\src\com\ais\beans
mt\src\com\ais\beans\CatalogBean.java
mt\src\com\ais\customtags
mt\src\com\ais\customtags\TableTag.java
mt\misc\pack1\myclasses
mt\misc\pack1\myclasses\DbHelper.java
mt\website
mt\website\AppName
mt\website\AppName\jsp\
mt\website\AppName\jsp\Search.jsp
mt\website\AppName\jsp\content\Catalog.jsp
mt\website\AppName\html\Index.html
mt\website\AppName\images\Logo.gif
mt\website\AppName\WEB-INF
mt\website\AppName\WEB-INF\Web.xml
mt\website\AppName\WEB-INF\MyTags.tld
```

Equivalent Converted Directory

```
outdir\src
outdir\src\com\ais\beans
outdir\src\com\ais\beans\CatalogBean.cs
outdir\src\com\ais\customtags
outdir\src\com\ais\customtags\TableTag.cs
outdir\misc\pack1\myclasses
outdir\misc\pakc1\myclasses\Dbhelper.cs
outdir\website
outdir\website\AppName
outdir\website\AppName\com\ais\servlets\Servlet1.aspx
outdir\website\AppName\com\ais\servlets\Servlet2.aspx
```

```
outdir\website\AppName\jsp\
outdir\website\AppName\jsp\Search.aspx
outdir\website\AppName\jsp\content\Catalog.aspx
outdir\website\AppName\html\Index.html
outdir\website\AppName\images\Logo.gif
outdir\website\AppName\WEB-INF
outdir\website\AppName\WEB-INF\Web.xml
outdir\website\AppName\WEB-INF\MyTags.tld
```

Link Replacement

Java Language Conversion Assistant replaces all links to work with the converted directory structure. This section assumes the directory structure defined in Establishing the Directory Structure for a Web Application.

There are basically three types of URLs:

- Absolute URLs with domain information.

```
http://www.somehost.com/servlet/pack1/Servlet1
```

- Absolute URLs without domain information.

```
/servlet/pack1/Servlet1
```

- Relative URLs.

```
pack1.Servlet1
```

The URL that a client uses to access a Web application has the following form:

```
http://hoststring/ContextPath/ServletPath/pathInfo.
```

In this example, the elements are defined as follows:

hoststring
Host name that is mapped either to a virtual host or to *hostname*:*portNumber*.
ContextPath
Name of the Web application. This can be empty.
ServletPath
Servlet that is mapped to the servlet path.
pathInfo
Remaining portion of the URL, typically the file name.

Java Language Conversion Assistant makes the following changes:

- Changes the context path (if present) to the name of the corresponding virtual root defined in the conversion, plus the Web application root.
- If the context path is not present, the VROOT is inserted in the link.
- If the file extension is .jsp, it is changed to .aspx.

Example

Original URL

```
http://hoststring/ContextPath//dir/JSP1.jsp?param1=1
http://HostNameProvided/dir/JSP1.jsp?param1=1
```

Equivalent Converted URL

```
http://hoststring/VRoot/website/AppName/dir/JSP1.aspx?param1=1
http://HostNameProvided/VRoot/website/AppName/dir/JSP1.aspx?param1=1
```

If the link contains the servlet token and the complete name of a servlet, the servlet token is eliminated, and dots (.) in the complete name are changed to slashes (/). If the link to the servlet is relative, the reference is changed to include the VROOT and the converted servlet directory structure.

Example

Original URL

```
http://hoststring/ContextPath/servlet/com.ais.servlets.Servlet1
com.ais.servlets.Servlet1
```

Equivalent Converted URL

```
http://hoststring/VRoot/com/ais/servlets/Servlet1.aspx
/VRoot/com/ais/servlets/Servlet1.aspx
```

If the servlet token is followed by a servlet name found in the Web.xml file, the complete name of the servlet is inserted in the link.

Example

Original XML

```
<servlet>
    <servlet-name>test</servlet-name>
    <servlet-class>com.ais.servlets.Servlet1</servlet-class>
</servlet>
```

Original URL

```
http://hoststring/ContextPath/servlets/test
```

Equivalent Converted URL

```
http://hoststring/VRoot/com/ais/servlets/Servlet1.aspx
```

If the Web application name is followed by a pattern defined in the servlet mapping in Web.xml, the name of the servlet is determined and inserted in the link. The pattern can contain wildcards (*).

Example

Original XML

```
<servlet>
    <servlet-name>test</servlet-name>
    <servlet-class>com.ais.servlets.Servlet1</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>test</servlet-name>
    <url-pattern>/*.asp</url-pattern>
</servlet-mapping>
```

Original URL

```
http://hoststring/ContextPath/abc.asp
```

Equivalent Converted URL

```
http://hoststring/VRoot/com/ais/servlets/Servlet1.aspx
```

Links calculated at run time are not converted by Java Language Conversion Assistant. Instead, you must manually add the necessary parameters. This applies to the following cases:

- Links with the expressions <%=%>, jsp:forward, jsp:include, or jsp:execute and the XML tags <a href> and <form action>.
- Any **javax.servlet** method whose parameters represent a path to a Web resource, for example, the **Response.redirect** method.

Example

Original Java-Language Code

```
<jsp:include page="<%=value%>">
```

Equivalent Visual C# Code

```
<%-- //UPGRADE_TODO: Expected format of parameters of action jsp:include are different in the
equivalent in .NET --%>
<% Server.Execute(value)%>
```

Resolution of Tag Library Descriptors

When the **uri** attribute of the **taglib** directive refers to a **taglib-uri** defined in the Web.xml file, the location of the tag library descriptor (TLD) is taken from that file.

Example

Original XML

```
<taglib>
    <taglib-uri>/MYTEST</taglib-uri>
    <taglib-location>WEB-INF/myTagLib.tld</taglib-location>
</taglib>
```

Original JSP

```
<%taglib uri="/MYTEST" prefix="myTag"%>
```

Equivalent TLD to be parsed

```
/WEB-INF/myTagLib.tld
```

Conversion of JSP Custom Tag Libraries

Due to the differences between JSP custom tags and .NET Web user controls, a set of support classes that emulate JSP custom tag processing are created during conversion. Those classes are the base of converted JSP custom tag class handlers. Support classes are created for the following classes in the **javax.servlet.jsp.tagext** package:

Original Java-Language Class	Generated Support Class
BodyContent	**WCBodyContent**
Tag	**WCBase**
IterationTag	**WCIterationBase**
BodyTag	**WCBodyBase**
BodyTagSupport	**WCBodyImpl**
TagSupport	**WCIterationImpl**

Compile Converted Servlet Classes Before Opening Them

When converting servlet classes, you must resolve all conversion issues and compile your code before you can open a Web Forms page for the application. The Web Forms page cannot be opened in the designer view until the code is compiled. If you try to open it before compiling, you receive an error.

Matching Code Pages

If you find that your code after conversion has strange characters, the most likely cause is a missing or mismatched code page for character encoding. Java Language Conversion Assistant matches encoding in the following manner:

- Files that are encoded in the current system ANSI code page keep this encoding after conversion.
- Any other encoding is converted to UTF-8 encoding.

Unicode files that start with a Unicode byte-order mark are identified automatically as Unicode. If a file does not start with a byte-order mark and the encoding switch is not specified, the file is assumed to be in the current ANSI code page of the system.

To change the system ANSI code page you are currently using

1. Go to **Control Panel**, and double-click **Regional Options** (in Windows 2000) or **Regional and Language Options** (in Windows XP).
2. Click **Advanced**, and select the desired code page.

If your source files are in a non-ANSI character encoding or you have Unicode source files that do not start with a byte-order mark, you must use the encoding switch.

The following character encodings can be specified in the encoding switch:

Encoding	Switch
Latin alphabet No. 2 (Central European)	ISO-8859-2
Latin alphabet No. 3	ISO-8859-3
Latin alphabet No. 4 (Baltic States)	ISO-8859-4
Latin/Cyrillic alphabet	ISO-8859-5
Latin/Arabic alphabet	ISO-8859-6
Latin/Greek alphabet	ISO-8859-7
Latin/Hebrew alphabet	ISO-8859-8
Latin alphabet No. 9	ISO-8859-15
Japanese	EUC-JP
Korean	EUC-KR
Chinese Simplified	EUC-CN
Chinese National Standard	GB18030
UTF-8 without byte-order mark	UTF-8

To use a specific encoding, you must have support for it installed on your system. See your system documentation for details. When using the encoding switch, all files in the project must be in the same encoding.

If you are using an encoding system that is not supported in JLCA, you might lose characters that cannot be converted. In some cases, a file might be lost.

JLCA Diagnostic Messages Listed by Package

This section contains links to all the diagnostic messages for the Java Language Conversion Assistant. The pages containing the links are organized by package.

Com.ms.activex Error Messages

Com.ms.awt Error Messages

Com.ms.com Error Messages

Com.ms.directX Error Messages

Com.ms.dll Error Messages

Com.ms.dxmedia Error Messages

Com.ms.fx Error Messages

Com.ms.io Error Messages

Com.ms.jdbc.odbc Error Messages

Com.ms.lang Error Messages

Com.ms.mtx Error Messages

Com.ms.object Error Messages

Com.ms.ui Error Messages

Com.ms.util Error Messages

Com.ms.wfc.app Error Messages

Com.ms.wfc.ax Error Messages

Com.ms.wfc.core Error Messages

Com.ms.wfc.data.adodb Error Messages

Com.ms.wfc.data Error Messages

Com.ms.wfc.data.ui Error Messages

Com.ms.wfc.io Error Messages

Com.ms.wfc.ole32 Error Messages

Com.ms.wfc.ui Error Messages

Com.ms.wfc.util Error Messages

Com.ms.wfc.win32 Error Messages

Com.ms.win32 Error Messages

Java Language Errors, Warnings, and Issues

Java.applet Error Messages

Java.awt Error Messages

Java.beans Error Messages

Java.io Error Messages

Java.lang Error Messages

Java.math Error Messages

Java.net Error Messages

Java.security Error Messages

Java.sql Error Messages

Java.text Error Messages

Java.text.resources Error Messages

Java.util Error Messages

Javax.servlet.http Error Messages

Javax.servlet Error Messages

Javax.servlet.jsp Error Messages

Javax.swing Error Messages

Java Language Errors, Warnings, and Issues

(1002) Compile Error: Invalid Statements

(1003) Note: Declaration changed from Final to Variable

(1005) Note: Initialize expression was moved to method or constructor

(1011) Note: New code element generated

(1012) Note: Break statement with label is not supported

(1014) Note: Labeled statement moved by Java Language Conversion Assistant

(1015) Note: Continue statement with label is not supported

(1019) Note: Inner classes not provided with enclosing instance

(1021) Note: Inner class member field this$0 is not supported

(1022) Note: Local class declaration is not supported

(1023) Note: Final variable copied to inner class as member field

(1024) Global Note: Anonymous class declaration is not supported

(1025) Note: Interfaces nested within interfaces are not allowed in C#

(1027) Note: Synchronized methods are not supported in .NET

(1042) Global Warning: Narrowing conversions for primitive types may generate runtime errors

(1043) To Do: Method or property returns a different value in the .NET Framework

(1045) Note: Interfaces cannot contain fields in the .NET Framework

(1062) To Do: NULL value of database columns can be checked by using OleDbDataReader.isDBNull

(1063) To Do: Modify connection string(s) to match the .NET Framework format

(1064) To Do: Modify the string connection to match the .NET Framework format and add the java.util.Properties information to it

(1066) To Do: setLoginTimeout must be a parameter to connection string for OleDbConnection constructor

(1075) Note: Font constructor may generate non-equivalent instances

(1077) To Do: Inheritance for class not supported

(1078) To Do: Make sure that resources used in this class are valid resources files

(1079) Compile Error: All the methods inherited from an abstract class must be implemented

(1083) Note: Bit arrays must be of the same size to allow the operation

(1086) Compile Error: Field equivalent cannot be assigned to an integer variable

(1088) To Do: There is no NULL value defined for a .NET structure

(1089) To Do: Access and file format of configuration settings are different

(1091) Runtime Warning: The equivalent constructor has a different number of parameters

(1092) To Do: Expected value or type of a parameter is different

(1093) To Do: Change code to access value of a .NET class member

(1095) To Do: Class element not mapped in the current release

(1096) Note: Fields mapped to enumerations initialized with integer values

(1099) Note: Exceptions thrown by method may be different

(1100) Note: Catch clauses in a try statement with the same exception

(1101) Runtime Warning: Mapped parameters may throw an exception

(1102) Compile Error: Class must be modified to call superclass

(1103) Compile Error: Field has a different type

(1104) To Do: Use ConnectionString property to set logon and password

(1105) To Do: Method invocation is not necessary

(1107) Global Note: Migration of setAllowAddNew, setAllowDelete, and setAllowUpdate of the DataGrid class

(1108) To Do: Property or method must be set or called before an invocation

(1109) Compile Error: Migrated parameter not compatible with parameter type of method

(1113) Compile Error: Member is reserved for internal use of its package

(1114) To Do: Parameter of setFilter in the OpenFileDialog and SaveFileDialog classes must be changed

(1115) Global Warning: Unsafe code requires a compiler switch to compile

(1116) To Do: COM Custom marshaller may become invalid

(1118) To Do: Check boxes do not provide mutual exclusion when grouped

(1119) Note: User control event attribute not valid

(1123) Note: Class maps to interface

(1124) Note: Code was removed from method body

(1128) Compile Error: Break statement at finally clause level is invalid

(1130) To Do: Equivalent of java.sql.DatabaseMetaData handles query results in System.Data.DataTable

(1132) To Do: Equivalent class does not throw exception for malformed URL

(1135) Global Error: The System.Windows.Forms.Application.Run method must be called for the main window

(1137) To Do: User get or set method converted to property has name collisions

(1139) Global Warning: Directory was not converted

(1140) Global Warning: Thread name can be set just once

(1141) Note: Java Language Conversion Assistant commented code from Java Callable Wrapper class for a COM coclass

(1142) To Do: The event method needs be overloaded

(1143) To Do: Method is not an override method

(1144) Runtime Warning: Add extra logic into componentHidden

(1145) Runtime Warning: Add extra logic into componentShown

(1146) To Do: Declaration and construction for class instance in different lines cannot be converted

(1147) To Do: The <classname> class is marked as sealed

(1148) Note: Synchronized keyword has been removed

(1151) To Do: Inaccesible constructor

(1152) To Do: You must manually migrate access to the registry

(1153) To Do: Untranslated statement

(1154) Note: Parameter to constructor of equivalent of class com.ms.wfc.data.ui.DataNavigator

(1155) To Do: Class that derives from Delegate or MulticastDelegate

(1156) Compile Error: Wrong statements

(1157) Compile Error: Method with the same parameter types

(1158) To Do: The way of accessing the file was modified

(1159) Global Warning: Path of archives that depend on the working directory

(1160) To Do: One interface includes the functionality of another one

(1167) Note: getParameter calls without a string literal

(1168) To Do: getParameter that requires rename of parameters

(1169) Note: The default value of property has been changed

(1170) Global Warning: Class mapped to a struct; cannot contain or return a null value

(1171) Note: getSource method must be in an event handling routine

(1172) Global Error: Input project file is corrupt

(1173) To Do: Hexadecimal literal returns a different value

(1174) Note: The form to load a binary file in .NET is different

(1175) Global Warning: Unsigned long detected

(1177) Runtime Warning: Implemenentation of interface was converted to a single class

(1178) To Do: Method needs to be modified

(1179) Global Warning: Request the OleDbManager connection string

(1180) Note: Only one transactional attribute is supported

(1181) Runtime Warning: Expressions used more than once in C# code

(1182) To Do: Session parameters are stored in a different order

(1183) Runtime Warning: Init functionality has been converted

(1184) To Do: <ClassName> requires changing the type of the receiving instance

(1185) To Do: Queue access is obtained differently

(1186) Compile Error: Class hierarchy differences may cause compile errors

(1187) Runtime Warning: <Member type> may be executed every time the ASP.NET page loads

(1191) Note: Instance field was converted to static

(1195) To Do: Change in override modifier for nonvirtual method

(1197) To Do: Delegate instantiated with String value for function name

(1198) Note: Inheritance of servlet classes can cause wrong manipulation of variables

(1200) Note: Base class shared fields have been redeclared with the new modifier

(1202) To Do: Reference conversion needs user modification

(1203) Note: Using the class interface to expose public members

(1204) Note: Access modifiers were changed

(1205) To Do: Methods with parameters of primitive type overloaded with parameters of wrapper class type

(1206) Global Note: Converted project must be compiled before opening in Visual Studio designer view

(1207) To Do: Types extending from COM objects should override all methods

(1209) Note: Method was not converted to property

(1220) To Do: Reset method must be implemented

(1221) To Do: Parent invocations in methods from applets

(1223) Global Warning: Output of bidirectional languages

(1224) To Do: Method returns different type

(2001) Runtime Warning: Some server-side tags that must be unique could be repeated when including files

(2002) Runtime Warning: Dynamic file inclusion is not allowed in Aspx

(2003) Note: Attribute "buffer" can have only two valid values in ASP.NET

(2004) Note: Request scope was changed to Session scope

(2005) Global Error: Web.Config customErrors Mode property must be set to "On"

(2006) To Do: File will not be converted

(2007) Note: Script variables can be out of scope when moving blocks inside custom tags to methods

(4013) Note: Specified TLD file could not be found or parsed

(4015) Note: User control ID has been changed

(5000) To Do: Method is not marked as virtual

Com.ms.activex Error Messages

com.ms.activeX could not be converted

Com.ms.awt Error Messages

com.ms.awt.AccessibleWrapper could not be converted

com.ms.awt.AWTFinalizeable could not be converted

com.ms.awt.AWTFinalizer could not be converted

com.ms.awt.AWTPermission.check could not be converted

com.ms.awt.AWTPermission.pid could not be converted

com.ms.awt.CaretX could not be converted

com.ms.awt.CharsetString could not be converted

com.ms.awt.CharToByteSymbol could not be converted

com.ms.awt.ColorX.getHilight could not be converted

com.ms.awt.ColorX.getShadow could not be converted

com.ms.awt.Device.Device could not be converted

com.ms.awt.Device.getBasics could not be converted

com.ms.awt.Device.getDisplayContext could not be converted

com.ms.awt.DrawingSurface could not be converted

com.ms.awt.DrawingSurfaceInfo could not be converted

com.ms.awt.EventFilterListener could not be converted

com.ms.awt.FocusEvent.getOtherComponent could not be converted

com.ms.awt.FocusingTextField could not be converted

com.ms.awt.FontDescriptor could not be converted

com.ms.awt.FontMetricsX.bytesWidth could not be converted

com.ms.awt.FontMetricsX.CHAR_KERNING could not be converted

com.ms.awt.FontMetricsX.charsWidth could not be converted

com.ms.awt.FontMetricsX.FontMetricsX could not be converted

com.ms.awt.FontMetricsX.getAveCharWidth could not be converted

com.ms.awt.FontMetricsX.getFace could not be converted

com.ms.awt.FontMetricsX.getFontMetrics could not be converted

com.ms.awt.FontMetricsX.getLeading could not be converted

com.ms.awt.FontMetricsX.getMaxAdvance could not be converted

com.ms.awt.FontMetricsX.getWidths could not be converted

com.ms.awt.FontMetricsX.stringWidth could not be converted

com.ms.awt.FontX.chooseFont could not be converted

com.ms.awt.FontX.EMBEDDED could not be converted

com.ms.awt.FontX.getAttributeList could not be converted

com.ms.awt.FontX.getFlags could not be converted

com.ms.awt.FontX.getFlagsVal could not be converted

com.ms.awt.FontX.getFont could not be converted

com.ms.awt.FontX.getFontList could not be converted

com.ms.awt.FontX.getFontNativeData could not be converted

com.ms.awt.FontX.getNativeData could not be converted

com.ms.awt.FontX.getStyleVal could not be converted

com.ms.awt.FontX.isTypeable could not be converted

com.ms.awt.FontX.matchFace could not be converted

com.ms.awt.FontX.OUTLINE could not be converted

com.ms.awt.FontX.USEDFONT could not be converted

com.ms.awt.GenericEvent could not be converted

com.ms.awt.GraphicsX.bitBlt could not be converted

com.ms.awt.GraphicsX.cng could not be converted

com.ms.awt.GraphicsX.comp could not be converted

com.ms.awt.GraphicsX.copyArea could not be converted

com.ms.awt.GraphicsX.drawOutlineChar could not be converted

com.ms.awt.GraphicsX.drawOutlinePolygon could not be converted

com.ms.awt.GraphicsX.drawPixels could not be converted

com.ms.awt.GraphicsX.drawScanLines could not be converted

com.ms.awt.GraphicsX.drawT2Curve could not be converted

com.ms.awt.GraphicsX.gdc could not be converted

com.ms.awt.GraphicsX.getColorType could not be converted

com.ms.awt.GraphicsX.getGlyphOutline could not be converted

com.ms.awt.GraphicsX.go could not be converted

com.ms.awt.GraphicsX.GraphicsX could not be converted

com.ms.awt.GraphicsX.image could not be converted

com.ms.awt.GraphicsX.originX could not be converted

com.ms.awt.GraphicsX.originY could not be converted

com.ms.awt.GraphicsX.resetSurfaceParams could not be converted

com.ms.awt.GraphicsX.setClip could not be converted

com.ms.awt.GraphicsX.setPaintMode could not be converted

com.ms.awt.GraphicsX.setSurfaceOffset could not be converted

com.ms.awt.GraphicsX.setSurfaceOwner could not be converted

com.ms.awt.GraphicsX.setSurfaceVisRgn could not be converted

com.ms.awt.GraphicsX.setXORMode could not be converted

com.ms.awt.GraphicsXConstants.BDR_VALID could not be converted

com.ms.awt.HeavyComponent could not be converted

com.ms.awt.HorizBagLayout could not be converted

com.ms.awt.ImageX could not be converted

com.ms.awt.ListLayout could not be converted

com.ms.awt.MenuBarX.getItemID could not be converted

com.ms.awt.MenuBarX.MenuBarX(int) could not be converted

com.ms.awt.MenuBarX.MenuBarX(int, Applet, string) could not be converted

com.ms.awt.MenuBarX.MenuBarX(int, string) could not be converted

com.ms.awt.MenuItemX.addNotify could not be converted

com.ms.awt.MenuItemX.getID could not be converted

com.ms.awt.MenuX.CheckMenuItem could not be converted

com.ms.awt.MenuX.getItemID could not be converted

com.ms.awt.MenuXConstants could not be converted

com.ms.awt.OrientableFlowLayout could not be converted

com.ms.awt.PhysicalDrawingSurface could not be converted

com.ms.awt.PlatformFont could not be converted

com.ms.awt.VariableGridLayout could not be converted

com.ms.awt.VerticalBagLayout could not be converted

com.ms.awt.WClipboard.lostClipboard could not be converted

com.ms.awt.WClipboard.lostSelectionOwnership could not be converted

com.ms.awt.WClipboard.setToolkit could not be converted

com.ms.awt.WClipboard.WClipboard could not be converted

com.ms.awt.WComponentPeer could not be converted

com.ms.awt.WDragSession could not be converted

com.ms.awt.WEventQueue could not be converted

com.ms.awt.WFileDialogPeer could not be converted

com.ms.awt.WGuiCallback could not be converted

com.ms.awt.WHeavyPeer could not be converted

com.ms.awt.Win32SystemResourceDecoder could not be converted

com.ms.awt.WinEvent.notify could not be converted

com.ms.awt.WPrintGraphics could not be converted

com.ms.awt.WPrintJob could not be converted

com.ms.awt.WToolkit could not be converted

Com.ms.com Error Messages

com.ms.com.AnsiStringMarshaller could not be converted

com.ms.com.AnsiStringRef could not be converted

com.ms.com.COAUTHIDENTITY.cbByValSize could not be converted

com.ms.com.COAUTHIDENTITY.domain could not be converted

com.ms.com.COAUTHIDENTITY.flags could not be converted

com.ms.com.COAUTHIDENTITY.fromPtr could not be converted

com.ms.com.COAUTHIDENTITY.password could not be converted

com.ms.com.COAUTHIDENTITY.toExternal could not be converted

com.ms.com.COAUTHIDENTITY.toJava could not be converted

com.ms.com.COAUTHIDENTITY.toPtr could not be converted

com.ms.com.COAUTHIDENTITY.user could not be converted

com.ms.com.COAUTHINFO.pAuthIdentityData could not be converted

com.ms.com.COAUTHINFO.pwszServerPrincName could not be converted

com.ms.com.ComContext could not be converted

com.ms.com.ComException.ComException could not be converted

com.ms.com.ComException.getHelpContext could not be converted

com.ms.com.ComException.getHResult could not be converted

com.ms.com.ComException.hr could not be converted

com.ms.com.ComException.m_helpContext could not be converted

com.ms.com.ComException.m_helpFile could not be converted

com.ms.com.ComException.m_source could not be converted

com.ms.com.ComFailException.ComFailException could not be converted

com.ms.com.ComLib.declareMessagePumpThread could not be converted

com.ms.com.ComLib.IID_IDispatch could not be converted

com.ms.com.ComLib.IID_IUnknown could not be converted

com.ms.com.ComLib.ComLib could not be converted

com.ms.com.ComLib.executeOnContext could not be converted

com.ms.com.ComLib.freeUnusedLibraries could not be converted

com.ms.com.ComLib.IENVNextMarshalerC2J could not be converted

com.ms.com.ComLib.IENVNextMarshalerJ2C could not be converted

com.ms.com.ComLib.isEqualUnknown could not be converted

com.ms.com.ComLib.jcdwClassOffsetOf could not be converted

com.ms.com.ComLib.jcdwOffsetOf could not be converted

com.ms.com.ComSuccessException.ComSuccessException could not be converted

com.ms.com.ComLib.makeProxyRef could not be converted

com.ms.com.ComLib.ownsCleanup could not be converted

com.ms.com.ComLib.ptrToUnknown could not be converted

com.ms.com.ComLib.setDataWrapperSize could not be converted

com.ms.com.ComLib.startMTAThread could not be converted

com.ms.com.ComLib.supportsInterface could not be converted

com.ms.com.ComLib.threadStartMTA could not be converted

com.ms.com.ComLib.unknownToPtr could not be converted

com.ms.com.CONNECTDATA.pUnk could not be converted

com.ms.com.COSERVERINFO.pAuthInfo could not be converted

com.ms.com.COSERVERINFO.pwszName could not be converted

com.ms.com.CUnknown could not be converted

com.ms.com.CustomLib could not be converted

com.ms.com.Dispatch could not be converted

com.ms.com.DispatchProxy could not be converted

com.ms.com.Generic could not be converted

com.ms.com.Guid.Guid could not be converted

com.ms.com.Guid.setByStr could not be converted

com.ms.com.IAccessible.iid could not be converted

com.ms.com.IAccessibleDefault.iid could not be converted

com.ms.com.IBindCtx.iid could not be converted

com.ms.com.IBindCtx.RegisterObjectBound could not be converted

com.ms.com.IBindCtx.RevokeObjectBound could not be converted

com.ms.com.IBindCtx.RevokeObjectParam could not be converted

com.ms.com.IClassFactory could not be converted

com.ms.com.IClassFactory2 could not be converted

com.ms.com.IConnectionPoint.Advise could not be converted

com.ms.com.IConnectionPoint.iid could not be converted

com.ms.com.IConnectionPointContainer.iid could not be converted

com.ms.com.IEnumConnectionPoints.iid could not be converted

com.ms.com.IEnumConnectionPoints.Next could not be converted

com.ms.com.IEnumConnections.iid could not be converted

com.ms.com.IEnumConnections.Next could not be converted

com.ms.com.IEnumConnections.Skip could not be converted

com.ms.com.IEnumMoniker.iid could not be converted

com.ms.com.IEnumMoniker.Next could not be converted

com.ms.com.IEnumSTATSTG could not be converted

com.ms.com.IEnumString.iid could not be converted

com.ms.com.IEnumString.Next could not be converted

com.ms.com.IEnumUnknown could not be converted

com.ms.com.IEnumVariant.Clone could not be converted

com.ms.com.IExternalConnectionSink could not be converted

com.ms.com.IIDIsMarshaler could not be converted

com.ms.com.ILicenseMgr could not be converted

com.ms.com.ILockBytes could not be converted

com.ms.com.IMarshal.GetMarshalSizeMax could not be converted

com.ms.com.IMarshal.GetUnmarshalClass could not be converted

com.ms.com.IMarshal.iid could not be converted

com.ms.com.IMoniker.BindToObject could not be converted

com.ms.com.IMoniker.iid could not be converted

com.ms.com.IMoniker.IsDirty could not be converted

com.ms.com.IMoniker.IsEqual could not be converted

com.ms.com.IMoniker.IsRunning could not be converted

com.ms.com.IParseDisplayName could not be converted

com.ms.com.IPersist.iid could not be converted

com.ms.com.IPersistFile.iid could not be converted

com.ms.com.IPersistFile.IsDirty could not be converted

com.ms.com.IPersistFile.Save could not be converted

com.ms.com.IPersistStorage could not be converted

com.ms.com.IPersistStream.GetClassID could not be converted

com.ms.com.IPersistStream.iid could not be converted

com.ms.com.IPersistStreamInit could not be converted

com.ms.com.IPropertyNotifySink could not be converted

com.ms.com.IROTData could not be converted

com.ms.com.IRunningObjectTable.GetObject could not be converted

com.ms.com.IRunningObjectTable.GetTimeOfLastChange could not be converted

com.ms.com.IRunningObjectTable.iid could not be converted

com.ms.com.IRunningObjectTable.IsRunning could not be converted

com.ms.com.IRunningObjectTable.NoteChangeTime could not be converted

com.ms.com.IRunningObjectTable.Register could not be converted

com.ms.com.ISequentialStream could not be converted

com.ms.com.ISequentialStream.iid could not be converted

com.ms.com.IServiceProvider could not be converted

com.ms.com.IStorage could not be converted

com.ms.com.IStream.CopyTo could not be converted

com.ms.com.IStream.iid could not be converted

com.ms.com.IStream.LOCK_EXCLUSIVE could not be converted

com.ms.com.IStream.LOCK_ONLYONCE could not be converted

com.ms.com.IStream.LOCK_WRITE could not be converted

com.ms.com.IStream.Read could not be converted

com.ms.com.IStream.STATFLAG_DEFAULT could not be converted

com.ms.com.IStream.STATFLAG_NONAME could not be converted

com.ms.com.IStream.STATFLAG_NOOPEN could not be converted

com.ms.com.IStream.STGC_DANGEROUSLYCOMMITMERELYTODISKCACHE could not be converted

com.ms.com.IStream.STGC_DEFAULT could not be converted

com.ms.com.IStream.STGC_ONLYIFCURRENT could not be converted

com.ms.com.IStream.STGC_OVERWRITE could not be converted

com.ms.com.IStream.STREAM_SEEK_CUR could not be converted

com.ms.com.IStream.STREAM_SEEK_END could not be converted

com.ms.com.IStream.STREAM_SEEK_SET could not be converted

com.ms.com.IStream.Write could not be converted

com.ms.com.LicenseMgr could not be converted

com.ms.com.LICINFO could not be converted

com.ms.com.MULTI_QI could not be converted

com.ms.com.NoAutoMarshaling could not be converted

com.ms.com.NoAutoScripting could not be converted

com.ms.com.SafeArray.destroy could not be converted

com.ms.com.SafeArray.getFeatures could not be converted

com.ms.com.SafeArray.getNumLocks could not be converted

com.ms.com.SafeArray.getPhysicalSafeArray could not be converted

com.ms.com.SafeArray.getvt could not be converted

com.ms.com.SafeArray.reinit could not be converted

com.ms.com.SafeArray.reinterpretType could not be converted

com.ms.com.SizeIsMarshaler could not be converted

com.ms.com.STATSTG.clsid_data1 could not be converted

com.ms.com.STATSTG.clsid_data2 could not be converted

com.ms.com.STATSTG.clsid_data3 could not be converted

com.ms.com.STATSTG.STGTY_LOCKBYTES could not be converted

com.ms.com.STATSTG.STGTY_PROPERTY could not be converted

com.ms.com.STATSTG.STGTY_STORAGE could not be converted

com.ms.com.STATSTG.STGTY_STREAM could not be converted

com.ms.com.StdCOMClassObject could not be converted

com.ms.com.UniStringMarshaller could not be converted

com.ms.com.UniStringRef could not be converted

com.ms.com.Variant.changeType could not be converted

com.ms.com.Variant.VariantClear could not be converted

com.ms.com.Variant.clone could not be converted

com.ms.com.Variant.cloneIndirect could not be converted

com.ms.com.Variant.getEmpty could not be converted

com.ms.com.Variant.getErrorRef could not be converted

com.ms.com.Variant.getNull could not be converted

com.ms.com.Variant.getVariantArray could not be converted

com.ms.com.Variant.getVariantArrayRef could not be converted

com.ms.com.Variant.noParam could not be converted

com.ms.com.Variant.putError could not be converted

com.ms.com.Variant.putSafeArrayRefHelper could not be converted

com.ms.com.Variant.toError could not be converted

com.ms.com.Variant.toScriptObject could not be converted

com.ms.com.Variant.toVariantArray could not be converted

com.ms.com.Variant.Variant could not be converted

com.ms.com.Variant.VariantByref could not be converted

com.ms.com.Variant.VariantCurrency could not be converted

com.ms.com.Variant.VariantEmpty could not be converted

com.ms.com.Variant.VariantError could not be converted

com.ms.com.Variant.VariantNull could not be converted

com.ms.com.Variant.VariantTypeMask could not be converted

com.ms.com.Variant.VariantVariant could not be converted

Com.ms.directX Error Messages

com.ms.directX.D3dFindDeviceResult.GetGuid could not be converted

com.ms.directX.D3dFindDeviceSearch.getGuid could not be converted

com.ms.directX.D3dFindDeviceSearch.setGuid could not be converted

com.ms.directX.Direct3dRMFace.getVertices could not be converted

com.ms.directX.Direct3dRMMesh.getVertices could not be converted

com.ms.directX.Direct3dRMMesh.setVertices could not be converted

com.ms.directX.Direct3dRMMeshBuilder.addFaces

com.ms.directX.Direct3dRMMeshBuilder.getVertices

com.ms.directX.DirectDraw.createPalette could not be converted

com.ms.directX.DirectDraw.setCooperativeLevel could not be converted

com.ms.directX.DirectDrawClipper.resetSurfaceParams could not be converted

com.ms.directX.DirectDrawClipper.setComponent could not be converted

com.ms.directX.DirectDrawClipper.setSurfaceOffset could not be converted

com.ms.directX.DirectDrawClipper.setSurfaceOwner could not be converted

com.ms.directX.DirectDrawClipper.setSurfaceVisRgn could not be converted

com.ms.directX.DirectDrawPalette.getColorEntries could not be converted

com.ms.directX.DirectDrawPalette.getPaletteEntries could not be converted

com.ms.directX.DirectDrawPalette.setEntries could not be converted

com.ms.directX.DirectSound.createSoundBuffer could not be converted

com.ms.directX.DirectSound.setCooperativeLevel could not be converted

com.ms.directX.DirectSoundBuffer.getFormat could not be converted

com.ms.directX.DirectSoundBuffer.initialize could not be converted

com.ms.directX.DirectSoundBuffer.setFormat could not be converted

com.ms.directX.DirectSoundResource.loadWaveFile could not be converted

com.ms.directX.DirectSoundResource.loadWaveResource could not be converted

com.ms.directX.DirectXConstants could not be converted

com.ms.directX.PaletteEntry could not be converted

com.ms.directX.WaveFormatEx could not be converted

Com.ms.dll Error Messages

com.ms.dll.Callback could not be converted

com.ms.dll.DllLib.addrOf could not be converted

com.ms.dll.DllLib.addrOfPinnedObject could not be converted

com.ms.dll.DllLib.copy could not be converted

com.ms.dll.DllLib.DllLib could not be converted

com.ms.dll.DllLib.freePinnedHandle could not be converted

com.ms.dll.DllLib.getPinnedHandle could not be converted

com.ms.dll.DllLib.getPinnedObject could not be converted

com.ms.dll.DllLib.isStruct could not be converted

com.ms.dll.DllLib.propagateStructFields could not be converted

com.ms.dll.DllLib.resize could not be converted

com.ms.dll.ParameterCountMismatchError could not be converted

com.ms.dll.StringMarshaler could not be converted

Com.ms.dxmedia Error Messages

com.ms.dxmedia.AppTriggeredEvent could not be converted

com.ms.dxmedia.ArrayBvr.ArrayBvr could not be converted

com.ms.dxmedia.ArrayBvr.getCOMPtr could not be converted

com.ms.dxmedia.ArrayBvr.newUninitBvr could not be converted

com.ms.dxmedia.ArrayBvr.setCOMBvr could not be converted

com.ms.dxmedia.Bbox2Bvr.Bbox2Bvr could not be converted

com.ms.dxmedia.Bbox2Bvr.getCOMPtr could not be converted

com.ms.dxmedia.Bbox2Bvr.newUninitBehavior could not be converted

com.ms.dxmedia.Bbox2Bvr.newUninitBvr could not be converted

com.ms.dxmedia.Bbox2Bvr.setCOMBvr could not be converted

com.ms.dxmedia.Bbox3Bvr.Bbox3Bvr could not be converted

com.ms.dxmedia.Bbox3Bvr.getCOMPtr could not be converted

com.ms.dxmedia.Bbox3Bvr.newUninitBehavior could not be converted

com.ms.dxmedia.Bbox3Bvr.newUninitBvr could not be converted

com.ms.dxmedia.Bbox3Bvr.setCOMBvr could not be converted

com.ms.dxmedia.Behavior.Behavior could not be converted

com.ms.dxmedia.Behavior.debug could not be converted

com.ms.dxmedia.Behavior.extract could not be converted

com.ms.dxmedia.Behavior.getCOMBvr could not be converted

com.ms.dxmedia.Behavior.newUninitBehavior could not be converted

com.ms.dxmedia.BooleanBvr.BooleanBvr could not be converted

com.ms.dxmedia.BooleanBvr.getCOMPtr could not be converted

com.ms.dxmedia.BooleanBvr.newUninitBehavior could not be converted

com.ms.dxmedia.BooleanBvr.setCOMBvr could not be converted

com.ms.dxmedia.CallbackNotifier could not be converted

com.ms.dxmedia.CameraBvr.CameraBvr could not be converted

com.ms.dxmedia.CameraBvr.getCOMPtr could not be converted

com.ms.dxmedia.CameraBvr.newUninitBehavior could not be converted

com.ms.dxmedia.CameraBvr.newUninitBvr could not be converted

com.ms.dxmedia.CameraBvr.setCOMBvr could not be converted

com.ms.dxmedia.ColorBvr.ColorBvr could not be converted

com.ms.dxmedia.ColorBvr.getCOMPtr could not be converted

com.ms.dxmedia.ColorBvr.newUninitBehavior could not be converted

com.ms.dxmedia.ColorBvr.NewUninitBvr could not be converted

com.ms.dxmedia.ColorBvr.setCOMBvr could not be converted

com.ms.dxmedia.Cycler could not be converted

com.ms.dxmedia.DashStyleBvr.DashStyleBvr could not be converted

com.ms.dxmedia.DashStyleBvr.getCOMPtr could not be converted

com.ms.dxmedia.DashStyleBvr.newUninitBehavior could not be converted

com.ms.dxmedia.DashStyleBvr.newUninitBvr could not be converted

com.ms.dxmedia.DashStyleBvr.setCOMBvr could not be converted

com.ms.dxmedia.DefaultErrReceiver could not be converted

com.ms.dxmedia.DXMApplet could not be converted

com.ms.dxmedia.DXMCanvas could not be converted

com.ms.dxmedia.DXMCanvasBase could not be converted

com.ms.dxmedia.DXMDebugCallback could not be converted

com.ms.dxmedia.DXMEvent.DXMEvent could not be converted

com.ms.dxmedia.DXMEvent.getCOMPtr could not be converted

com.ms.dxmedia.DXMEvent.newUninitBehavior could not be converted

com.ms.dxmedia.DXMEvent.registerCallback could not be converted

com.ms.dxmedia.DXMEvent.setCOMBvr could not be converted

com.ms.dxmedia.DXMException.DXMException could not be converted

com.ms.dxmedia.EndStyleBvr.EndStyleBvr could not be converted

com.ms.dxmedia.EndStyleBvr.getCOMPtr could not be converted

com.ms.dxmedia.EndStyleBvr.setCOMBvr could not be converted

com.ms.dxmedia.EndStyleBvr.newUninitBehavior could not be converted

com.ms.dxmedia.EndStyleBvr.newUninitBvr could not be converted

com.ms.dxmedia.ErrorAndWarningReceiver could not be converted

com.ms.dxmedia.EventCallbackObject could not be converted

com.ms.dxmedia.FontStyleBvr.FontStyleBvr could not be converted

com.ms.dxmedia.FontStyleBvr.getCOMPtr could not be converted

com.ms.dxmedia.FontStyleBvr.newUninitBehavior could not be converted

com.ms.dxmedia.FontStyleBvr.NewUninitBvr could not be converted

com.ms.dxmedia.FontStyleBvr.setCOMBvr could not be converted

com.ms.dxmedia.GeometryBvr.GeometryBvr could not be converted

com.ms.dxmedia.GeometryBvr.getCOMPtr could not be converted

com.ms.dxmedia.GeometryBvr.newUninitBehavior could not be converted

com.ms.dxmedia.GeometryBvr.setCOMBvr could not be converted

com.ms.dxmedia.ImageBvr.getCOMPtr could not be converted

com.ms.dxmedia.ImageBvr.ImageBvr could not be converted

com.ms.dxmedia.ImageBvr.newUninitBehavior could not be converted

com.ms.dxmedia.ImageBvr.setCOMBvr could not be converted

com.ms.dxmedia.JoinStyleBvr.getCOMPtr could not be converted

com.ms.dxmedia.JoinStyleBvr.JoinStyleBvr could not be converted

com.ms.dxmedia.JoinStyleBvr.newUninitBehavior could not be converted

com.ms.dxmedia.JoinStyleBvr.setCOMBvr could not be converted

com.ms.dxmedia.LineStyleBvr.getCOMPtr could not be converted

com.ms.dxmedia.LineStyleBvr.LineStyleBvr could not be converted

com.ms.dxmedia.LineStyleBvr.newUninitBehavior could not be converted

com.ms.dxmedia.LineStyleBvr.setCOMBvr could not be converted

com.ms.dxmedia.MatteBvr.getCOMPtr could not be converted

com.ms.dxmedia.MatteBvr.MatteBvr could not be converted

com.ms.dxmedia.MatteBvr.newUninitBehavior could not be converted

com.ms.dxmedia.MatteBvr.setCOMBvr could not be converted

com.ms.dxmedia.MicrophoneBvr.getCOMPtr could not be converted

com.ms.dxmedia.MicrophoneBvr.MicrophoneBvr could not be converted

com.ms.dxmedia.MicrophoneBvr.newUninitBehavior could not be converted

com.ms.dxmedia.MicrophoneBvr.setCOMBvr could not be converted

com.ms.dxmedia.Model.cleanup could not be converted

com.ms.dxmedia.Model.createModel could not be converted

com.ms.dxmedia.Model.getImportBase could not be converted

com.ms.dxmedia.Model.modifyPreferences could not be converted

com.ms.dxmedia.Model.receiveInputImages could not be converted

com.ms.dxmedia.Model.setImportBase could not be converted

com.ms.dxmedia.ModelMakerApplet could not be converted

com.ms.dxmedia.ModifiableBehavior could not be converted

com.ms.dxmedia.MontageBvr.getCOMPtr could not be converted

com.ms.dxmedia.MontageBvr.MontageBvr could not be converted

com.ms.dxmedia.MontageBvr.newUninitBehavior could not be converted

com.ms.dxmedia.MontageBvr.setCOMBvr could not be converted

com.ms.dxmedia.NumberBvr.getCOMPtr could not be converted

com.ms.dxmedia.NumberBvr.newUninitBehavior could not be converted

com.ms.dxmedia.NumberBvr.NumberBvr could not be converted

com.ms.dxmedia.NumberBvr.setCOMBvr could not be converted

com.ms.dxmedia.PairObject.PairObject could not be converted

com.ms.dxmedia.Path2Bvr.getCOMPtr could not be converted

com.ms.dxmedia.Path2Bvr.newUninitBehavior could not be converted

com.ms.dxmedia.Path2Bvr.Path2Bvr could not be converted

com.ms.dxmedia.Path2Bvr.setCOMBvr could not be converted

com.ms.dxmedia.PickableGeometry.PickableGeometry could not be converted

com.ms.dxmedia.PickableImage.PickableImage could not be converted

com.ms.dxmedia.Point2Bvr.getCOMPtr could not be converted

com.ms.dxmedia.Point2Bvr.newUninitBehavior could not be converted

com.ms.dxmedia.Point2Bvr.Point2Bvr could not be converted

com.ms.dxmedia.Point2Bvr.setCOMBvr could not be converted

com.ms.dxmedia.Point3Bvr.getCOMPtr could not be converted

com.ms.dxmedia.Point3Bvr.newUninitBehavior could not be converted

com.ms.dxmedia.Point3Bvr.Point3Bvr could not be converted

com.ms.dxmedia.Point3Bvr.setCOMBvr could not be converted

com.ms.dxmedia.Preferences.COLOR_KEY_BLUE could not be converted

com.ms.dxmedia.Preferences.COLOR_KEY_GREEN could not be converted

com.ms.dxmedia.Preferences.COLOR_KEY_RED could not be converted

com.ms.dxmedia.Preferences.DITHERING could not be converted

com.ms.dxmedia.Preferences.ENGINE_OPTIMIZATIONS could not be converted

com.ms.dxmedia.Preferences.FILL_MODE could not be converted

com.ms.dxmedia.Preferences.FILL_MODE_POINT could not be converted

com.ms.dxmedia.Preferences.FILL_MODE_SOLID could not be converted

com.ms.dxmedia.Preferences.FILL_MODE_WIREFRAME could not be converted

com.ms.dxmedia.Preferences.MAX_FRAMES_PER_SEC could not be converted

com.ms.dxmedia.Preferences.OVERRIDE_APPLICATION_PREFERENCES could not be converted

com.ms.dxmedia.Preferences.PERSPECTIVE_CORRECT could not be converted

com.ms.dxmedia.Preferences.RGB_LIGHTING_MODE could not be converted

com.ms.dxmedia.Preferences.SHADE_MODE could not be converted

com.ms.dxmedia.Preferences.SHADE_MODE_FLAT could not be converted

com.ms.dxmedia.Preferences.SHADE_MODE_GOURAUD could not be converted

com.ms.dxmedia.Preferences.SHADE_MODE_PHONG could not be converted

com.ms.dxmedia.Preferences.TEXTURE_QUALITY could not be converted

com.ms.dxmedia.Preferences.TEXTURE_QUALITY_LINEAR could not be converted

com.ms.dxmedia.Preferences.TEXTURE_QUALITY_NEAREST could not be converted

com.ms.dxmedia.Preferences.USE_3D_HW could not be converted

com.ms.dxmedia.Preferences.USE_VIDEOMEM could not be converted

com.ms.dxmedia.PropertyDispatcher could not be converted

com.ms.dxmedia.SoundBvr.getCOMPtr could not be converted

com.ms.dxmedia.SoundBvr.newUninitBehavior could not be converted

com.ms.dxmedia.SoundBvr.setCOMBvr could not be converted

com.ms.dxmedia.SoundBvr.SoundBvr could not be converted

com.ms.dxmedia.Statics.makeBvrFromInterface could not be converted

com.ms.dxmedia.StaticsBase._site could not be converted

com.ms.dxmedia.StaticsBase.BvrHook could not be converted

com.ms.dxmedia.StaticsBase.checkRead could not be converted

com.ms.dxmedia.StaticsBase.cm could not be converted

com.ms.dxmedia.StaticsBase.foot could not be converted

com.ms.dxmedia.StaticsBase.getCOMPtr could not be converted

com.ms.dxmedia.StaticsBase.handleError could not be converted

com.ms.dxmedia.StaticsBase.importGeometry could not be converted

com.ms.dxmedia.StaticsBase.importImage could not be converted

com.ms.dxmedia.StaticsBase.importMovie (asynchronous) could not be converted

com.ms.dxmedia.StaticsBase.importMovie (synchronous) could not be converted

com.ms.dxmedia.StaticsBase.importSound (asynchronous) could not be converted

com.ms.dxmedia.StaticsBase.importSound (synchronous) could not be converted

com.ms.dxmedia.StaticsBase.inch could not be converted

com.ms.dxmedia.StaticsBase.meter could not be converted

com.ms.dxmedia.StaticsBase.mm could not be converted

com.ms.dxmedia.StaticsBase.registerErrorAndWarningReceiver could not be converted

com.ms.dxmedia.StaticsBase.unregisterCallback could not be converted

com.ms.dxmedia.StringBvr.getCOMPtr could not be converted

com.ms.dxmedia.StringBvr.newUninitBehavior could not be converted

com.ms.dxmedia.StringBvr.setCOMBvr could not be converted

com.ms.dxmedia.StringBvr.StringBvr could not be converted

com.ms.dxmedia.Transform2Bvr.getCOMPtr could not be converted

com.ms.dxmedia.Transform2Bvr.newUninitBehavior could not be converted

com.ms.dxmedia.Transform2Bvr.setCOMBvr could not be converted

com.ms.dxmedia.Transform2Bvr.Transform2Bvr could not be converted

com.ms.dxmedia.Transform3Bvr.getCOMPtr could not be converted

com.ms.dxmedia.Transform3Bvr.newUninitBehavior could not be converted

com.ms.dxmedia.Transform3Bvr.setCOMBvr could not be converted

com.ms.dxmedia.Transform3Bvr.Transform3Bvr could not be converted

com.ms.dxmedia.TupleBvr.getCOMPtr could not be converted

com.ms.dxmedia.TupleBvr.setCOMBvr could not be converted

com.ms.dxmedia.TupleBvr.TupleBvr could not be converted

com.ms.dxmedia.UntilNotifierCB could not be converted

com.ms.dxmedia.Vector2Bvr.getCOMPtr could not be converted

com.ms.dxmedia.Vector2Bvr.newUninitBehavior could not be converted

com.ms.dxmedia.Vector2Bvr.setCOMBvr could not be converted

com.ms.dxmedia.Vector2Bvr.Vector2Bvr could not be converted

com.ms.dxmedia.Vector3Bvr.getCOMPtr could not be converted

com.ms.dxmedia.Vector3Bvr.newUninitBehavior could not be converted

com.ms.dxmedia.Vector3Bvr.setCOMBvr could not be converted

com.ms.dxmedia.Vector3Bvr.Vector3Bvr could not be converted

com.ms.dxmedia.Viewer.getCurrentTickTime could not be converted

com.ms.dxmedia.Viewer.registerErrorAndWarningReceiver could not be converted

com.ms.dxmedia.Viewer.startModel could not be converted

com.ms.dxmedia.Viewer.tick could not be converted

Com.ms.fx Error Messages

com.ms.fx.fullTxtRun could not be converted

com.ms.fx.FxBrushPen.drawCharsCallback could not be converted

com.ms.fx.FxBrushPen.drawRoundRectCallback could not be converted

com.ms.fx.FxBrushPen.drawScanLinesCallback could not be converted

com.ms.fx.FxBrushPen.drawStringCallback could not be converted

com.ms.fx.FxBrushPen.fill3DRectCallback could not be converted

com.ms.fx.FxBrushPen.fillRoundRectCallback could not be converted

com.ms.fx.FxCaret could not be converted

com.ms.fx.FxColor.brightenColor could not be converted

com.ms.fx.FxColor.darkenColor could not be converted

com.ms.fx.FxColor.getHilight could not be converted

com.ms.fx.FxColor.getShadow could not be converted

com.ms.fx.FxComponentImage could not be converted

com.ms.fx.FxComponentTexture.FxComponentTexture could not be converted

com.ms.fx.FxCurve could not be converted

com.ms.fx.FxEllipse could not be converted

com.ms.fx.FxFont.ANTIALIAS could not be converted

com.ms.fx.FxFont.drawEffects could not be converted

com.ms.fx.FxFont.FONTXFONT could not be converted

com.ms.fx.FxFont.FxFont could not be converted

com.ms.fx.FxFont.getAttributeList could not be converted

com.ms.fx.FxFont.getEmboldenedFont could not be converted

com.ms.fx.FxFont.getFlags could not be converted

com.ms.fx.FxFont.getFlagsVal could not be converted

com.ms.fx.FxFont.getFont could not be converted

com.ms.fx.FxFont.getFontList could not be converted

com.ms.fx.FxFont.getStyleVal could not be converted

com.ms.fx.FxFont.matchFace could not be converted

com.ms.fx.FxFont.STRIKEOUT could not be converted

com.ms.fx.FxFont.UNDERLINE could not be converted

com.ms.fx.FxFont.USEDFONT could not be converted

com.ms.fx.FxFontMetrics could not be converted

com.ms.fx.FxFontMetricsOther could not be converted

com.ms.fx.FxFormattedText could not be converted

com.ms.fx.FxGraphics.drawChars could not be converted

com.ms.fx.FxGraphics.drawCharsWithoutFxFont could not be converted

com.ms.fx.FxGraphics.drawOutlineChar could not be converted

com.ms.fx.FxGraphics.drawOutlinePolygon could not be converted

com.ms.fx.FxGraphics.drawPixels could not be converted

com.ms.fx.FxGraphics.drawScanLines could not be converted

com.ms.fx.FxGraphics.drawString could not be converted

com.ms.fx.FxGraphics.drawStringFormatted could not be converted

com.ms.fx.FxGraphics.drawStringWithoutFxFont could not be converted

com.ms.fx.FxGraphics.drawT2Curve could not be converted

com.ms.fx.FxGraphics.getClip could not be converted

com.ms.fx.FxGraphics.getExtendedGraphics could not be converted

com.ms.fx.FxGraphics.getGlyphOutline could not be converted

com.ms.fx.FxGraphics.intelliFont could not be converted

com.ms.fx.FxGraphics.setClip could not be converted

com.ms.fx.FxGraphicMetaFile.addObjectToTable could not be converted

com.ms.fx.FxGraphicMetaFile.debugging could not be converted

com.ms.fx.FxGraphicMetaFile.enumerate could not be converted

com.ms.fx.FxGraphicMetaFile.FxGraphicMetaFile could not be converted

com.ms.fx.FxGraphicMetaFile.hdc could not be converted

com.ms.fx.FxGraphicMetaFile.hdcStates could not be converted

com.ms.fx.FxGraphicMetaFile.play could not be converted

com.ms.fx.FxGraphicMetaFile.records could not be converted

com.ms.fx.FxGraphicMetaFile.removeObjectFromTable could not be converted

com.ms.fx.FxGraphicsOVM.baseGraphics could not be converted

com.ms.fx.FxGraphicsOVM.clearRect could not be converted

com.ms.fx.FxGraphicsOVM.clipRect could not be converted

com.ms.fx.FxGraphicsOVM.copyArea could not be converted

com.ms.fx.FxGraphicsOVM.create could not be converted

com.ms.fx.FxGraphicsOVM.dispose could not be converted

com.ms.fx.FxGraphicsOVM.drawArc could not be converted

com.ms.fx.FxGraphicsOVM.drawBezier could not be converted

com.ms.fx.FxGraphicsOVM.drawBorder could not be converted

com.ms.fx.FxGraphicsOVM.drawChars could not be converted

com.ms.fx.FxGraphicsOVM.drawCharsWithoutFxFont could not be converted

com.ms.fx.FxGraphicsOVM.drawOutlinePolygon could not be converted

com.ms.fx.FxGraphicsOVM.drawPixels could not be converted

com.ms.fx.FxGraphicsOVM.drawScanLines could not be converted

com.ms.fx.FxGraphicsOVM.FxGraphicsOVM could not be converted

com.ms.fx.FxGraphicsOVM.getGlyphOutline could not be converted

com.ms.fx.FxGraphicsOVM.hardClipRect could not be converted

com.ms.fx.FxGraphicsOVM.originX could not be converted

com.ms.fx.FxGraphicsOVM.originY could not be converted

com.ms.fx.FxGraphicsOVM.runningAFC could not be converted

com.ms.fx.FxGraphicsOVM.setClip could not be converted

com.ms.fx.FxGraphicsOVM.setClip(int, int, int) could not be converted

com.ms.fx.FxGraphicsOVM.setClip(Shape) could not be converted

com.ms.fx.FxGraphicsOVM.setPaintMode could not be converted

com.ms.fx.FxGraphicsOVM.setXORMode could not be converted

com.ms.fx.FxGraphicsOVM.systemInterface could not be converted

com.ms.fx.FxGraphicsOVM11 could not be converted

com.ms.fx.FxGraphicsOVM11.copyArea could not be converted

com.ms.fx.FxGraphicsOVM11.drawOutlinePolygon could not be converted

com.ms.fx.FxGraphicsOVM11.drawPixels could not be converted

com.ms.fx.FxGraphicsOVM11.drawScanLines could not be converted

com.ms.fx.FxGraphicsOVM11.getGlyphOutline could not be converted

com.ms.fx.FxGraphicsOVM11.originX could not be converted

com.ms.fx.FxGraphicsOVM11.originY could not be converted

com.ms.fx.FxGraphicsOVM11.runningAFC could not be converted

com.ms.fx.FxGraphicsOVM11.setClip could not be converted

com.ms.fx.FxGraphicsOVM11.setPaintMode could not be converted

com.ms.fx.FxGraphicsOVM11.setXORMode could not be converted

com.ms.fx.FxGraphicsOVM11.systemInterface could not be converted

com.ms.fx.FxOutlineFont could not be converted

com.ms.fx.FxPen.calcLineValues could not be converted

com.ms.fx.FxPen.drawRoundRectCallback could not be converted

com.ms.fx.FxPen.drawScanLinesCallback could not be converted

com.ms.fx.FxPen.fillRectCallback could not be converted

com.ms.fx.FxPen.FxPen could not be converted

com.ms.fx.FxPen.myDrawOval could not be converted

com.ms.fx.FxRubberPen could not be converted

com.ms.fx.FxStateConfigurableImage could not be converted

com.ms.fx.FxStateConfigurableUIImage.FxStateConfigurableUIImage could not be converted

com.ms.fx.FxStateConfigurableUIImage.getImageState could not be converted

com.ms.fx.FxStyledPen could not be converted

com.ms.fx.FxStyledPen.FxStyledPen could not be converted

com.ms.fx.FxSystemFont could not be converted

com.ms.fx.FxSystemIcon could not be converted

com.ms.fx.FxText.buffer could not be converted

com.ms.fx.FxText.getChar could not be converted

com.ms.fx.FxText.getWordBreak could not be converted

com.ms.fx.FxText.isDelimiter could not be converted

com.ms.fx.FxText.isWhite could not be converted

com.ms.fx.FxText.nChars could not be converted

com.ms.fx.FxText.setText could not be converted

com.ms.fx.FxTexture.DRAW_BL could not be converted

com.ms.fx.FxTexture.DRAW_BOTTOM could not be converted

com.ms.fx.FxTexture.DRAW_BR could not be converted

com.ms.fx.FxTexture.DRAW_CENTER could not be converted

com.ms.fx.FxTexture.DRAW_LEFT could not be converted

com.ms.fx.FxTexture.DRAW_RIGHT could not be converted

com.ms.fx.FxTexture.DRAW_TL could not be converted

com.ms.fx.FxTexture.DRAW_TOP could not be converted

com.ms.fx.FxTexture.DRAW_TR could not be converted

com.ms.fx.FxTexture.drawScanLinesCallback could not be converted

com.ms.fx.FxTexture.FxTexture could not be converted

com.ms.fx.FxTexture.getBottomAxis could not be converted

com.ms.fx.FxTexture.getInner could not be converted

com.ms.fx.FxTexture.getLeftAxis could not be converted

com.ms.fx.FxTexture.getPinOrigin could not be converted

com.ms.fx.FxTexture.getRightAxis could not be converted

com.ms.fx.FxTexture.getSnapDraw could not be converted

com.ms.fx.FxTexture.getStretch could not be converted

com.ms.fx.FxTexture.getTopAxis could not be converted

com.ms.fx.FxTexture.getUpdatedAreasMask could not be converted

com.ms.fx.FxTexture.imageUpdate could not be converted

com.ms.fx.FxTexture.REPEAT_PIN could not be converted

com.ms.fx.FxTexture.setAxis could not be converted

com.ms.fx.FxTexture.setPinOrigin could not be converted

com.ms.fx.FxTexture.setSnapDraw could not be converted

com.ms.fx.FxTexture.setStretch could not be converted

com.ms.fx.FxTexture.setUpdateCallback could not be converted

com.ms.fx.FxTexture.setUpdatedAreasMask could not be converted

com.ms.fx.FxTexture.size could not be converted

com.ms.fx.FxTexture.SNAP_EDGES could not be converted

com.ms.fx.FxTexture.STRETCH_MIDDLE could not be converted

com.ms.fx.FxTexture.STRETCH_OUTER could not be converted

com.ms.fx.FxToolkit could not be converted

com.ms.fx.GlyphMetrics could not be converted

com.ms.fx.GlyphOutline could not be converted

com.ms.fx.IFxShape could not be converted

com.ms.fx.IFxSystemInterface could not be converted

com.ms.fx.IFxTextCallback could not be converted

com.ms.fx.IFxTextConstants could not be converted

com.ms.fx.IFxTextConstants.DIRLAYOUT could not be converted

com.ms.fx.IFxTextConstants.htaCenter could not be converted

com.ms.fx.IFxTextConstants.htaJustified could not be converted

com.ms.fx.IFxTextConstants.htaLeft could not be converted

com.ms.fx.IFxTextConstants.htaRight could not be converted

com.ms.fx.IFxTextConstants.htaScriptDefault could not be converted

com.ms.fx.IFxTextConstants.htaStretch could not be converted

com.ms.fx.IFxTextConstants.MOVE_DOWN could not be converted

com.ms.fx.IFxTextConstants.MOVE_LEFT could not be converted

com.ms.fx.IFxTextConstants.MOVE_RIGHT could not be converted

com.ms.fx.IFxTextConstants.MOVE_UP could not be converted

com.ms.fx.IFxTextConstants.NEXT_DOWN could not be converted

com.ms.fx.IFxTextConstants.NEXT_LEFT could not be converted

com.ms.fx.IFxTextConstants.NEXT_RIGHT could not be converted

com.ms.fx.IFxTextConstants.NEXT_UP could not be converted

com.ms.fx.IFxTextConstants.OPAQUE_BODY could not be converted

com.ms.fx.IFxTextConstants.OPAQUE_POST could not be converted

com.ms.fx.IFxTextConstants.OPAQUE_PRIOR could not be converted

com.ms.fx.IFxTextConstants.SCRIPT_DEF could not be converted

com.ms.fx.IFxTextConstants.tdBT_LR could not be converted

com.ms.fx.IFxTextConstants.tdBT_RL could not be converted

com.ms.fx.IFxTextConstants.tdHebrewNormal could not be converted

com.ms.fx.IFxTextConstants.tdJapanTradNormal could not be converted

com.ms.fx.IFxTextConstants.tdLatinNormal could not be converted

com.ms.fx.IFxTextConstants.tdLeftToRightReading could not be converted

com.ms.fx.IFxTextConstants.tdLR_BT could not be converted

com.ms.fx.IFxTextConstants.tdLR_TB could not be converted

com.ms.fx.IFxTextConstants.tdMongolianNormal could not be converted

com.ms.fx.IFxTextConstants.tdRightToLeftReading could not be converted

com.ms.fx.IFxTextConstants.tdRL_BT could not be converted

com.ms.fx.IFxTextConstants.tdRL_TB could not be converted

com.ms.fx.IFxTextConstants.tdScriptDefault could not be converted

com.ms.fx.IFxTextConstants.tdTB_LR could not be converted

com.ms.fx.IFxTextConstants.tdTB_RL could not be converted

com.ms.fx.IFxTextConstants.tdVisualLayout could not be converted

com.ms.fx.IFxTextConstants.VISUAL_DEF could not be converted

com.ms.fx.IFxTextConstants.vtaBaseline could not be converted

com.ms.fx.IFxTextConstants.vtaBottom could not be converted

com.ms.fx.IFxTextConstants.vtaCenter could not be converted

com.ms.fx.IFxTextConstants.vtaScriptDefault could not be converted

com.ms.fx.IFxTextConstants.vtaStretch could not be converted

com.ms.fx.IFxTextConstants.vtaTop could not be converted

com.ms.fx.IFxTextConstants.wwCleanEdges could not be converted

com.ms.fx.IFxTextConstants.wwKeepWordIntact could not be converted

com.ms.fx.IFxTextConstants.wwMask could not be converted

com.ms.fx.IFxTextConstants.wwNone could not be converted

com.ms.fx.IFxTextConstants.wwTypeMask could not be converted

com.ms.fx.IFxTextConstants.wwVirtualRectEnd could not be converted

com.ms.fx.IFxTextConstants.wwVirtualRectSide could not be converted

com.ms.fx.IFxTextConstants.wwWrap could not be converted

com.ms.fx.IFxTextureUpdate could not be converted

com.ms.fx.OutlineCurve could not be converted

com.ms.fx.OutlinePolygon could not be converted

com.ms.fx.PeerConstants could not be converted

com.ms.fx.Region.clone could not be converted

com.ms.fx.Region.COMPLEX could not be converted

com.ms.fx.Region.complexity could not be converted

com.ms.fx.Region.copy could not be converted

com.ms.fx.Region.EMPTY~ could not be converted

com.ms.fx.Region.equals could not be converted

com.ms.fx.Region.fillBounds could not be converted

com.ms.fx.Region.getBounds could not be converted

com.ms.fx.Region.getGeometry could not be converted

com.ms.fx.Region.invert could not be converted

com.ms.fx.Region.isEmpty could not be converted

com.ms.fx.Region.Region could not be converted

com.ms.fx.Region.set could not be converted

com.ms.fx.Region.SIMPLE could not be converted

com.ms.fx.RegionConverter could not be converted

com.ms.fx.txtRun could not be converted

com.ms.fx.Version could not be converted

Com.ms.io Error Messages

com.ms.io.console.Console could not be converted

com.ms.io.console.DefaultConsole could not be converted

com.ms.io.ObjectInputStreamWithLoader could not be converted

com.ms.io.OffsetInputStreamFilter.mark could not be converted

com.ms.io.OffsetInputStreamFilter.reset could not be converted

com.ms.io.Path.hasExecExtensionType could not be converted

com.ms.io.Path.isRoot could not be converted

com.ms.io.Path.validateFilename could not be converted

com.ms.io.SystemInputStream could not be converted

com.ms.io.SystemOutputStream could not be converted

com.ms.io.UserFileDialog could not be converted

Com.ms.jdbc.odbc Error Messages

com.ms.jdbc.odbc.JdbcOdbcConnection.getAutoCommit could not be converted

com.ms.jdbc.odbc.JdbcOdbcConnection.getMetaData could not be converted

Com.ms.lang Error Messages

com.ms.lang.MulticastDelegate.invokeHelperMulticast could not be converted

com.ms.lang.RegKey.enumKey could not be converted

com.ms.lang.RegKey.finalize could not be converted

com.ms.lang.RegKey.getBinaryValue could not be converted

com.ms.lang.RegKey.KEYOPEN_ALL could not be converted

com.ms.lang.RegKey.KEYOPEN_CREATE could not be converted

com.ms.lang.RegKey.KEYOPEN_READ could not be converted

com.ms.lang.RegKey.KEYOPEN_WRITE could not be converted

com.ms.lang.RegKey.loadKey could not be converted

com.ms.lang.RegKey.queryInfo could not be converted

com.ms.lang.RegKey.replace could not be converted

com.ms.lang.RegKey.restore could not be converted

com.ms.lang.RegKey.unload could not be converted

com.ms.lang.RegQueryInfo could not be converted

com.ms.lang.SystemThread could not be converted

com.ms.lang.SystemX.arrayCompare could not be converted

com.ms.lang.SystemX.blockcopy could not be converted

com.ms.lang.SystemX.exitProcessAfterMainThreadReturns could not be converted

com.ms.lang.SystemX.getDeclaredMethodFromSignature could not be converted

com.ms.lang.SystemX.getDefaultInputManager could not be converted

com.ms.lang.SystemX.getInputManager could not be converted

com.ms.lang.SystemX.getKeyboardLanguageName could not be converted

com.ms.lang.SystemX.getKeyboardLanguages could not be converted

com.ms.lang.SystemX.getMethodFromSignature could not be converted

com.ms.lang.SystemX.getNativeServices could not be converted

com.ms.lang.SystemX.isLocalCharDBCSLeadByte could not be converted

com.ms.lang.SystemX.JavaStringToLocalString could not be converted

com.ms.lang.SystemX.LocalStringToJavaString could not be converted

com.ms.lang.SystemX.setInputManager could not be converted

com.ms.lang.SystemX.setKeyboardLanguage could not be converted

com.ms.lang.VerifyErrorEx could not be converted

Com.ms.mtx Error Messages

com.ms.mtx.AppServer could not be converted

com.ms.mtx.Context.createObject could not be converted

com.ms.mtx.Context.disableCommit could not be converted

com.ms.mtx.Context.enableCommit could not be converted

com.ms.mtx.Context.getContextId could not be converted

com.ms.mtx.Context.getDeactivateOnReturn could not be converted

com.ms.mtx.Context.getDirectCallerName could not be converted

com.ms.mtx.Context.getDirectCreatorName could not be converted

com.ms.mtx.Context.getMyTransactionVote could not be converted

com.ms.mtx.Context.getObjectContext could not be converted

com.ms.mtx.Context.getOriginalCallerName could not be converted

com.ms.mtx.Context.getOriginalCreatorName could not be converted

com.ms.mtx.Context.getProperty could not be converted

com.ms.mtx.Context.getPropertyNames could not be converted

com.ms.mtx.Context.getSafeRef could not be converted

com.ms.mtx.Context.getTransaction could not be converted

com.ms.mtx.Context.getTransactionId could not be converted

com.ms.mtx.Context.isCallerInRole could not be converted

com.ms.mtx.Context.isInTransaction could not be converted

com.ms.mtx.Context.isSecurityEnabled could not be converted

com.ms.mtx.Context.setAbort could not be converted

com.ms.mtx.Context.setComplete could not be converted

com.ms.mtx.Context.setDeactivateOnReturn could not be converted

com.ms.mtx.Context.setMyTransactionVote could not be converted

com.ms.mtx.Context.TxAbort could not be converted

com.ms.mtx.Context.TxCommit could not be converted

com.ms.mtx.IContextState.GetDeactivateOnReturn could not be converted

com.ms.mtx.IContextState.GetMyTransactionVote could not be converted

com.ms.mtx.IContextState.iid could not be converted

com.ms.mtx.IContextState.SetDeactivateOnReturn could not be converted

com.ms.mtx.IContextState.SetMyTransactionVote could not be converted

com.ms.mtx.IEnumNames could not be converted

com.ms.mtx.IGetContextProperties could not be converted

com.ms.mtx.IMTxAS could not be converted

com.ms.mtx.IMTxAS.GetObjectContext could not be converted

com.ms.mtx.IMTxAS.iid could not be converted

com.ms.mtx.IObjectContext.CreateInstance could not be converted

com.ms.mtx.IObjectContext.iid could not be converted

com.ms.mtx.IObjectContextInfo.iid could not be converted

com.ms.mtx.IObjectControl.iid could not be converted

com.ms.mtx.IObjectControl could not be converted

com.ms.mtx.ISecurityCallContext.getItem could not be converted

com.ms.mtx.ISecurityCallContext.iid could not be converted

com.ms.mtx.ISecurityCallersColl.iid could not be converted

com.ms.mtx.ISecurityIdentityColl.getItem could not be converted

com.ms.mtx.ISecurityIdentityColl.iid could not be converted

com.ms.mtx.SecurityProperty.GetDirectCallerName could not be converted

com.ms.mtx.SecurityProperty.GetOriginalCallerName could not be converted

com.ms.mtx.ISharedProperty.iid could not be converted

com.ms.mtx.ISharedPropertyGroup.iid could not be converted

com.ms.mtx.ISharedPropertyGroupManager.iid could not be converted

com.ms.mtx.ITransactionContextEx could not be converted

com.ms.mtx.MTx could not be converted

com.ms.mtx.ObjectContext.CreateInstance could not be converted

com.ms.mtx.ObjectContext.DisableCommit could not be converted

com.ms.mtx.ObjectContext.EnableCommit could not be converted

com.ms.mtx.ObjectContext.get_NewEnum could not be converted

com.ms.mtx.ObjectContext.getCount could not be converted

com.ms.mtx.ObjectContext.getItem could not be converted

com.ms.mtx.ObjectContext.getSecurity could not be converted

com.ms.mtx.ObjectContext.iid could not be converted

com.ms.mtx.ObjectContext.IsCallerInRole could not be converted

com.ms.mtx.ObjectContext.IsInTransaction could not be converted

com.ms.mtx.ObjectContext.IsSecurityEnabled could not be converted

com.ms.mtx.ObjectContext.SetAbort could not be converted

com.ms.mtx.ObjectContext.SetComplete could not be converted

com.ms.mtx.SecurityCallContext.getCallers could not be converted

com.ms.mtx.SecurityCallContext.getDirectCaller could not be converted

com.ms.mtx.SecurityCallContext.getNumCallers could not be converted

com.ms.mtx.SecurityCallContext.getProperty could not be converted

com.ms.mtx.SecurityCallContext.getPropertyNames could not be converted

com.ms.mtx.SecurityCallContext.isUserInRole could not be converted

com.ms.mtx.SecurityCaller could not be converted

com.ms.mtx.SecurityProperty.GetDirectCreatorName could not be converted

com.ms.mtx.SecurityProperty.GetOriginalCreatorName could not be converted

com.ms.mtx.SecurityProperty.iid could not be converted

com.ms.mtx.SharedPropertyGroupManager.clsid could not be converted

com.ms.mtx.SharedPropertyGroupManager.get_NewEnum could not be converted

com.ms.mtx.TransactionContextEx could not be converted

Com.ms.object Error Messages

com.ms.object.Category could not be converted

com.ms.object.dragdrop.DragHandler could not be converted

com.ms.object.dragdrop.DragHelper could not be converted

com.ms.object.dragdrop.DragProxy could not be converted

com.ms.object.dragdrop.DragSession.getDragModifiers could not be converted

com.ms.object.dragdrop.DragSource.DEFAULT_ACTION could not be converted

com.ms.object.dragdrop.DragSource.queryDragCursor could not be converted

com.ms.object.dragdrop.DragSource.queryDragStatus could not be converted

com.ms.object.IServiceObjectProvider could not be converted

com.ms.object.ISite could not be converted

com.ms.object.ISiteable could not be converted

com.ms.object.MetaObject could not be converted

com.ms.object.ObjectBag could not be converted

com.ms.object.SimpleTransferSession could not be converted

com.ms.object.TransferSession could not be converted

Com.ms.ui Error Messages

com.ms.ui.AwtUICheckButton.getBase could not be converted

com.ms.ui.AwtUIChoice.add could not be converted

com.ms.ui.AwtUIChoice.addSelectedIndex could not be converted

com.ms.ui.AwtUIChoice.getBase could not be converted

com.ms.ui.AwtUIMessageBox could not be converted

com.ms.ui.AwtUIProgress.getBase could not be converted

com.ms.ui.AwtUIPushButton.getBase could not be converted

com.ms.ui.AwtUIRadioButton.getBase could not be converted

com.ms.ui.AwtUIScrollBar.getBase could not be converted

com.ms.ui.AwtUIStatus.getBase could not be converted

com.ms.ui.event.UIKeyEvent.CHAR_UNDEFINED could not be converted

com.ms.ui.event.UIKeyEvent.getKeyChar could not be converted

com.ms.ui.event.UIKeyEvent.getKeyCode could not be converted

com.ms.ui.event.UIKeyEvent.getOldEventKey could not be converted

com.ms.ui.event.UIKeyEvent.KEY_EVENT_BASE could not be converted

com.ms.ui.event.UIKeyEvent.KEY_PRESSED could not be converted

com.ms.ui.event.UIKeyEvent.KEY_RELEASED could not be converted

com.ms.ui.event.UIKeyEvent.KEY_TYPED could not be converted

com.ms.ui.event.UIKeyEvent.VK_BACK_QUOTE could not be converted

com.ms.ui.event.UIKeyEvent.VK_EQUALS could not be converted

com.ms.ui.event.UIKeyEvent.VK_META could not be converted

com.ms.ui.event.UIKeyEvent.VK_UNDEFINED could not be converted

com.ms.ui.IUILayoutManager could not be converted

com.ms.ui.UIBarLayout could not be converted

com.ms.ui.UIBorderLayout could not be converted

com.ms.ui.UIButtonBar could not be converted

com.ms.ui.UICardLayout could not be converted

com.ms.ui.UIComponent.getToolkit could not be converted

com.ms.ui.UIFixedFlowLayout could not be converted

com.ms.ui.UIFlowLayout could not be converted

com.ms.ui.UIGridBagConstraints could not be converted

com.ms.ui.UIGridBagLayout could not be converted

com.ms.ui.UIGridLayout could not be converted

com.ms.ui.UIGroup.UIGroup could not be converted

com.ms.ui.UILayoutManager could not be converted

com.ms.ui.UIMessageBox could not be converted

com.ms.ui.UIRadioGroup.UIRadioGroup could not be converted

com.ms.ui.UIRowLayout could not be converted

com.ms.ui.UIScroll could not be converted

com.ms.ui.UISelector could not be converted

com.ms.ui.UISlider.add could not be converted

com.ms.ui.UISlider.clearSelection could not be converted

com.ms.ui.UISlider.getLayoutComponent could not be converted

com.ms.ui.UISlider.getSelectionEnd could not be converted

com.ms.ui.UISlider.getSelectionStart could not be converted

com.ms.ui.UISlider.setSelection could not be converted

com.ms.ui.UISlider.setSelectionEnd could not be converted

com.ms.ui.UISlider.setSelectionStart could not be converted

com.ms.ui.UISpinner could not be converted

com.ms.ui.UISpinnerEdit.CENTER could not be converted

com.ms.ui.UIText.paint could not be converted

com.ms.ui.UIText.setHot could not be converted

com.ms.ui.UITreeLayout could not be converted

com.ms.ui.Version could not be converted

com.ms.ui.UIWindow.pack could not be converted

Com.ms.util Error Messages

com.ms.util.ArraySort.compare could not be converted

com.ms.util.ArraySort.sort could not be converted

com.ms.util.ArraySort.swap could not be converted

com.ms.util.EventLog.reportEvent could not be converted

com.ms.util.HTMLTokenizer could not be converted

com.ms.util.IIntRangeComparator could not be converted

com.ms.util.IncludeExcludeIntRanges could not be converted

com.ms.util.IncludeExcludeWildcards could not be converted

com.ms.util.ini.IniFile could not be converted

com.ms.util.ini.IniSection could not be converted

com.ms.util.ini.IniSyntaxErrorException could not be converted

com.ms.util.IntRanges.compare could not be converted

com.ms.util.IntRanges.compareSet could not be converted

com.ms.util.IntRanges.ComparisonResulttoString could not be converted

com.ms.util.IntRanges.condense could not be converted

com.ms.util.IntRanges.intersect could not be converted

com.ms.util.IntRanges.IntRanges could not be converted

com.ms.util.IntRanges.invertComparisonResult could not be converted

com.ms.util.IntRanges.lock could not be converted

com.ms.util.IntRanges.parse could not be converted

com.ms.util.IntRanges.removeRange could not be converted

com.ms.util.IntRanges.removeRanges could not be converted

com.ms.util.IntRanges.removeSingleton could not be converted

com.ms.util.IntRanges.sort could not be converted

com.ms.util.IntRanges.unlock could not be converted

com.ms.util.IWildcardExpressionComparator could not be converted

com.ms.util.OrdinalMap.insertInCognate could not be converted

com.ms.util.OrdinalMap.moveCognate could not be converted

com.ms.util.OrdinalMap.newCognate could not be converted

com.ms.util.OrdinalMap.removeFromCognate could not be converted

com.ms.util.ProvideSetComparisonInfo could not be converted

com.ms.util.Queue.capacity could not be converted

com.ms.util.Queue.elementFromHead could not be converted

com.ms.util.Queue.elementFromTail could not be converted

com.ms.util.Queue.hasMoreElements could not be converted

com.ms.util.Queue.nextElement could not be converted

com.ms.util.Queue.setCapacity could not be converted

com.ms.util.Queue.toString could not be converted

com.ms.util.SetComparer could not be converted

com.ms.util.SetComparison could not be converted

com.ms.util.SetTemplate.foundAt could not be converted

com.ms.util.SetTemplate.foundCognate could not be converted

com.ms.util.SetTemplate.foundIn could not be converted

com.ms.util.SetTemplate.generation could not be converted

com.ms.util.SetTemplate.insertInCognate could not be converted

com.ms.util.SetTemplate.keys could not be converted

com.ms.util.SetTemplate.locate could not be converted

com.ms.util.SetTemplate.moveCognate could not be converted

com.ms.util.SetTemplate.newCognate could not be converted

com.ms.util.SetTemplate.rehash could not be converted

com.ms.util.SetTemplate.removeFromCognate could not be converted

com.ms.util.SetTemplate.reportFinds could not be converted

com.ms.util.SetTemplate.reportRequests could not be converted

com.ms.util.SetTemplate.reportUnits could not be converted

com.ms.util.SetTemplate.SetTemplate could not be converted

com.ms.util.Sort.compare could not be converted

com.ms.util.Sort.swap could not be converted

com.ms.util.StringComparison.ascending could not be converted

com.ms.util.StringComparison.descending could not be converted

com.ms.util.SystemVersionManager could not be converted

com.ms.util.Task could not be converted

com.ms.util.TaskManager could not be converted

com.ms.util.ThreadLocalStorage could not be converted

com.ms.util.Timer.getUserID could not be converted

com.ms.util.Timer.Timer could not be converted

com.ms.util.TimerEvent.getSource could not be converted

com.ms.util.TimerEvent.getTime could not be converted

com.ms.util.TimerEvent.getUserID could not be converted

com.ms.util.TimerListener could not be converted

com.ms.util.UnsignedIntRanges.intersect could not be converted

com.ms.util.UnsignedIntRanges.removeRange could not be converted

com.ms.util.UnsignedIntRanges.UnsignedIntRanges could not be converted

com.ms.util.VectorSort.compare could not be converted

com.ms.util.VectorSort.swap could not be converted

com.ms.util.WildcardExpression could not be converted

com.ms.util.zip.ZipInputStreamEx could not be converted

com.ms.util.zip.ZipOutputStreamEx could not be converted

Com.ms.wfc.app Error Messages

com.ms.wfc.app.Application.addOnSettingChange could not be converted

com.ms.wfc.app.Application.addOnSystemShutdown could not be converted

com.ms.wfc.app.Application.allocThreadStorage could not be converted

com.ms.wfc.app.Application.createThread could not be converted

com.ms.wfc.app.Application.doEvents could not be converted

com.ms.wfc.app.Application.freeThreadStorage could not be converted

com.ms.wfc.app.Application.getParkingForm could not be converted

com.ms.wfc.app.Application.getThreadStorage could not be converted

com.ms.wfc.app.Application.removeOnSettingChange could not be converted

com.ms.wfc.app.Application.removeOnSystemShutdown could not be converted

com.ms.wfc.app.Application.runDialog could not be converted

com.ms.wfc.app.Application.setThreadStorage could not be converted

com.ms.wfc.app.CharacterSet could not be converted

com.ms.wfc.app.Clipboard.Clipboard could not be converted

com.ms.wfc.app.DataFormats.DataFormats could not be converted

com.ms.wfc.app.DataFormats.Format.Format could not be converted

com.ms.wfc.app.DataFormats.Format.win32Handle could not be converted

com.ms.wfc.app.DataObject.OleDAdvise could not be converted

com.ms.wfc.app.DataObject.OleDUnadvise could not be converted

com.ms.wfc.app.DataObject.OleEnumDAdvise could not be converted

com.ms.wfc.app.DataObject.OleEnumFormatEtc could not be converted

com.ms.wfc.app.DataObject.OleGetCanonicalFormatEtc could not be converted

com.ms.wfc.app.DataObject.OleGetData could not be converted

com.ms.wfc.app.DataObject.OleGetDataHere could not be converted

com.ms.wfc.app.DataObject.OleQueryGetData could not be converted

com.ms.wfc.app.DataObject.OleSetData could not be converted

com.ms.wfc.app.IMessageFilter.postFilterMessage could not be converted

com.ms.wfc.app.KeywordVk could not be converted

com.ms.wfc.app.Locale.CalendarType could not be converted

com.ms.wfc.app.Locale.DateFormatOrder could not be converted

com.ms.wfc.app.Locale.getCalendarType could not be converted

com.ms.wfc.app.Locale.getCenturyFormat could not be converted

com.ms.wfc.app.Locale.getCharacterSet could not be converted

com.ms.wfc.app.Locale.getCompareIgnoreCase could not be converted

com.ms.wfc.app.Locale.getCompareIgnoreKana could not be converted

com.ms.wfc.app.Locale.getCompareIgnoreKashida could not be converted

com.ms.wfc.app.Locale.getCompareIgnoreNonSpace could not be converted

com.ms.wfc.app.Locale.getCompareIgnoreSymbols could not be converted

com.ms.wfc.app.Locale.getCompareIgnoreWidth could not be converted

com.ms.wfc.app.Locale.getCountryCode could not be converted

com.ms.wfc.app.Locale.getDefaultCountry could not be converted

com.ms.wfc.app.Locale.getEnglishCurrencyName could not be converted

com.ms.wfc.app.Locale.getLeadingZero could not be converted

com.ms.wfc.app.Locale.getNativeDigits could not be converted

com.ms.wfc.app.Locale.getSortId could not be converted

com.ms.wfc.app.Locale.getSupportedLocales could not be converted

com.ms.wfc.app.Locale.Languages.Locale.Languages could not be converted

com.ms.wfc.app.Locale.Languages.RHAETO_ROMAN could not be converted

com.ms.wfc.app.Locale.LeadingZeros could not be converted

com.ms.wfc.app.Locale.Locale could not be converted

com.ms.wfc.app.Locale.MeasurementSystem could not be converted

com.ms.wfc.app.Locale.NegativeNumberMode.Locale.NegativeNumberMode could not be converted

com.ms.wfc.app.Locale.OptionalCalendarType could not be converted

com.ms.wfc.app.Locale.PositiveCurrencyMode.Locale.PositiveCurrencyMode could not be converted

com.ms.wfc.app.Locale.setCalendarType could not be converted

com.ms.wfc.app.Locale.setCompareIgnoreCase could not be converted

com.ms.wfc.app.Locale.setCompareIgnoreKana could not be converted

com.ms.wfc.app.Locale.setCompareIgnoreKashida could not be converted

com.ms.wfc.app.Locale.setCompareIgnoreNonSpace could not be converted

com.ms.wfc.app.Locale.setCompareIgnoreSymbols could not be converted

com.ms.wfc.app.Locale.setCompareIgnoreWidth could not be converted

com.ms.wfc.app.Locale.setFirstDayOfWeek could not be converted

com.ms.wfc.app.Locale.setFirstWeekOfYear could not be converted

com.ms.wfc.app.Locale.setLeadingZero could not be converted

com.ms.wfc.app.Locale.setListSeparator could not be converted

com.ms.wfc.app.Locale.setMeasurementSystem could not be converted

com.ms.wfc.app.Locale.Sort could not be converted

com.ms.wfc.app.Locale.SubLanguages.CHINESE_SIMPLIFIED could not be converted

com.ms.wfc.app.Locale.SubLanguages.CHINESE_TRADITIONAL could not be converted

com.ms.wfc.app.Locale.SubLanguages.DUTCH_BELGIAN could not be converted

com.ms.wfc.app.Locale.SubLanguages.ENGLISH_AUS could not be converted

com.ms.wfc.app.Locale.SubLanguages.ENGLISH_CAN could not be converted

com.ms.wfc.app.Locale.SubLanguages.ENGLISH_EIRE could not be converted

com.ms.wfc.app.Locale.SubLanguages.ENGLISH_NZ could not be converted

com.ms.wfc.app.Locale.SubLanguages.ENGLISH_UK could not be converted

com.ms.wfc.app.Locale.SubLanguages.ENGLISH_US could not be converted

com.ms.wfc.app.Locale.SubLanguages.FRENCH_BELGIAN could not be converted

com.ms.wfc.app.Locale.SubLanguages.FRENCH_CANADIAN could not be converted

com.ms.wfc.app.Locale.SubLanguages.FRENCH_SWISS could not be converted

com.ms.wfc.app.Locale.SubLanguages.GERMAN_AUSTRIAN could not be converted

com.ms.wfc.app.Locale.SubLanguages.GERMAN_SWISS could not be converted

com.ms.wfc.app.Locale.SubLanguages.ITALIAN_SWISS could not be converted

com.ms.wfc.app.Locale.SubLanguages.Locale.SubLanguages could not be converted

com.ms.wfc.app.Locale.SubLanguages.NORWEGIAN_BOKMAL could not be converted

com.ms.wfc.app.Locale.SubLanguages.NORWEGIAN_NYNORSK could not be converted

com.ms.wfc.app.Locale.SubLanguages.PORTUGUESE_BRAZILIAN could not be converted

com.ms.wfc.app.Locale.SubLanguages.SERBO_CROATIAN_CYRILLIC could not be converted

com.ms.wfc.app.Locale.SubLanguages.SERBO_CROATIAN_LATIN could not be converted

com.ms.wfc.app.Locale.SubLanguages.SPANISH_MEXICAN could not be converted

com.ms.wfc.app.Locale.SubLanguages.SPANISH_MODERN could not be converted

com.ms.wfc.app.Locale.SubLanguages.SubLanguages could not be converted

com.ms.wfc.app.Locale.SubLanguages.SYS_DEFAULT could not be converted

com.ms.wfc.app.Message.free could not be converted

com.ms.wfc.app.MethodInvoker.MethodInvoker could not be converted

com.ms.wfc.app.Registry.Registry could not be converted

com.ms.wfc.app.RegistryKey.getBaseKey could not be converted

com.ms.wfc.app.SendKeysHookProc could not be converted

com.ms.wfc.app.SKEvent could not be converted

com.ms.wfc.app.SystemInformation.getArrange could not be converted

com.ms.wfc.app.Time.getConstructorArgs

com.ms.wfc.app.Time.getExtension could not be converted

com.ms.wfc.app.Time.save could not be converted

com.ms.wfc.app.Time.Time could not be converted

com.ms.wfc.app.Time.toSystemTime could not be converted

com.ms.wfc.app.Time.toVariant could not be converted

com.ms.wfc.app.Window.callback could not be converted

Com.ms.wfc.ax Error Messages

com.ms.wfc.ax could not be converted

com.ms.wfc.ax.IPersist.iid could not be converted

Com.ms.wfc.core Error Messages

com.ms.wfc.core.ArrayDialog could not be converted

com.ms.wfc.core.ArrayEditor.createNewInstance could not be converted

com.ms.wfc.core.ArrayEditor.defaultPropInfo could not be converted

com.ms.wfc.core.ArrayEditor.editValue could not be converted

com.ms.wfc.core.ArrayEditor.getBaseName could not be converted

com.ms.wfc.core.ArrayEditor.getDisplayText could not be converted

com.ms.wfc.core.ArrayEditor.getTextFromValue could not be converted

com.ms.wfc.core.ArrayEditor.getTypeDescription could not be converted

com.ms.wfc.core.BooleanEditor.getStyle could not be converted

com.ms.wfc.core.CategoryAttribute.Position could not be converted

com.ms.wfc.core.Component.appendEventHandlers could not be converted

com.ms.wfc.core.Component.componentChanged could not be converted

com.ms.wfc.core.Component.getDisposing could not be converted

com.ms.wfc.core.Component.getResource could not be converted

com.ms.wfc.core.Component.getService could not be converted

com.ms.wfc.core.ComponentInfo.getClassInfo could not be converted

com.ms.wfc.core.ComponentInfo.getDefaultEventInfo could not be converted

com.ms.wfc.core.ComponentInfo.getExtenders could not be converted

com.ms.wfc.core.ComponentManager.createClassInfo could not be converted

com.ms.wfc.core.ComponentManager.createValueEditor could not be converted

com.ms.wfc.core.ComponentManager.getValueEditor could not be converted

com.ms.wfc.core.ComponentManager.registerClassInfo could not be converted

com.ms.wfc.core.ComponentManager.registerValueEditorClass could not be converted

com.ms.wfc.core.ConstructorArg could not be converted

com.ms.wfc.core.CustomizerVerb.addOnVerbExecute could not be converted

com.ms.wfc.core.CustomizerVerb.CustomizerVerb could not be converted

com.ms.wfc.core.CustomizerVerb.getBitmap could not be converted

com.ms.wfc.core.CustomizerVerb.getData could not be converted

com.ms.wfc.core.CustomizerVerb.onVerbExecute could not be converted

com.ms.wfc.core.CustomizerVerb.performVerbExecute could not be converted

com.ms.wfc.core.CustomizerVerb.removeOnVerbExecute could not be converted

com.ms.wfc.core.CustomizerVerb.setBitmap could not be converted

com.ms.wfc.core.CustomizerVerb.setData could not be converted

com.ms.wfc.core.DescriptionAttribute.getDescription could not be converted

com.ms.wfc.core.DesignForm.DesignForm could not be converted

com.ms.wfc.core.DesignForm.getPage could not be converted

com.ms.wfc.core.DesignForm.getPageCount could not be converted

com.ms.wfc.core.DesignForm.showForm could not be converted

com.ms.wfc.core.DesignPage.getPageMessage could not be converted

com.ms.wfc.core.DesignPage.NULLVALUE could not be converted

com.ms.wfc.core.DesignPage.objects could not be converted

com.ms.wfc.core.DesignPage.onReadProperty could not be converted

com.ms.wfc.core.DesignPage.onWriteProperty could not be converted

com.ms.wfc.core.DesignPage.properties could not be converted

com.ms.wfc.core.DesignPage.setObjects could not be converted

com.ms.wfc.core.Enum.Enum could not be converted

com.ms.wfc.core.Event.Event could not be converted

com.ms.wfc.core.Event.extendedInfo could not be converted

com.ms.wfc.core.EventInfo.addEventHandler could not be converted

com.ms.wfc.core.EventInfo.getAddMethod could not be converted

com.ms.wfc.core.EventInfo.getMulticast could not be converted

com.ms.wfc.core.EventInfo.getRemoveMethod could not be converted

com.ms.wfc.core.EventInfo.getType could not be converted

com.ms.wfc.core.EventInfo.removeEventHandler could not be converted

com.ms.wfc.core.ExtenderInfo could not be converted

com.ms.wfc.core.FieldsEditor could not be converted

com.ms.wfc.core.IAttributes could not be converted

com.ms.wfc.core.IClassInfo could not be converted

com.ms.wfc.core.IComponent.dispose could not be converted

com.ms.wfc.core.IComponent.getChildOf could not be converted

com.ms.wfc.core.IComponentSite.componentChanged could not be converted

com.ms.wfc.core.IComponentSite.getDisposing could not be converted

com.ms.wfc.core.IConstructable could not be converted

com.ms.wfc.core.IContainer.dispose could not be converted

com.ms.wfc.core.IContainer.getComponent could not be converted

com.ms.wfc.core.IContainer.getComponents could not be converted

com.ms.wfc.core.ICustomizer.getDesignPages could not be converted

com.ms.wfc.core.ICustomizer.getVerbs could not be converted

com.ms.wfc.core.IDesignPage.getSize could not be converted

com.ms.wfc.core.IDesignPage.setObjects could not be converted

com.ms.wfc.core.IDesignPage.setSize could not be converted

com.ms.wfc.core.IEditorHost.getHostHandle could not be converted

com.ms.wfc.core.IEditorSite.getType could not be converted

com.ms.wfc.core.IEditorSite.getValueOwner could not be converted

com.ms.wfc.core.IEditorSite.getValueOwners could not be converted

com.ms.wfc.core.IResourceLoader could not be converted

com.ms.wfc.core.IResourceManager.containsProperty could not be converted

com.ms.wfc.core.IResourceManager.getNames could not be converted

com.ms.wfc.core.IResourceManager.getProperties could not be converted

com.ms.wfc.core.IResourceManager.saveAllProperties could not be converted

com.ms.wfc.core.IResourceManager.setProperty could not be converted

com.ms.wfc.core.IResourceManager.setResourceSet could not be converted

com.ms.wfc.core.IValueEditor.getConstantName could not be converted

com.ms.wfc.core.IValueEditor.getValueFromSubPropertyValues could not be converted

com.ms.wfc.core.IValueEditor.STYLE_IMMEDIATE could not be converted

com.ms.wfc.core.IValueEditor.STYLE_NOARRAYEXPANSION could not be converted

com.ms.wfc.core.IValueEditor.STYLE_NOARRAYMULTISELECT could not be converted

com.ms.wfc.core.IValueEditor.STYLE_NOEDITABLETEXT could not be converted

com.ms.wfc.core.IValueEditor.STYLE_PAINTVALUE could not be converted

com.ms.wfc.core.IValueEditor.STYLE_PROPERTIES could not be converted

com.ms.wfc.core.IValueEditor.STYLE_SHOWFIELDS could not be converted

com.ms.wfc.core.IValueEditor.STYLE_VALUES could not be converted

com.ms.wfc.core.NonPersistableAttribute could not be converted

com.ms.wfc.core.PropertyInfo.canResetValue could not be converted

com.ms.wfc.core.PropertyInfo.getGetMethod could not be converted

com.ms.wfc.core.PropertyInfo.getIsReadOnly could not be converted

com.ms.wfc.core.PropertyInfo.getResetMethod could not be converted

com.ms.wfc.core.PropertyInfo.getSetMethod could not be converted

com.ms.wfc.core.PropertyInfo.getShouldPersistMethod could not be converted

com.ms.wfc.core.PropertyInfo.getType could not be converted

com.ms.wfc.core.PropertyInfo.getValue could not be converted

com.ms.wfc.core.PropertyInfo.getValueEditor could not be converted

com.ms.wfc.core.PropertyInfo.resetValue could not be converted

com.ms.wfc.core.PropertyInfo.setValue could not be converted

com.ms.wfc.core.PropertyInfo.shouldPersistValue could not be converted

com.ms.wfc.core.ResourceManager.containsProperty could not be converted

com.ms.wfc.core.ResourceManager.EXTENSION could not be converted

com.ms.wfc.core.ResourceManager.getLocaleFileNameFromBase could not be converted

com.ms.wfc.core.ResourceManager.getNames could not be converted

com.ms.wfc.core.ResourceManager.getProperties could not be converted

com.ms.wfc.core.ResourceManager.getResourceLoader could not be converted

com.ms.wfc.core.ResourceManager.ResourceManager could not be converted

com.ms.wfc.core.ResourceManager.save could not be converted

com.ms.wfc.core.ResourceManager.saveAllProperties could not be converted

com.ms.wfc.core.ResourceManager.setBaseName could not be converted

com.ms.wfc.core.ResourceManager.setProperty could not be converted

com.ms.wfc.core.ResourceManager.setResourceLoader could not be converted

com.ms.wfc.core.ResourceManager.setResourceSet could not be converted

com.ms.wfc.core.ResourceReader could not be converted

com.ms.wfc.core.ResourceWriter could not be converted

com.ms.wfc.core.ShowInToolboxAttribute.NO could not be converted

com.ms.wfc.core.ShowInToolboxAttribute.YES could not be converted

com.ms.wfc.core.StringListDialog could not be converted

com.ms.wfc.core.StringListEditor could not be converted

com.ms.wfc.core.Sys could not be converted

com.ms.wfc.core.ValueEditor.editValue could not be converted

com.ms.wfc.core.ValueEditor.getValueFromSubPropertyValues could not be converted

com.ms.wfc.core.ValueEditorAttribute.valueEditorType could not be converted

com.ms.wfc.core.VerbExecuteEvent could not be converted

com.ms.wfc.core.VerbExecuteEventHandler could not be converted

com.ms.wfc.core.WFCException.getBaseException could not be converted

com.ms.wfc.core.WFCException.printStackTrace could not be converted

Com.ms.wfc.data Error Messages

com.ms.wfc.data.AdoEnums could not be converted

com.ms.wfc.data.AdoEvent could not be converted

com.ms.wfc.data.AdoProperties.AdoProperties could not be converted

com.ms.wfc.data.BooleanDataFormat could not be converted

com.ms.wfc.data.Connection.addOnBeginTransComplete could not be converted

com.ms.wfc.data.Connection.addOnCommitTransComplete could not be converted

com.ms.wfc.data.Connection.addOnConnectComplete could not be converted

com.ms.wfc.data.Connection.addOnDisconnect could not be converted

com.ms.wfc.data.Connection.addOnExecuteComplete could not be converted

com.ms.wfc.data.Connection.addOnInfoMessage could not be converted

com.ms.wfc.data.Connection.addOnRollbackTransComplete could not be converted

com.ms.wfc.data.Connection.addOnWillConnect could not be converted

com.ms.wfc.data.Connection.addOnWillExecute could not be converted

com.ms.wfc.data.Connection.Connection could not be converted

com.ms.wfc.data.Connection.getPeer could not be converted

com.ms.wfc.data.Connection.removeOnBeginTransComplete could not be converted

com.ms.wfc.data.Connection.removeOnCommitTransComplete could not be converted

com.ms.wfc.data.Connection.removeOnConnectComplete could not be converted

com.ms.wfc.data.Connection.removeOnDisconnect could not be converted

com.ms.wfc.data.Connection.removeOnExecuteComplete could not be converted

com.ms.wfc.data.Connection.removeOnInfoMessage could not be converted

com.ms.wfc.data.Connection.removeOnRollbackTransComplete could not be converted

com.ms.wfc.data.Connection.removeOnWillConnect could not be converted

com.ms.wfc.data.Connection.removeOnWillExecute could not be converted

com.ms.wfc.data.ConnectionEvent could not be converted

com.ms.wfc.data.ConnectionEventHandler could not be converted

com.ms.wfc.data.DataFormat could not be converted

com.ms.wfc.data.DataFormat.FormatWrapper could not be converted

com.ms.wfc.data.DateDataFormat could not be converted

com.ms.wfc.data.DateFormat.valid could not be converted

com.ms.wfc.data.dsl.<ClassName>.clsid could not be converted

com.ms.wfc.data.dsl.<ClassName>.iid could not be converted

com.ms.wfc.data.dsl._COAUTHIDENTITY._COAUTHIDENTITY could not be converted

com.ms.wfc.data.dsl._COAUTHINFO._COAUTHINFO could not be converted

com.ms.wfc.data.dsl._COSERVERINFO._COSERVERINFO could not be converted

com.ms.wfc.data.dsl._RemotableHandle._RemotableHandle could not be converted

com.ms.wfc.data.dsl.tagMULTI_QI could not be converted

com.ms.wfc.data.EventHandlerList.clear could not be converted

com.ms.wfc.data.EventHandlerList.getList could not be converted

com.ms.wfc.data.EventsListener could not be converted

com.ms.wfc.data.Field.getDataTimestamp could not be converted

com.ms.wfc.data.Field.getDispatch could not be converted

com.ms.wfc.data.Field.getGuid could not be converted

com.ms.wfc.data.Field.getObject could not be converted

com.ms.wfc.data.Field.getTime could not be converted

com.ms.wfc.data.Field.getTimestamp could not be converted

com.ms.wfc.data.Field.setDataDate could not be converted

com.ms.wfc.data.Field.setDispatch could not be converted

com.ms.wfc.data.Field.setGuid could not be converted

com.ms.wfc.data.Field.setTime could not be converted

com.ms.wfc.data.Field.setTimestamp could not be converted

com.ms.wfc.data.IDataFormat could not be converted

com.ms.wfc.data.IDataSource.addDataSourceListener could not be converted

com.ms.wfc.data.IDataSource.getDataMember could not be converted

com.ms.wfc.data.IDataSource.getDataMemberCount could not be converted

com.ms.wfc.data.IDataSource.getDataMemberName could not be converted

com.ms.wfc.data.IDataSource.iid could not be converted

com.ms.wfc.data.IDataSource.removeDataSourceListener could not be converted

com.ms.wfc.data.IDataSourceListener could not be converted

com.ms.wfc.data.IMarshal.iid could not be converted

com.ms.wfc.data.IPersist.iid could not be converted

com.ms.wfc.data.IPersistStream.GetSizeMax could not be converted

com.ms.wfc.data.IPersistStream.iid could not be converted

com.ms.wfc.data.IPersistStream.Load could not be converted

com.ms.wfc.data.NumberDataFormat could not be converted

com.ms.wfc.data.ObjectProxy.call could not be converted

com.ms.wfc.data.rds.<ClassName>.clsid could not be converted

com.ms.wfc.data.rds.<ClassName>.iid could not be converted

com.ms.wfc.data.Recordset.addDataSourceListener could not be converted

com.ms.wfc.data.Recordset.addOnEndOfRecordset could not be converted

com.ms.wfc.data.Recordset.addOnFetchComplete could not be converted

com.ms.wfc.data.Recordset.addOnFetchProgress could not be converted

com.ms.wfc.data.Recordset.addOnFieldChangeComplete could not be converted

com.ms.wfc.data.Recordset.addOnMoveComplete could not be converted

com.ms.wfc.data.Recordset.addOnRecordChangeComplete could not be converted

com.ms.wfc.data.Recordset.addOnRecordsetChangeComplete could not be converted

com.ms.wfc.data.Recordset.addOnWillChangeField could not be converted

com.ms.wfc.data.Recordset.addOnWillChangeRecord could not be converted

com.ms.wfc.data.Recordset.addOnWillChangeRecordset could not be converted

com.ms.wfc.data.Recordset.addOnWillMove could not be converted

com.ms.wfc.data.Recordset.DisconnectObject could not be converted

com.ms.wfc.data.Recordset.GetClassID could not be converted

com.ms.wfc.data.Recordset.getCommand could not be converted

com.ms.wfc.data.Recordset.getDataMember could not be converted

com.ms.wfc.data.Recordset.getDataMemberCount could not be converted

com.ms.wfc.data.Recordset.getDataMemberName could not be converted

com.ms.wfc.data.Recordset.GetMarshalSizeMax could not be converted

com.ms.wfc.data.Recordset.getRecordset could not be converted

com.ms.wfc.data.Recordset.GetSizeMax could not be converted

com.ms.wfc.data.Recordset.GetUnmarshalClass could not be converted

com.ms.wfc.data.Recordset.IsDirty could not be converted

com.ms.wfc.data.Recordset.Load could not be converted

com.ms.wfc.data.Recordset.MarshalInterface could not be converted

com.ms.wfc.data.Recordset.nextRecordset could not be converted

com.ms.wfc.data.Recordset.open could not be converted

com.ms.wfc.data.Recordset(object).open could not be converted

com.ms.wfc.data.Recordset.Recordset(object) could not be converted

com.ms.wfc.data.Recordset.Recordset(IDataSource) could not be converted

com.ms.wfc.data.Recordset.release could not be converted

com.ms.wfc.data.Recordset.ReleaseMarshalData could not be converted

com.ms.wfc.data.Recordset.removeDataSourceListener could not be converted

com.ms.wfc.data.Recordset.removeOnEndOfRecordset could not be converted

com.ms.wfc.data.Recordset.removeOnFetchComplete could not be converted

com.ms.wfc.data.Recordset.removeOnFetchProgress could not be converted

com.ms.wfc.data.Recordset.removeOnFieldChangeComplete could not be converted

com.ms.wfc.data.Recordset.removeOnMoveComplete could not be converted

com.ms.wfc.data.Recordset.removeOnRecordChangeComplete could not be converted

com.ms.wfc.data.Recordset.removeOnRecordsetChangeComplete could not be converted

com.ms.wfc.data.Recordset.removeOnWillChangeField could not be converted

com.ms.wfc.data.Recordset.removeOnWillChangeRecord could not be converted

com.ms.wfc.data.Recordset.removeOnWillChangeRecordset could not be converted

com.ms.wfc.data.Recordset.removeOnWillMove could not be converted

com.ms.wfc.data.Recordset.setCommand could not be converted

com.ms.wfc.data.Recordset.UnmarshalInterface could not be converted

com.ms.wfc.data.RecordsetEvent could not be converted

com.ms.wfc.data.RecordsetEventHandler could not be converted

com.ms.wfc.data.StringManager.getString could not be converted

Com.ms.wfc.data.adodb Error Messages

com.ms.wfc.data.adodb.<ClassName>.clsid could not be converted

com.ms.wfc.data.adodb.<ClassName>.iid could not be converted

com.ms.wfc.data.adodb._Command.setActiveConnection could not be converted

com.ms.wfc.data.adodb.Command.setActiveConnection could not be converted

com.ms.wfc.data.adodb._Recordset.getCollect could not be converted

com.ms.wfc.data.adodb.Recordset.getCollect could not be converted

com.ms.wfc.data.adodb._Recordset.setCollect could not be converted

com.ms.wfc.data.adodb.Recordset.setCollect could not be converted

Com.ms.wfc.data.ui Error Messages

com.ms.wfc.data.ui.Column.getAllowSizing could not be converted

com.ms.wfc.data.ui.Column.getBackColor could not be converted

com.ms.wfc.data.ui.Column.getDataFormat could not be converted

com.ms.wfc.data.ui.Column.getDataType could not be converted

com.ms.wfc.data.ui.Column.getFont could not be converted

com.ms.wfc.data.ui.Column.getForeColor could not be converted

com.ms.wfc.data.ui.Column.getSelectedBackColor could not be converted

com.ms.wfc.data.ui.Column.getSelectedForeColor could not be converted

com.ms.wfc.data.ui.Column.getVisible could not be converted

com.ms.wfc.data.ui.Column.getWriteAllowed could not be converted

com.ms.wfc.data.ui.Column.setAllowSizing could not be converted

com.ms.wfc.data.ui.Column.setBackColor could not be converted

com.ms.wfc.data.ui.Column.setDataFormat could not be converted

com.ms.wfc.data.ui.Column.setFont could not be converted

com.ms.wfc.data.ui.Column.setForeColor could not be converted

com.ms.wfc.data.ui.Column.setIndex could not be converted

com.ms.wfc.data.ui.Column.setSelectedBackColor could not be converted

com.ms.wfc.data.ui.Column.setSelectedForeColor could not be converted

com.ms.wfc.data.ui.Column.setValue could not be converted

com.ms.wfc.data.ui.Column.setVisible could not be converted

com.ms.wfc.data.ui.Column.shouldPersistBackColor could not be converted

com.ms.wfc.data.ui.Column.shouldPersistFont could not be converted

com.ms.wfc.data.ui.Column.shouldPersistForeColor could not be converted

com.ms.wfc.data.ui.Column.shouldPersistSelectedBackColor could not be converted

com.ms.wfc.data.ui.Column.shouldPersistSelectedForeColor could not be converted

com.ms.wfc.data.ui.ColumnEditingEvent could not be converted

com.ms.wfc.data.ui.ColumnEditingEventHandler could not be converted

com.ms.wfc.data.ui.ColumnEvent could not be converted

com.ms.wfc.data.ui.ColumnEventHandler could not be converted

com.ms.wfc.data.ui.ColumnResizeEvent could not be converted

com.ms.wfc.data.ui.ColumnResizeEventHandler could not be converted

com.ms.wfc.data.ui.ColumnUpdatingEvent could not be converted

com.ms.wfc.data.ui.ColumnUpdatingEventHandler could not be converted

com.ms.wfc.data.ui.DataBinder could not be converted

com.ms.wfc.data.ui.DataBinder.Customizer could not be converted

com.ms.wfc.data.ui.DataBinder.GeneralPage could not be converted

com.ms.wfc.data.ui.DataBinding.bindTarget could not be converted

com.ms.wfc.data.ui.DataBinding.CallbackType could not be converted

com.ms.wfc.data.ui.DataBinding.commitChange could not be converted

com.ms.wfc.data.ui.DataBinding.DataBinding could not be converted

com.ms.wfc.data.ui.DataBinding.FalseArgument could not be converted

com.ms.wfc.data.ui.DataBinding.getConstructorArgs could not be converted

com.ms.wfc.data.ui.DataBinding.getDataFormat could not be converted

com.ms.wfc.data.ui.DataBinding.getFieldName could not be converted

com.ms.wfc.data.ui.DataBinding.getPropertyValue could not be converted

com.ms.wfc.data.ui.DataBinding.getTarget could not be converted

com.ms.wfc.data.ui.DataBinding.m_propertyChange could not be converted

com.ms.wfc.data.ui.DataBinding.onChanged could not be converted

com.ms.wfc.data.ui.DataBinding.onChanging could not be converted

com.ms.wfc.data.ui.DataBinding.onTargetDispose could not be converted

com.ms.wfc.data.ui.DataBinding.propertyChange could not be converted

com.ms.wfc.data.ui.DataBinding.refreshPropertyValue could not be converted

com.ms.wfc.data.ui.DataBinding.setDataFormat could not be converted

com.ms.wfc.data.ui.DataBinding.setFieldName could not be converted

com.ms.wfc.data.ui.DataBinding.setPropertyValue could not be converted

com.ms.wfc.data.ui.DataBinding.setTarget could not be converted

com.ms.wfc.data.ui.DataGrid.addColumn could not be converted

com.ms.wfc.data.ui.DataGrid.addOnColumnEdited could not be converted

com.ms.wfc.data.ui.DataGrid.addOnColumnEditing could not be converted

com.ms.wfc.data.ui.DataGrid.addOnColumnResize could not be converted

com.ms.wfc.data.ui.DataGrid.addOnColumnUpdated could not be converted

com.ms.wfc.data.ui.DataGrid.addOnColumnUpdating could not be converted

com.ms.wfc.data.ui.DataGrid.addOnDeleted could not be converted

com.ms.wfc.data.ui.DataGrid.addOnDeleting could not be converted

com.ms.wfc.data.ui.DataGrid.addOnError could not be converted

com.ms.wfc.data.ui.DataGrid.addOnHeaderClick could not be converted

com.ms.wfc.data.ui.DataGrid.addOnInserted could not be converted

com.ms.wfc.data.ui.DataGrid.addOnInserting could not be converted

com.ms.wfc.data.ui.DataGrid.addOnPositionChange could not be converted

com.ms.wfc.data.ui.DataGrid.addOnRowResize could not be converted

com.ms.wfc.data.ui.DataGrid.addOnScroll could not be converted

com.ms.wfc.data.ui.DataGrid.addOnSelChange could not be converted

com.ms.wfc.data.ui.DataGrid.addOnUpdated could not be converted

com.ms.wfc.data.ui.DataGrid.addOnUpdating could not be converted

com.ms.wfc.data.ui.DataGrid.Customizer could not be converted

com.ms.wfc.data.ui.DataGrid.DataGrid could not be converted

com.ms.wfc.data.ui.DataGrid.DataGridErrorDialog could not be converted

com.ms.wfc.data.ui.DataGrid.fireErrorEvent could not be converted

com.ms.wfc.data.ui.DataGrid.getAllowAddNew could not be converted

com.ms.wfc.data.ui.DataGrid.getAllowArrows could not be converted

com.ms.wfc.data.ui.DataGrid.getAllowDelete could not be converted

com.ms.wfc.data.ui.DataGrid.getAllowRowSizing could not be converted

com.ms.wfc.data.ui.DataGrid.getAllowUpdate could not be converted

com.ms.wfc.data.ui.DataGrid.getCurrentRow could not be converted

com.ms.wfc.data.ui.DataGrid.getCurrentRowModified could not be converted

com.ms.wfc.data.ui.DataGrid.getDisplayIndex could not be converted

com.ms.wfc.data.ui.DataGrid.getDynamicColumns could not be converted

com.ms.wfc.data.ui.DataGrid.getEnterAction could not be converted

com.ms.wfc.data.ui.DataGrid.getFirstRow could not be converted

com.ms.wfc.data.ui.DataGrid.getHeaderLineCount could not be converted

com.ms.wfc.data.ui.DataGrid.getRowHeight could not be converted

com.ms.wfc.data.ui.DataGrid.getScrollbars could not be converted

com.ms.wfc.data.ui.DataGrid.getSelectedColumns could not be converted

com.ms.wfc.data.ui.DataGrid.getSelectedRows could not be converted

com.ms.wfc.data.ui.DataGrid.getShowPhantom could not be converted

com.ms.wfc.data.ui.DataGrid.getTabAction could not be converted

com.ms.wfc.data.ui.DataGrid.getWrapCellPointer could not be converted

com.ms.wfc.data.ui.DataGrid.makeCurrentCellVisible could not be converted

com.ms.wfc.data.ui.DataGrid.MIN_COLUMN_WIDTH could not be converted

com.ms.wfc.data.ui.DataGrid.MIN_ROW_HEIGHT could not be converted

com.ms.wfc.data.ui.DataGrid.onColumnEdited could not be converted

com.ms.wfc.data.ui.DataGrid.onColumnEditing could not be converted

com.ms.wfc.data.ui.DataGrid.onColumnResize could not be converted

com.ms.wfc.data.ui.DataGrid.onColumnUpdated could not be converted

com.ms.wfc.data.ui.DataGrid.onColumnUpdating could not be converted

com.ms.wfc.data.ui.DataGrid.onDeleted could not be converted

com.ms.wfc.data.ui.DataGrid.onDeleting could not be converted

com.ms.wfc.data.ui.DataGrid.onError could not be converted

com.ms.wfc.data.ui.DataGrid.onHeaderClick could not be converted

com.ms.wfc.data.ui.DataGrid.onInserted could not be converted

com.ms.wfc.data.ui.DataGrid.onInserting could not be converted

com.ms.wfc.data.ui.DataGrid.onPositionChange could not be converted

com.ms.wfc.data.ui.DataGrid.onRowResize could not be converted

com.ms.wfc.data.ui.DataGrid.onScroll could not be converted

com.ms.wfc.data.ui.DataGrid.onSelChange could not be converted

com.ms.wfc.data.ui.DataGrid.onUpdated could not be converted

com.ms.wfc.data.ui.DataGrid.onUpdating could not be converted

com.ms.wfc.data.ui.DataGrid.rebind could not be converted

com.ms.wfc.data.ui.DataGrid.removeOnColumnEdited could not be converted

com.ms.wfc.data.ui.DataGrid.removeOnColumnEditing could not be converted

com.ms.wfc.data.ui.DataGrid.removeOnColumnResize could not be converted

com.ms.wfc.data.ui.DataGrid.removeOnColumnUpdated could not be converted

com.ms.wfc.data.ui.DataGrid.removeOnColumnUpdating could not be converted

com.ms.wfc.data.ui.DataGrid.removeOnDeleted could not be converted

com.ms.wfc.data.ui.DataGrid.removeOnDeleting could not be converted

com.ms.wfc.data.ui.DataGrid.removeOnError could not be converted

com.ms.wfc.data.ui.DataGrid.removeOnHeaderClick could not be converted

com.ms.wfc.data.ui.DataGrid.removeOnInserted could not be converted

com.ms.wfc.data.ui.DataGrid.removeOnInserting could not be converted

com.ms.wfc.data.ui.DataGrid.removeOnPositionChange could not be converted

com.ms.wfc.data.ui.DataGrid.removeOnRowResize could not be converted

com.ms.wfc.data.ui.DataGrid.removeOnScroll could not be converted

com.ms.wfc.data.ui.DataGrid.removeOnSelChange could not be converted

com.ms.wfc.data.ui.DataGrid.removeOnUpdated could not be converted

com.ms.wfc.data.ui.DataGrid.removeOnUpdating could not be converted

com.ms.wfc.data.ui.DataGrid.rowBookmark could not be converted

com.ms.wfc.data.ui.DataGrid.rowTop could not be converted

com.ms.wfc.data.ui.DataGrid.scroll could not be converted

com.ms.wfc.data.ui.DataGrid.setAllowAddNew could not be converted

com.ms.wfc.data.ui.DataGrid.setAllowArrows could not be converted

com.ms.wfc.data.ui.DataGrid.setAllowDelete could not be converted

com.ms.wfc.data.ui.DataGrid.setAllowRowSizing could not be converted

com.ms.wfc.data.ui.DataGrid.setAllowUpdate could not be converted

com.ms.wfc.data.ui.DataGrid.setCurrentRow could not be converted

com.ms.wfc.data.ui.DataGrid.setDynamicColumns could not be converted

com.ms.wfc.data.ui.DataGrid.setEnterAction could not be converted

com.ms.wfc.data.ui.DataGrid.setFirstRow could not be converted

com.ms.wfc.data.ui.DataGrid.setHeaderLineCount could not be converted

com.ms.wfc.data.ui.DataGrid.setLeftColumn could not be converted

com.ms.wfc.data.ui.DataGrid.setRowHeight could not be converted

com.ms.wfc.data.ui.DataGrid.setScrollbars could not be converted

com.ms.wfc.data.ui.DataGrid.setSelectedColumns could not be converted

com.ms.wfc.data.ui.DataGrid.setSelectedRows could not be converted

com.ms.wfc.data.ui.DataGrid.setTabAction could not be converted

com.ms.wfc.data.ui.DataGrid.setWrapCellPointer could not be converted

com.ms.wfc.data.ui.DataNavigator.Customizer could not be converted

com.ms.wfc.data.ui.DataNavigator.getDataMember could not be converted

com.ms.wfc.data.ui.DataNavigator.moveFirst could not be converted

com.ms.wfc.data.ui.DataNavigator.moveLast could not be converted

com.ms.wfc.data.ui.DataNavigator.moveNext could not be converted

com.ms.wfc.data.ui.DataNavigator.movePrevious could not be converted

com.ms.wfc.data.ui.DataNavigator.propertyChanged could not be converted

com.ms.wfc.data.ui.DataNavigator.setDataMember could not be converted

com.ms.wfc.data.ui.DataSource.addDataSourceListener could not be converted

com.ms.wfc.data.ui.DataSource.addOnBeginTransComplete could not be converted

com.ms.wfc.data.ui.DataSource.addOnCommitTransComplete could not be converted

com.ms.wfc.data.ui.DataSource.addOnConnectComplete could not be converted

com.ms.wfc.data.ui.DataSource.addOnDisconnect could not be converted

com.ms.wfc.data.ui.DataSource.addOnEndOfRecordset could not be converted

com.ms.wfc.data.ui.DataSource.addOnExecuteComplete could not be converted

com.ms.wfc.data.ui.DataSource.addOnFetchComplete could not be converted

com.ms.wfc.data.ui.DataSource.addOnFetchProgress could not be converted

com.ms.wfc.data.ui.DataSource.addOnFieldChangeComplete could not be converted

com.ms.wfc.data.ui.DataSource.addOnInfoMessage could not be converted

com.ms.wfc.data.ui.DataSource.addOnMoveComplete could not be converted

com.ms.wfc.data.ui.DataSource.addOnRecordChangeComplete could not be converted

com.ms.wfc.data.ui.DataSource.addOnRecordsetChangeComplete could not be converted

com.ms.wfc.data.ui.DataSource.addOnRollbackTransComplete could not be converted

com.ms.wfc.data.ui.DataSource.addOnWillChangeField could not be converted

com.ms.wfc.data.ui.DataSource.addOnWillChangeRecord could not be converted

com.ms.wfc.data.ui.DataSource.addOnWillChangeRecordset could not be converted

com.ms.wfc.data.ui.DataSource.addOnWillConnect could not be converted

com.ms.wfc.data.ui.DataSource.addOnWillExecute could not be converted

com.ms.wfc.data.ui.DataSource.addOnWillMove could not be converted

com.ms.wfc.data.ui.DataSource.dataMemberAdded could not be converted

com.ms.wfc.data.ui.DataSource.dataMemberChanged could not be converted

com.ms.wfc.data.ui.DataSource.dataMemberRemoved could not be converted

com.ms.wfc.data.ui.DataSource.getAsyncConnect could not be converted

com.ms.wfc.data.ui.DataSource.getAsyncExecute could not be converted

com.ms.wfc.data.ui.DataSource.getAsyncFetch could not be converted

com.ms.wfc.data.ui.DataSource.getCacheSize could not be converted

com.ms.wfc.data.ui.DataSource.getCursorLocation could not be converted

com.ms.wfc.data.ui.DataSource.getCursorType could not be converted

com.ms.wfc.data.ui.DataSource.getDataMember could not be converted

com.ms.wfc.data.ui.DataSource.getDataMemberCount could not be converted

com.ms.wfc.data.ui.DataSource.getDataMemberName could not be converted

com.ms.wfc.data.ui.DataSource.getDesignTimeData could not be converted

com.ms.wfc.data.ui.DataSource.getLockType could not be converted

com.ms.wfc.data.ui.DataSource.getMaxRecords could not be converted

com.ms.wfc.data.ui.DataSource.getMode could not be converted

com.ms.wfc.data.ui.DataSource.getParentDataSource could not be converted

com.ms.wfc.data.ui.DataSource.getParentFieldName could not be converted

com.ms.wfc.data.ui.DataSource.getPassword could not be converted

com.ms.wfc.data.ui.DataSource.getPrepared could not be converted

com.ms.wfc.data.ui.DataSource.getRecordset could not be converted

com.ms.wfc.data.ui.DataSource.getSort could not be converted

com.ms.wfc.data.ui.DataSource.getStayInSync could not be converted

com.ms.wfc.data.ui.DataSource.getUserId could not be converted

com.ms.wfc.data.ui.DataSource.isChildDataSource could not be converted

com.ms.wfc.data.ui.DataSource.removeDataSourceListener could not be converted

com.ms.wfc.data.ui.DataSource.removeOnBeginTransComplete could not be converted

com.ms.wfc.data.ui.DataSource.removeOnCommitTransComplete could not be converted

com.ms.wfc.data.ui.DataSource.removeOnConnectComplete could not be converted

com.ms.wfc.data.ui.DataSource.removeOnDisconnect could not be converted

com.ms.wfc.data.ui.DataSource.removeOnEndOfRecordset could not be converted

com.ms.wfc.data.ui.DataSource.removeOnExecuteComplete could not be converted

com.ms.wfc.data.ui.DataSource.removeOnFetchComplete could not be converted

com.ms.wfc.data.ui.DataSource.removeOnFetchProgress could not be converted

com.ms.wfc.data.ui.DataSource.removeOnFieldChangeComplete could not be converted

com.ms.wfc.data.ui.DataSource.removeOnInfoMessage could not be converted

com.ms.wfc.data.ui.DataSource.removeOnMoveComplete could not be converted

com.ms.wfc.data.ui.DataSource.removeOnRecordChangeComplete could not be converted

com.ms.wfc.data.ui.DataSource.removeOnRecordsetChangeComplete could not be converted

com.ms.wfc.data.ui.DataSource.removeOnRollbackTransComplete could not be converted

com.ms.wfc.data.ui.DataSource.removeOnWillChangeField could not be converted

com.ms.wfc.data.ui.DataSource.removeOnWillChangeRecord could not be converted

com.ms.wfc.data.ui.DataSource.removeOnWillChangeRecordset could not be converted

com.ms.wfc.data.ui.DataSource.removeOnWillConnect could not be converted

com.ms.wfc.data.ui.DataSource.removeOnWillExecute could not be converted

com.ms.wfc.data.ui.DataSource.removeOnWillMove could not be converted

com.ms.wfc.data.ui.DataSource.setAsyncConnect could not be converted

com.ms.wfc.data.ui.DataSource.setAsyncExecute could not be converted

com.ms.wfc.data.ui.DataSource.setAsyncFetch could not be converted

com.ms.wfc.data.ui.DataSource.setCacheSize could not be converted

com.ms.wfc.data.ui.DataSource.setConnectionTimeout could not be converted

com.ms.wfc.data.ui.DataSource.setCursorLocation could not be converted

com.ms.wfc.data.ui.DataSource.setCursorType could not be converted

com.ms.wfc.data.ui.DataSource.setDesignTimeData could not be converted

com.ms.wfc.data.ui.DataSource.setFilter could not be converted

com.ms.wfc.data.ui.DataSource.setIsolationLevel could not be converted

com.ms.wfc.data.ui.DataSource.setLockType could not be converted

com.ms.wfc.data.ui.DataSource.setMaxRecords could not be converted

com.ms.wfc.data.ui.DataSource.setMode could not be converted

com.ms.wfc.data.ui.DataSource.setParentDataSource could not be converted

com.ms.wfc.data.ui.DataSource.setParentFieldName could not be converted

com.ms.wfc.data.ui.DataSource.setPassword could not be converted

com.ms.wfc.data.ui.DataSource.setPrepared could not be converted

com.ms.wfc.data.ui.DataSource.setSort could not be converted

com.ms.wfc.data.ui.DataSource.setStayInSync could not be converted

com.ms.wfc.data.ui.DataSource.setUserId could not be converted

com.ms.wfc.data.ui.DataSource.unrealize could not be converted

com.ms.wfc.data.ui.EnterAction could not be converted

com.ms.wfc.data.ui.ErrorEvent could not be converted

com.ms.wfc.data.ui.ErrorEventHandler could not be converted

com.ms.wfc.data.ui.GridLineStyle.GridLineStyle could not be converted

com.ms.wfc.data.ui.GridLineStyle.RAISED3D could not be converted

com.ms.wfc.data.ui.GridLineStyle.SUNKEN3D could not be converted

com.ms.wfc.data.ui.PositionChangeEvent.lastColumn could not be converted

com.ms.wfc.data.ui.PositionChangeEvent.lastRow could not be converted

com.ms.wfc.data.ui.PositionChangeEvent.PositionChangeEvent could not be converted

com.ms.wfc.data.ui.RecordsetProvider could not be converted

com.ms.wfc.data.ui.TabAction could not be converted

Com.ms.wfc.io Error Messages

com.ms.wfc.io.BufferedStream.getStream could not be converted

com.ms.wfc.io.BufferedStream.readCore could not be converted

com.ms.wfc.io.BufferedStream.readStringCharsAnsi could not be converted

com.ms.wfc.io.BufferedStream.readStringNull could not be converted

com.ms.wfc.io.BufferedStream.readStringNullAnsi could not be converted

com.ms.wfc.io.BufferedStream.writeCore could not be converted

com.ms.wfc.io.CodePage.ANSI could not be converted

com.ms.wfc.io.CodePage.CodePage could not be converted

com.ms.wfc.io.CodePage.MAC could not be converted

com.ms.wfc.io.CodePage.OEM could not be converted

com.ms.wfc.io.DataStream.comStream could not be converted

com.ms.wfc.io.DataStream.DataStream could not be converted

com.ms.wfc.io.DataStream.fromComStream could not be converted

com.ms.wfc.io.DataStream.getComStream could not be converted

com.ms.wfc.io.DataStream.readCore could not be converted

com.ms.wfc.io.DataStream.readStringNull could not be converted

com.ms.wfc.io.DataStream.readStringNullAnsi could not be converted

com.ms.wfc.io.DataStream.readUTF could not be converted

com.ms.wfc.io.DataStream.toComStream could not be converted

com.ms.wfc.io.DataStream.writeCore could not be converted

com.ms.wfc.io.DataStream.writeStringChars could not be converted

com.ms.wfc.io.DataStream.writeStringCharsAnsi could not be converted

com.ms.wfc.io.DataStream.writeStringNull could not be converted

com.ms.wfc.io.DataStream.writeStringNullAnsi could not be converted

com.ms.wfc.io.DataStream.writeUTF could not be converted

com.ms.wfc.io.DataStreamFromComStream could not be converted

com.ms.wfc.io.File.openStandardError could not be converted

com.ms.wfc.io.File.openStandardInput could not be converted

com.ms.wfc.io.File.openStandardOutput could not be converted

com.ms.wfc.io.File.readCore could not be converted

com.ms.wfc.io.File.writeCore could not be converted

com.ms.wfc.io.FileEnumerator.close could not be converted

com.ms.wfc.io.FileEnumerator.FileEnumerator could not be converted

com.ms.wfc.io.FileEnumerator.FileEnumerator could not be converted

com.ms.wfc.io.FileEnumerator.finalize could not be converted

com.ms.wfc.io.IDataStream.getComStream could not be converted

com.ms.wfc.io.IDataStream.readStringNull could not be converted

com.ms.wfc.io.IDataStream.readStringNullAnsi could not be converted

com.ms.wfc.io.IDataStream.readUTF could not be converted

com.ms.wfc.io.IDataStream.writeStringChars could not be converted

com.ms.wfc.io.IDataStream.writeStringCharsAnsi could not be converted

com.ms.wfc.io.IDataStream.writeStringNull could not be converted

com.ms.wfc.io.IDataStream.writeStringNullAnsi could not be converted

com.ms.wfc.io.IDataStream.writeUTF could not be converted

com.ms.wfc.io.IDataStreamProvider could not be converted

com.ms.wfc.io.MemoryStream.MemoryStream could not be converted

com.ms.wfc.io.MemoryStream.readCore could not be converted

com.ms.wfc.io.MemoryStream.readStringCharsAnsi could not be converted

com.ms.wfc.io.MemoryStream.readStringNull could not be converted

com.ms.wfc.io.MemoryStream.readStringNullAnsi could not be converted

com.ms.wfc.io.MemoryStream.writeCore could not be converted

com.ms.wfc.io.MemoryStream.writeTo could not be converted

com.ms.wfc.io.Reader.Reader could not be converted

com.ms.wfc.io.TextReader.TextReader(ByteStream, int, int) could not be converted

com.ms.wfc.io.TextReader.TextReader(ByteStream, int) could not be converted

com.ms.wfc.io.TextReader.TextReader(ByteStream) could not be converted

com.ms.wfc.io.WinIOException.getErrorCode could not be converted

com.ms.wfc.io.WinIOException.WinIOException could not be converted

com.ms.wfc.io.Writer.Writer could not be converted

Com.ms.wfc.ole32 Error Messages

com.ms.wfc.ole32.DBBINDING.pTypeInfo could not be converted

com.ms.wfc.ole32.DBCOLUMNINFO.pTypeInfo could not be converted

com.ms.wfc.ole32.IEnumFORMATETC.iid could not be converted

com.ms.wfc.ole32.IEnumSTATDATA.iid could not be converted

com.ms.wfc.ole32.ILockBytes.iid could not be converted

com.ms.wfc.ole32.IMalloc.iid could not be converted

com.ms.wfc.ole32.IOleDataFormat.iid could not be converted

com.ms.wfc.ole32.IOleDataObject.iid could not be converted

com.ms.wfc.ole32.IOleDataObjectWithIStream.iid could not be converted

com.ms.wfc.ole32.IOleDropSource.iid could not be converted

com.ms.wfc.ole32.IOleDropTarget.iid could not be converted

com.ms.wfc.ole32.IPersist.iid could not be converted

com.ms.wfc.ole32.IPersistStream.iid could not be converted

com.ms.wfc.ole32.IPicture.iid could not be converted

com.ms.wfc.ole32.IStorage.iid could not be converted

Com.ms.wfc.ui Error Messages

com.ms.wfc.ui.AnchorEditor.editValue could not be converted

com.ms.wfc.ui.AnchorEditor.getConstantName could not be converted

com.ms.wfc.ui.AnchorEditor.getTextFromValue could not be converted

com.ms.wfc.ui.AnimationFileNameEditor could not be converted

com.ms.wfc.ui.AutoSizeEvent could not be converted

com.ms.wfc.ui.AutoSizeEventHandler could not be converted

com.ms.wfc.ui.AxHost.AboutBoxDelegate could not be converted

com.ms.wfc.ui.AxHost.AxPropertyInfo could not be converted

com.ms.wfc.ui.AxHost.DispidAttribute could not be converted

com.ms.wfc.ui.AxHost.fireOnClick could not be converted

com.ms.wfc.ui.AxHost.fireOnDblClick could not be converted

com.ms.wfc.ui.AxHost.fireOnKeyDown could not be converted

com.ms.wfc.ui.AxHost.fireOnKeyPress could not be converted

com.ms.wfc.ui.AxHost.fireOnKeyUp could not be converted

com.ms.wfc.ui.AxHost.fireOnMouseDown could not be converted

com.ms.wfc.ui.AxHost.fireOnMouseMove could not be converted

com.ms.wfc.ui.AxHost.fireOnMouseUp could not be converted

com.ms.wfc.ui.AxHost.Flags could not be converted

com.ms.wfc.ui.AxHost.getAmbientProperty could not be converted

com.ms.wfc.ui.AxHost.getClientAttributes could not be converted

com.ms.wfc.ui.AxHost.getColorFromOleColor could not be converted

com.ms.wfc.ui.AxHost.getFontFromIFont could not be converted

com.ms.wfc.ui.AxHost.getFontFromIFontDisp could not be converted

com.ms.wfc.ui.AxHost.getIFontDispFromFont could not be converted

com.ms.wfc.ui.AxHost.getIFontFromFont could not be converted

com.ms.wfc.ui.AxHost.getIPictureDispFromPicture could not be converted

com.ms.wfc.ui.AxHost.getIPictureFromPicture could not be converted

com.ms.wfc.ui.AxHost.getOleColorFromColor could not be converted

com.ms.wfc.ui.AxHost.getPictureFromIPicture could not be converted

com.ms.wfc.ui.AxHost.getPictureFromIPictureDisp could not be converted

com.ms.wfc.ui.AxHost.propertyChanged could not be converted

com.ms.wfc.ui.AxHost.setAboutBoxDelegate could not be converted

com.ms.wfc.ui.AxHost.setTopLevel could not be converted

com.ms.wfc.ui.AxHost.shouldPersistContainingForm could not be converted

com.ms.wfc.ui.AxHost.State.save could not be converted

com.ms.wfc.ui.AxHost.State.State could not be converted

com.ms.wfc.ui.AxPersist could not be converted

com.ms.wfc.ui.Bitmap.Bitmap could not be converted

com.ms.wfc.ui.Bitmap.Bitmap(Bitmap, Bitmap) could not be converted

com.ms.wfc.ui.Bitmap.Bitmap(int) could not be converted

com.ms.wfc.ui.Bitmap.Bitmap(int, int, int, int, short[]) could not be converted

com.ms.wfc.ui.Bitmap.Bitmap(IPicture) could not be converted

com.ms.wfc.ui.Bitmap.Bitmap(Palette) could not be converted

com.ms.wfc.ui.Bitmap.copyHandle could not be converted

com.ms.wfc.ui.Bitmap.destroyHandle could not be converted

com.ms.wfc.ui.Bitmap.drawStretchTo could not be converted

com.ms.wfc.ui.Bitmap.drawTo could not be converted

com.ms.wfc.ui.Bitmap.getColorMask could not be converted

com.ms.wfc.ui.Bitmap.getGraphics could not be converted

com.ms.wfc.ui.Bitmap.getHandle could not be converted

com.ms.wfc.ui.Bitmap.getMonochromeMask could not be converted

com.ms.wfc.ui.Bitmap.getPICTDESC could not be converted

com.ms.wfc.ui.Bitmap.getTransparent could not be converted

com.ms.wfc.ui.Bitmap.getTransparentColor could not be converted

com.ms.wfc.ui.Bitmap.initialize could not be converted

com.ms.wfc.ui.Brush.Brush(Bitmap) could not be converted

com.ms.wfc.ui.Brush.Brush(int) could not be converted

com.ms.wfc.ui.Brush.copyHandle could not be converted

com.ms.wfc.ui.Brush.destroyHandle could not be converted

com.ms.wfc.ui.Brush.getHandle could not be converted

com.ms.wfc.ui.Brush.HALFTONE could not be converted

com.ms.wfc.ui.BrushStyle.HOLLOW could not be converted

com.ms.wfc.ui.BrushStyle.PATTERN could not be converted

com.ms.wfc.ui.Button.propertyChanged could not be converted

com.ms.wfc.ui.CheckBox.getGroupValue could not be converted

com.ms.wfc.ui.CheckBox.setGroupValue could not be converted

com.ms.wfc.ui.CheckBox.setTextAlign could not be converted

com.ms.wfc.ui.CheckBox.setTextAlign could not be converted

com.ms.wfc.ui.CheckedListBox.propertyChanged could not be converted

com.ms.wfc.ui.Color.fromCMYK could not be converted

com.ms.wfc.ui.Color.getConstructorArgs could not be converted

com.ms.wfc.ui.Color.toString could not be converted

com.ms.wfc.ui.ComboBox.propertyChanged could not be converted

com.ms.wfc.ui.ContentAlignment.toTextFormat could not be converted

com.ms.wfc.ui.Control.getVisible could not be converted

com.ms.wfc.ui.Control.invokeAsync could not be converted

com.ms.wfc.ui.Control.PROP_BACKCOLOR could not be converted

com.ms.wfc.ui.Control.PROP_CURSOR could not be converted

com.ms.wfc.ui.Control.PROP_ENABLED could not be converted

com.ms.wfc.ui.Control.PROP_FONT could not be converted

com.ms.wfc.ui.Control.PROP_FORECOLOR could not be converted

com.ms.wfc.ui.Control.PROP_LOCATION could not be converted

com.ms.wfc.ui.Control.PROP_PARENTBACKCOLOR could not be converted

com.ms.wfc.ui.Control.PROP_PARENTFONT could not be converted

com.ms.wfc.ui.Control.PROP_PARENTFORECOLOR could not be converted

com.ms.wfc.ui.Control.PROP_SIZE could not be converted

com.ms.wfc.ui.Control.PROP_TEXT could not be converted

com.ms.wfc.ui.Control.PROP_VISIBLE could not be converted

com.ms.wfc.ui.Control.propertyChanged could not be converted

com.ms.wfc.ui.Control.sendMessage could not be converted

com.ms.wfc.ui.Control.STYLE_ACCEPTSCHILDREN could not be converted

com.ms.wfc.ui.CoordinateSystem could not be converted

com.ms.wfc.ui.Cursor.Cursor could not be converted

com.ms.wfc.ui.Cursor.destroyHandle could not be converted

com.ms.wfc.ui.Cursor.getPICTDESC could not be converted

com.ms.wfc.ui.Cursor.initialize could not be converted

com.ms.wfc.ui.DateTimeFormatEvent could not be converted

com.ms.wfc.ui.DateTimeFormatEventHandler could not be converted

com.ms.wfc.ui.DateTimeFormatQueryEvent could not be converted

com.ms.wfc.ui.DateTimeFormatQueryEventHandler could not be converted

com.ms.wfc.ui.DateTimePicker.addOnFormat could not be converted

com.ms.wfc.ui.DateTimePicker.addOnFormatQuery could not be converted

com.ms.wfc.ui.DateTimePicker.addOnUserString could not be converted

com.ms.wfc.ui.DateTimePicker.getAllowUserString could not be converted

com.ms.wfc.ui.DateTimePicker.onFormat could not be converted

com.ms.wfc.ui.DateTimePicker.onFormatQuery could not be converted

com.ms.wfc.ui.DateTimePicker.onUserString could not be converted

com.ms.wfc.ui.DateTimePicker.removeOnFormat could not be converted

com.ms.wfc.ui.DateTimePicker.removeOnFormatQuery could not be converted

com.ms.wfc.ui.DateTimePicker.removeOnUserString could not be converted

com.ms.wfc.ui.DateTimePicker.setAllowUserString could not be converted

com.ms.wfc.ui.DateTimeUserStringEvent could not be converted

com.ms.wfc.ui.DateTimeUserStringEventHandler could not be converted

com.ms.wfc.ui.DateTimeWmKeyDownEvent.DateTimeWmKeyDownEvent could not be converted

com.ms.wfc.ui.DateTimeWmKeyDownEvent.format could not be converted

com.ms.wfc.ui.DateTimeWmKeyDownEvent.time could not be converted

com.ms.wfc.ui.Dimensions.getConstructorArgs could not be converted

com.ms.wfc.ui.Dimensions.save could not be converted

com.ms.wfc.ui.DockEditor.editValue could not be converted

com.ms.wfc.ui.DockEditor.getConstantName could not be converted

com.ms.wfc.ui.DockEditor.getTextFromValue could not be converted

com.ms.wfc.ui.DocumentReadyEvent could not be converted

com.ms.wfc.ui.DocumentReadyEventHandler could not be converted

com.ms.wfc.ui.Edit.propertyChanged could not be converted

com.ms.wfc.ui.Edit.undo could not be converted

com.ms.wfc.ui.EraseBackgroundEvent could not be converted

com.ms.wfc.ui.FileDialog.promptFileCreate could not be converted

com.ms.wfc.ui.FileDialog.promptFileNotFound could not be converted

com.ms.wfc.ui.FileDialog.promptFileOverwrite could not be converted

com.ms.wfc.ui.FloodFillType could not be converted

com.ms.wfc.ui.Font.ANSI_VAR could not be converted

com.ms.wfc.ui.Font.destroyHandle could not be converted

com.ms.wfc.ui.Font.DEVICE_DEFAULT could not be converted

com.ms.wfc.ui.Font.equalsBase could not be converted

com.ms.wfc.ui.Font.Font could not be converted

com.ms.wfc.ui.Font.getCharacterSet could not be converted

com.ms.wfc.ui.Font.getConstructorArgs could not be converted

com.ms.wfc.ui.Font.getFamily could not be converted

com.ms.wfc.ui.Font.getFontMetrics could not be converted

com.ms.wfc.ui.Font.getOrientation could not be converted

com.ms.wfc.ui.Font.getPitch could not be converted

com.ms.wfc.ui.Font.getStock could not be converted

com.ms.wfc.ui.Font.OEM_FIXED could not be converted

com.ms.wfc.ui.Font.save could not be converted

com.ms.wfc.ui.Font.SYSTEM could not be converted

com.ms.wfc.ui.Font.SYSTEM_FIXED could not be converted

com.ms.wfc.ui.FontDescriptor could not be converted

com.ms.wfc.ui.FontDevice could not be converted

com.ms.wfc.ui.FontDialog.getFontDevice could not be converted

com.ms.wfc.ui.FontDialog.getPrinterDC could not be converted

com.ms.wfc.ui.FontDialog.getScalableOnly could not be converted

com.ms.wfc.ui.FontDialog.getTrueTypeOnly could not be converted

com.ms.wfc.ui.FontDialog.getWysiwyg could not be converted

com.ms.wfc.ui.FontDialog.hookProc could not be converted

com.ms.wfc.ui.FontDialog.setFontDevice could not be converted

com.ms.wfc.ui.FontDialog.setPrinterDC could not be converted

com.ms.wfc.ui.FontDialog.setScalableOnly could not be converted

com.ms.wfc.ui.FontDialog.setTrueTypeOnly could not be converted

com.ms.wfc.ui.FontDialog.setWysiwyg could not be converted

com.ms.wfc.ui.FontFamily could not be converted

com.ms.wfc.ui.FontMetrics could not be converted

com.ms.wfc.ui.FontPitch could not be converted

com.ms.wfc.ui.FontSize.CELLHEIGHT could not be converted

com.ms.wfc.ui.FontSize.CENTIMETERS could not be converted

com.ms.wfc.ui.FontSize.CHARACTERHEIGHT could not be converted

com.ms.wfc.ui.FontSize.EM could not be converted

com.ms.wfc.ui.FontSize.EX could not be converted

com.ms.wfc.ui.FontType could not be converted

com.ms.wfc.ui.FontWeight.EXTRALIGHT could not be converted

com.ms.wfc.ui.FontWeight.LIGHT could not be converted

com.ms.wfc.ui.FontWeight.THIN could not be converted

com.ms.wfc.ui.Form.FORMSTATE_AUTOSCALING could not be converted

com.ms.wfc.ui.Form.FORMSTATE_AUTOSCROLLING could not be converted

com.ms.wfc.ui.Form.FORMSTATE_BORDERSTYLE could not be converted

com.ms.wfc.ui.Form.FORMSTATE_CONTROLBOX could not be converted

com.ms.wfc.ui.Form.FORMSTATE_HELPBUTTON could not be converted

com.ms.wfc.ui.Form.FORMSTATE_HSCROLLVISIBLE could not be converted

com.ms.wfc.ui.Form.FORMSTATE_KEYPREVIEW could not be converted

com.ms.wfc.ui.Form.FORMSTATE_MAXIMIZEBOX could not be converted

com.ms.wfc.ui.Form.FORMSTATE_MINIMIZEBOX could not be converted

com.ms.wfc.ui.Form.FORMSTATE_PALETTEMODE could not be converted

com.ms.wfc.ui.Form.FORMSTATE_SETCLIENTSIZE could not be converted

com.ms.wfc.ui.Form.FORMSTATE_SHOWWINDOWONCREATE could not be converted

com.ms.wfc.ui.Form.FORMSTATE_STARTPOS could not be converted

com.ms.wfc.ui.Form.FORMSTATE_TASKBAR could not be converted

com.ms.wfc.ui.Form.FORMSTATE_TOPMOST could not be converted

com.ms.wfc.ui.Form.FORMSTATE_USERHASSCROLLED could not be converted

com.ms.wfc.ui.Form.FORMSTATE_VSCROLLVISIBLE could not be converted

com.ms.wfc.ui.Form.FORMSTATE_WINDOWSTATE could not be converted

com.ms.wfc.ui.Form.getFormState could not be converted

com.ms.wfc.ui.Form.getPalette could not be converted

com.ms.wfc.ui.Form.getPaletteMode could not be converted

com.ms.wfc.ui.Form.getPaletteSource could not be converted

com.ms.wfc.ui.Form.hasFormState could not be converted

com.ms.wfc.ui.Form.notifyPaletteChange could not be converted

com.ms.wfc.ui.Form.onMDIChildActivate could not be converted

com.ms.wfc.ui.Form.onNewPalette could not be converted

com.ms.wfc.ui.Form.propertyChanged could not be converted

com.ms.wfc.ui.Form.setBorderStyle could not be converted

com.ms.wfc.ui.Form.setFormState could not be converted

com.ms.wfc.ui.Form.setPaletteMode could not be converted

com.ms.wfc.ui.Form.setPaletteSource could not be converted

com.ms.wfc.ui.FormPaletteMode could not be converted

com.ms.wfc.ui.Graphics.drawArc could not be converted

com.ms.wfc.ui.Graphics.drawChord could not be converted

com.ms.wfc.ui.Graphics.drawPie could not be converted

com.ms.wfc.ui.Graphics.floodFill could not be converted

com.ms.wfc.ui.Graphics.getCoordinateSystem could not be converted

com.ms.wfc.ui.Graphics.getDeviceOrigin could not be converted

com.ms.wfc.ui.Graphics.getDevicePoint could not be converted

com.ms.wfc.ui.Graphics.getDeviceScale could not be converted

com.ms.wfc.ui.Graphics.getFontDescriptors could not be converted

com.ms.wfc.ui.Graphics.getHandle could not be converted

com.ms.wfc.ui.Graphics.getLogicalPoint could not be converted

com.ms.wfc.ui.Graphics.getLogicalSizeX could not be converted

com.ms.wfc.ui.Graphics.getLogicalSizeY could not be converted

com.ms.wfc.ui.Graphics.getOpaque could not be converted

com.ms.wfc.ui.Graphics.getPageOrigin could not be converted

com.ms.wfc.ui.Graphics.getPageScale could not be converted

com.ms.wfc.ui.Graphics.getPhysicalSizeX could not be converted

com.ms.wfc.ui.Graphics.getPhysicalSizeY could not be converted

com.ms.wfc.ui.Graphics.getPixel could not be converted

com.ms.wfc.ui.Graphics.getTextSize could not be converted

com.ms.wfc.ui.Graphics.getTextSpace could not be converted

com.ms.wfc.ui.Graphics.Graphics could not be converted

com.ms.wfc.ui.Graphics.invert could not be converted

com.ms.wfc.ui.Graphics.renderPalette could not be converted

com.ms.wfc.ui.Graphics.scroll could not be converted

com.ms.wfc.ui.Graphics.setCoordinateOrigin could not be converted

com.ms.wfc.ui.Graphics.setCoordinateScale could not be converted

com.ms.wfc.ui.Graphics.setCoordinateSystem could not be converted

com.ms.wfc.ui.Graphics.setHandle could not be converted

com.ms.wfc.ui.Graphics.setOpaque could not be converted

com.ms.wfc.ui.Graphics.setPixel could not be converted

com.ms.wfc.ui.Graphics.setTextSpace could not be converted

com.ms.wfc.ui.Help.Help could not be converted

com.ms.wfc.ui.HelpEvent.component could not be converted

com.ms.wfc.ui.HelpEvent.contextId could not be converted

com.ms.wfc.ui.HelpEvent.contextType could not be converted

com.ms.wfc.ui.HelpEvent.controlId could not be converted

com.ms.wfc.ui.HelpFileFileNameEditor could not be converted

com.ms.wfc.ui.HelpProvider.shouldPersistShowHelp could not be converted

com.ms.wfc.ui.IActiveXCustomPropertyDialog could not be converted

com.ms.wfc.ui.Icon.getPICTDESC could not be converted

com.ms.wfc.ui.Icon.Icon could not be converted

com.ms.wfc.ui.Icon.initialize could not be converted

com.ms.wfc.ui.Icon.loadPicture could not be converted

com.ms.wfc.ui.IHandleHook could not be converted

com.ms.wfc.ui.Image.copyHandle could not be converted

com.ms.wfc.ui.Image.dirty could not be converted

com.ms.wfc.ui.Image.drawStretchTo could not be converted

com.ms.wfc.ui.Image.drawTo could not be converted

com.ms.wfc.ui.Image.getExtension could not be converted

com.ms.wfc.ui.Image.getHandle could not be converted

com.ms.wfc.ui.Image.getPICTDESC could not be converted

com.ms.wfc.ui.Image.getPicture could not be converted

com.ms.wfc.ui.Image.Image could not be converted

com.ms.wfc.ui.Image.initialize could not be converted

com.ms.wfc.ui.Image.loadImage could not be converted

com.ms.wfc.ui.Image.loadPicture could not be converted

com.ms.wfc.ui.Image.PictureList could not be converted

com.ms.wfc.ui.ImageIndexEditor could not be converted

com.ms.wfc.ui.ImageList.createHandle could not be converted

com.ms.wfc.ui.ImageList.destroyHandle could not be converted

com.ms.wfc.ui.ImageList.getBackColor could not be converted

com.ms.wfc.ui.ImageList.getIcon could not be converted

com.ms.wfc.ui.ImageList.getMaskColor could not be converted

com.ms.wfc.ui.ImageList.getUseMask could not be converted

com.ms.wfc.ui.ImageList.ImageList could not be converted

com.ms.wfc.ui.ImageList.recreateHandle could not be converted

com.ms.wfc.ui.ImageList.SetBackColor could not be converted

com.ms.wfc.ui.ImageList.setImages could not be converted

com.ms.wfc.ui.ImageList.setMaskColor could not be converted

com.ms.wfc.ui.ImageList.setUseMask could not be converted

com.ms.wfc.ui.ImageListStreamer.getExtension could not be converted

com.ms.wfc.ui.ImageListStreamer.ImageListStreamer could not be converted

com.ms.wfc.ui.ImageListStreamer.save could not be converted

com.ms.wfc.ui.ISelectionService.setSelectionStyle could not be converted

com.ms.wfc.ui.Label.propertyChanged could not be converted

com.ms.wfc.ui.Label.setTextAlign could not be converted

com.ms.wfc.ui.ListItem.getConstructorArgs could not be converted

com.ms.wfc.ui.ListItem.ListItem could not be converted

com.ms.wfc.ui.ListItem.save could not be converted

com.ms.wfc.ui.MDIWindowDialog could not be converted

com.ms.wfc.ui.Menu.Menu could not be converted

com.ms.wfc.ui.Menu.Menu could not be converted

com.ms.wfc.ui.MenuItem.setChecked could not be converted

com.ms.wfc.ui.Metafile.copyHandle could not be converted

com.ms.wfc.ui.Metafile.destroyHandle could not be converted

com.ms.wfc.ui.Metafile.drawStretchTo could not be converted

com.ms.wfc.ui.Metafile.drawTo could not be converted

com.ms.wfc.ui.Metafile.getHandle could not be converted

com.ms.wfc.ui.Metafile.getPICTDESC could not be converted

com.ms.wfc.ui.Metafile.getRenderedSize could not be converted

com.ms.wfc.ui.Metafile.initialize could not be converted

com.ms.wfc.ui.Metafile.Metafile could not be converted

com.ms.wfc.ui.MonthCalendar.propertyChanged could not be converted

com.ms.wfc.ui.MouseEvent.MouseEvent could not be converted

com.ms.wfc.ui.PaintEvent.graphics could not be converted

com.ms.wfc.ui.Palette could not be converted

com.ms.wfc.ui.Palette.clone could not be converted

com.ms.wfc.ui.Palette.copyHandle could not be converted

com.ms.wfc.ui.Palette.destroyHandle could not be converted

com.ms.wfc.ui.Palette.dispose could not be converted

com.ms.wfc.ui.Palette.equals could not be converted

com.ms.wfc.ui.Palette.finalize could not be converted

com.ms.wfc.ui.Palette.getHalftonePalette could not be converted

com.ms.wfc.ui.Palette.getHandle could not be converted

com.ms.wfc.ui.Palette.getPaletteSupported could not be converted

com.ms.wfc.ui.Palette.Palette could not be converted

com.ms.wfc.ui.Pen.copyHandle could not be converted

com.ms.wfc.ui.Pen.destroyHandle could not be converted

com.ms.wfc.ui.Pen.getHandle could not be converted

com.ms.wfc.ui.Pen.Pen could not be converted

com.ms.wfc.ui.PenEntry could not be converted

com.ms.wfc.ui.PenStyle.INSIDEFRAME could not be converted

com.ms.wfc.ui.PenStyle.NULL could not be converted

com.ms.wfc.ui.PictureBox.setImage could not be converted

com.ms.wfc.ui.PictureBox.setImage could not be converted

com.ms.wfc.ui.Point.getConstructorArgs could not be converted

com.ms.wfc.ui.Point.Point could not be converted

com.ms.wfc.ui.Point.save could not be converted

com.ms.wfc.ui.PopulatedMenusEditor could not be converted

com.ms.wfc.ui.ProgressBar.addOnValueChanged could not be converted

com.ms.wfc.ui.ProgressBar.removeOnValueChanged could not be converted

com.ms.wfc.ui.RadioButton.setTextAlign could not be converted

com.ms.wfc.ui.Radix could not be converted

com.ms.wfc.ui.RasterOp could not be converted

com.ms.wfc.ui.ReadyStateEvent could not be converted

com.ms.wfc.ui.ReadyStateEventHandler could not be converted

com.ms.wfc.ui.Rebar.addBand could not be converted

com.ms.wfc.ui.Rebar.addOnAutoSize could not be converted

com.ms.wfc.ui.Rebar.addOnHeightChange could not be converted

com.ms.wfc.ui.Rebar.addOnLayoutChange could not be converted

com.ms.wfc.ui.Rebar.applyAutoSize could not be converted

com.ms.wfc.ui.Rebar.getBandBorders could not be converted

com.ms.wfc.ui.Rebar.getBands could not be converted

com.ms.wfc.ui.Rebar.getDoubleClickToggle could not be converted

com.ms.wfc.ui.Rebar.getFixedOrder could not be converted

com.ms.wfc.ui.Rebar.getOrientation could not be converted

com.ms.wfc.ui.Rebar.onAutoSize could not be converted

com.ms.wfc.ui.Rebar.onHeightChange could not be converted

com.ms.wfc.ui.Rebar.onLayoutChange could not be converted

com.ms.wfc.ui.Rebar.removeAllBands could not be converted

com.ms.wfc.ui.Rebar.removeBand could not be converted

com.ms.wfc.ui.Rebar.removeOnAutoSize could not be converted

com.ms.wfc.ui.Rebar.removeOnHeightChange could not be converted

com.ms.wfc.ui.Rebar.removeOnLayoutChange could not be converted

com.ms.wfc.ui.Rebar.setBandBorders could not be converted

com.ms.wfc.ui.Rebar.setBands could not be converted

com.ms.wfc.ui.Rebar.setComponentSite could not be converted

com.ms.wfc.ui.Rebar.setDoubleClickToggle could not be converted

com.ms.wfc.ui.Rebar.setFixedOrder could not be converted

com.ms.wfc.ui.Rebar.setNewControls could not be converted

com.ms.wfc.ui.Rebar.setOrientation could not be converted

com.ms.wfc.ui.RebarBand.getAllowVariableHeight could not be converted

com.ms.wfc.ui.RebarBand.getAllowVertical could not be converted

com.ms.wfc.ui.RebarBand.getBandBreak could not be converted

com.ms.wfc.ui.RebarBand.getChildControl could not be converted

com.ms.wfc.ui.RebarBand.getChildEdge could not be converted

com.ms.wfc.ui.RebarBand.getFixedBitmap could not be converted

com.ms.wfc.ui.RebarBand.getGrowBy could not be converted

com.ms.wfc.ui.RebarBand.getHeaderWidth could not be converted

com.ms.wfc.ui.RebarBand.getIdealWidth could not be converted

com.ms.wfc.ui.RebarBand.getImageIndex could not be converted

com.ms.wfc.ui.RebarBand.getIndex could not be converted

com.ms.wfc.ui.RebarBand.getMaxInitialHeight could not be converted

com.ms.wfc.ui.RebarBand.getMinChildHeight could not be converted

com.ms.wfc.ui.RebarBand.getMinChildWidth could not be converted

com.ms.wfc.ui.RebarBand.getVisibleGripper could not be converted

com.ms.wfc.ui.RebarBand.maximize could not be converted

com.ms.wfc.ui.RebarBand.minimize could not be converted

com.ms.wfc.ui.RebarBand.setAllowVariableHeight could not be converted

com.ms.wfc.ui.RebarBand.setAllowVertical could not be converted

com.ms.wfc.ui.RebarBand.setBandBreak could not be converted

com.ms.wfc.ui.RebarBand.setChildControl could not be converted

com.ms.wfc.ui.RebarBand.setChildEdge could not be converted

com.ms.wfc.ui.RebarBand.setFixedBitmap could not be converted

com.ms.wfc.ui.RebarBand.setGrowBy could not be converted

com.ms.wfc.ui.RebarBand.setHeaderWidth could not be converted

com.ms.wfc.ui.RebarBand.setIdealWidth could not be converted

com.ms.wfc.ui.RebarBand.setImageIndex could not be converted

com.ms.wfc.ui.RebarBand.setIndex could not be converted

com.ms.wfc.ui.RebarBand.setMaxInitialHeight could not be converted

com.ms.wfc.ui.RebarBand.setMinChildHeight could not be converted

com.ms.wfc.ui.RebarBand.setMinChildWidth could not be converted

com.ms.wfc.ui.RebarBand.setVisibleGripper could not be converted

com.ms.wfc.ui.RebarBand.updateStyle could not be converted

com.ms.wfc.ui.Rectangle.Editor.Editor could not be converted

com.ms.wfc.ui.Rectangle.getConstructorArgs could not be converted

com.ms.wfc.ui.Rectangle.Rectangle could not be converted

com.ms.wfc.ui.Rectangle.save could not be converted

com.ms.wfc.ui.Rectangle.setBounds could not be converted

com.ms.wfc.ui.Rectangle.toRECT could not be converted

com.ms.wfc.ui.Region.copyHandle could not be converted

com.ms.wfc.ui.Region.createPolygonal could not be converted

com.ms.wfc.ui.Region.destroyHandle could not be converted

com.ms.wfc.ui.RichEdit.DLL_RICHEDIT could not be converted

com.ms.wfc.ui.RichEdit.getDelimiter could not be converted

com.ms.wfc.ui.RichEdit.getFollowPunctuation could not be converted

com.ms.wfc.ui.RichEdit.getIMEColor could not be converted

com.ms.wfc.ui.RichEdit.getIMEOptions could not be converted

com.ms.wfc.ui.RichEdit.getLeadPunctuation could not be converted

com.ms.wfc.ui.RichEdit.getWordBreak could not be converted

com.ms.wfc.ui.RichEdit.getWordPunctuation could not be converted

com.ms.wfc.ui.RichEdit.moveInsertionPoint could not be converted

com.ms.wfc.ui.RichEdit.RICHEDIT_CLASS10A could not be converted

com.ms.wfc.ui.RichEdit.RICHEDIT_CLASSA could not be converted

com.ms.wfc.ui.RichEdit.RICHEDIT_CLASSW could not be converted

com.ms.wfc.ui.RichEdit.RICHEDIT_DLL10 could not be converted

com.ms.wfc.ui.RichEdit.RICHEDIT_DLL20 could not be converted

com.ms.wfc.ui.RichEdit.setFollowPunctuation could not be converted

com.ms.wfc.ui.RichEdit.setIMEColor could not be converted

com.ms.wfc.ui.RichEdit.setIMEOptions could not be converted

com.ms.wfc.ui.RichEdit.setLeadPunctuation could not be converted

com.ms.wfc.ui.RichEdit.setWordBreak could not be converted

com.ms.wfc.ui.RichEdit.setWordPunctuation could not be converted

com.ms.wfc.ui.RichEdit.span could not be converted

com.ms.wfc.ui.RichEdit.WC_RICHEDIT could not be converted

com.ms.wfc.ui.RichEditIMEColor could not be converted

com.ms.wfc.ui.RichEditIMEOptions could not be converted

com.ms.wfc.ui.SameParentReferenceEditor could not be converted

com.ms.wfc.ui.SaveFileDialog.runFileDialog could not be converted

com.ms.wfc.ui.Screen.MonitorEnumProc could not be converted

com.ms.wfc.ui.Screen.MONITORINFO could not be converted

com.ms.wfc.ui.Screen.MONITORINFOEX could not be converted

com.ms.wfc.ui.ScrollBar.propertyChanged could not be converted

com.ms.wfc.ui.ScrollEvent.ScrollEvent could not be converted

com.ms.wfc.ui.SelectionRange.getConstructorArgs could not be converted

com.ms.wfc.ui.SelectionStyle could not be converted

com.ms.wfc.ui.Splitter.postFilterMessage could not be converted

com.ms.wfc.ui.SplitterEventHandler.SplitterEventHandler could not be converted

com.ms.wfc.ui.TabBase.propertyChanged could not be converted

com.ms.wfc.ui.TabBase.setSelectedIndex could not be converted

com.ms.wfc.ui.TabStrip.insertTab could not be converted

com.ms.wfc.ui.TextFormat.BOTTOM could not be converted

com.ms.wfc.ui.TextFormat.EDITCONTROL could not be converted

com.ms.wfc.ui.TextFormat.ENDELLIPSIS could not be converted

com.ms.wfc.ui.TextFormat.EXPANDTABS could not be converted

com.ms.wfc.ui.TextFormat.HORIZONTALCENTER could not be converted

com.ms.wfc.ui.TextFormat.LEFT could not be converted

com.ms.wfc.ui.TextFormat.NOPREFIX could not be converted

com.ms.wfc.ui.TextFormat.PATHELLIPSIS could not be converted

com.ms.wfc.ui.TextFormat.RIGHT could not be converted

com.ms.wfc.ui.TextFormat.RIGHTTOLEFT could not be converted

com.ms.wfc.ui.TextFormat.SINGLELINE could not be converted

com.ms.wfc.ui.TextFormat.TextFormat could not be converted

com.ms.wfc.ui.TextFormat.TOP could not be converted

com.ms.wfc.ui.TextFormat.valid could not be converted

com.ms.wfc.ui.TextFormat.VERTICALCENTER could not be converted

com.ms.wfc.ui.TextFormat.WORDBREAK could not be converted

com.ms.wfc.ui.Toolbar.propertyChanged could not be converted

com.ms.wfc.ui.ToolTip.shouldPersistAutomaticDelay could not be converted

com.ms.wfc.ui.ToolTip.shouldPersistAutoPopDelay could not be converted

com.ms.wfc.ui.ToolTip.shouldPersistInitialDelay could not be converted

com.ms.wfc.ui.ToolTip.shouldPersistReshowDelay could not be converted

com.ms.wfc.ui.TrackBar.propertyChanged could not be converted

com.ms.wfc.ui.TreeNode.getConstructorArgs could not be converted

com.ms.wfc.ui.TreeNode.save could not be converted

com.ms.wfc.ui.TreeNode.TreeNode could not be converted

com.ms.wfc.ui.TreeNodesEditDialog could not be converted

com.ms.wfc.ui.TreeNodesEditor could not be converted

com.ms.wfc.ui.TreeView.shouldPersistIndent could not be converted

com.ms.wfc.ui.UpDown.getAcceleration could not be converted

com.ms.wfc.ui.UpDown.getAutoBuddy could not be converted

com.ms.wfc.ui.UpDown.getAutoSize could not be converted

com.ms.wfc.ui.UpDown.getBuddyControl could not be converted

com.ms.wfc.ui.UpDown.getHorizontal could not be converted

com.ms.wfc.ui.UpDown.getModifyBuddy could not be converted

com.ms.wfc.ui.UpDown.getRadix could not be converted

com.ms.wfc.ui.UpDown.getWrap could not be converted

com.ms.wfc.ui.UpDown.onCreateHandle could not be converted

com.ms.wfc.ui.UpDown.setAcceleration could not be converted

com.ms.wfc.ui.UpDown.setAutoBuddy could not be converted

com.ms.wfc.ui.UpDown.setAutoSize could not be converted

com.ms.wfc.ui.UpDown.setBuddyControl could not be converted

com.ms.wfc.ui.UpDown.setHorizontal could not be converted

com.ms.wfc.ui.UpDown.setModifyBuddy could not be converted

com.ms.wfc.ui.UpDown.setRadix could not be converted

com.ms.wfc.ui.UpDown.setWrap could not be converted

com.ms.wfc.ui.UpDown.shouldPersistBuddyControl could not be converted

com.ms.wfc.ui.UpDownAcceleration could not be converted

com.ms.wfc.ui.UpDownAlignment.MANUAL could not be converted

com.ms.wfc.ui.UpDownChangedEvent.delta could not be converted

com.ms.wfc.ui.UpDownChangedEvent.value could not be converted

Com.ms.wfc.util Error Messages

com.ms.wfc.util.Debug.addOnDisplayAssert could not be converted

com.ms.wfc.util.Debug.addOnDisplayMessage could not be converted

com.ms.wfc.util.Debug.Debug could not be converted

com.ms.wfc.util.Debug.displaySwitches could not be converted

com.ms.wfc.util.Debug.getSwitchListText could not be converted

com.ms.wfc.util.Debug.printEnumIf could not be converted

com.ms.wfc.util.Debug.printExceptionIf could not be converted

com.ms.wfc.util.Debug.printIf could not be converted

com.ms.wfc.util.Debug.printlnIf could not be converted

com.ms.wfc.util.Debug.printObjectIf could not be converted

com.ms.wfc.util.Debug.printSwitchList could not be converted

com.ms.wfc.util.Debug.printStackTraceIf could not be converted

com.ms.wfc.util.Debug.removeOnDisplayAssert could not be converted

com.ms.wfc.util.Debug.removeOnDisplayMessage could not be converted

com.ms.wfc.util.DebugMessageEventHandler could not be converted

com.ms.wfc.util.HandleCollector could not be converted

com.ms.wfc.util.HashTable.addSlot could not be converted

com.ms.wfc.util.HashTable.findItem could not be converted

com.ms.wfc.util.HashTable.getKeys could not be converted

com.ms.wfc.util.HashTable.getValues could not be converted

com.ms.wfc.util.HashTable.growHashTable could not be converted

com.ms.wfc.util.HashTable.HashTable could not be converted

com.ms.wfc.util.HashTable.hashValues could not be converted

com.ms.wfc.util.HashTable.iMax could not be converted

com.ms.wfc.util.HashTable.maxAverageDepth could not be converted

com.ms.wfc.util.HashTable.rehash could not be converted

com.ms.wfc.util.HashTable.removeSlot could not be converted

com.ms.wfc.util.HashTable.removeValue could not be converted

com.ms.wfc.util.HashTable.sizes could not be converted

com.ms.wfc.util.List.capInc could not be converted

com.ms.wfc.util.List.clone could not be converted

com.ms.wfc.util.List.getVersion could not be converted

com.ms.wfc.util.List.version could not be converted

com.ms.wfc.util.NumberFormat could not be converted

com.ms.wfc.util.Root.checkLeaks could not be converted

com.ms.wfc.util.Root.get could not be converted

com.ms.wfc.util.StringSorter.compare could not be converted

com.ms.wfc.util.StringSorter.DESCENDING could not be converted

com.ms.wfc.util.StringSorter.sort could not be converted

com.ms.wfc.util.StringSorter.sort(Locale, string,.Object, int, int, int) could not be converted

com.ms.wfc.util.Utils.getJavaIdentifier could not be converted

com.ms.wfc.util.Utils.getPrimitiveWord could not be converted

com.ms.wfc.util.Utils.getReservedWord could not be converted

com.ms.wfc.util.Utils.validateIdentifier could not be converted

com.ms.wfc.util.Value.format could not be converted

com.ms.wfc.util.Value.formatCurrency could not be converted

com.ms.wfc.util.Value.formatNumber could not be converted

com.ms.wfc.util.Value.SysFreeString could not be converted

Com.ms.wfc.win32 Error Messages

com.ms.wfc.win32.CharBuffer could not be converted

com.ms.wfc.win32.Windows.ACCELERATORHANDLE could not be converted

com.ms.wfc.win32.Windows.BeginPaint could not be converted

com.ms.wfc.win32.Windows.CloseEnhMetaFile could not be converted

com.ms.wfc.win32.Windows.CloseHandle could not be converted

com.ms.wfc.win32.Windows.CopyEnhMetaFile could not be converted

com.ms.wfc.win32.Windows.CopyImage could not be converted

com.ms.wfc.win32.Windows.CreateAcceleratorTable could not be converted

com.ms.wfc.win32.Windows.CreateBitmap could not be converted

com.ms.wfc.win32.Windows.CreateBitmap(int, int, int, int, int[]) could not be converted

com.ms.wfc.win32.Windows.CreateBitmap(int, int, int, int, short[]) could not be converted

com.ms.wfc.win32.Windows.CreateBrushIndirect could not be converted

com.ms.wfc.win32.Windows.CreateCompatibleBitmap could not be converted

com.ms.wfc.win32.Windows.CreateCompatibleDC could not be converted

com.ms.wfc.win32.Windows.CreateDC could not be converted

com.ms.wfc.win32.Windows.CreateDIBitmap could not be converted

com.ms.wfc.win32.Windows.CreateEllipticRgn could not be converted

com.ms.wfc.win32.Windows.CreateEllipticRgnIndirect could not be converted

com.ms.wfc.win32.Windows.CreateEnhMetaFile could not be converted

com.ms.wfc.win32.Windows.CreateFile could not be converted

com.ms.wfc.win32.Windows.CreateFont could not be converted

com.ms.wfc.win32.Windows.CreateFontIndirect could not be converted

com.ms.wfc.win32.Windows.CreateHalftonePalette could not be converted

com.ms.wfc.win32.Windows.CreateHatchBrush could not be converted

com.ms.wfc.win32.Windows.CreateIC could not be converted

com.ms.wfc.win32.Windows.CreateILockBytesOnHGlobal could not be converted

com.ms.wfc.win32.Windows.CreateMappedBitmap could not be converted

com.ms.wfc.win32.Windows.CreateMenu could not be converted

com.ms.wfc.win32.Windows.CreatePalette could not be converted

com.ms.wfc.win32.Windows.CreatePatternBrush could not be converted

com.ms.wfc.win32.Windows.CreatePen could not be converted

com.ms.wfc.win32.Windows.CreatePenIndirect could not be converted

com.ms.wfc.win32.Windows.CreatePolygonRgn could not be converted

com.ms.wfc.win32.Windows.CreatePolyPolygonRgn could not be converted

com.ms.wfc.win32.Windows.CreatePopupMenu could not be converted

com.ms.wfc.win32.Windows.CreateProcess could not be converted

com.ms.wfc.win32.Windows.CreateRectRgn could not be converted

com.ms.wfc.win32.Windows.CreateRectRgnIndirect could not be converted

com.ms.wfc.win32.Windows.CreateRoundRectRgn could not be converted

com.ms.wfc.win32.Windows.CreateSolidBrush could not be converted

com.ms.wfc.win32.Windows.CreateWindowEx could not be converted

com.ms.wfc.win32.Windows.CURSORHANDLE could not be converted

com.ms.wfc.win32.Windows.DeleteDC could not be converted

com.ms.wfc.win32.Windows.DeleteEnhMetaFile could not be converted

com.ms.wfc.win32.Windows.DeleteObject could not be converted

com.ms.wfc.win32.Windows.DestroyAcceleratorTable could not be converted

com.ms.wfc.win32.Windows.DestroyCursor could not be converted

com.ms.wfc.win32.Windows.DestroyIcon could not be converted

com.ms.wfc.win32.Windows.DestroyMenu could not be converted

com.ms.wfc.win32.Windows.DestroyWindow could not be converted

com.ms.wfc.win32.Windows.DoDragDrop could not be converted

com.ms.wfc.win32.Windows.DrawState could not be converted

com.ms.wfc.win32.Windows.DuplicateHandle could not be converted

com.ms.wfc.win32.Windows.EMFHANDLE could not be converted

com.ms.wfc.win32.Windows.EndPaint could not be converted

com.ms.wfc.win32.Windows.EnumObjects could not be converted

com.ms.wfc.win32.Windows.EnumThreadWindows could not be converted

com.ms.wfc.win32.Windows.FindClose could not be converted

com.ms.wfc.win32.Windows.FindFirstFile could not be converted

com.ms.wfc.win32.Windows.FINDHANDLE could not be converted

com.ms.wfc.win32.Windows.GDIHANDLE could not be converted

com.ms.wfc.win32.Windows.GetDC could not be converted

com.ms.wfc.win32.Windows.GetDCEx could not be converted

com.ms.wfc.win32.Windows.GetEnhMetaFile could not be converted

com.ms.wfc.win32.Windows.GetHGlobalFromILockBytes could not be converted

com.ms.wfc.win32.Windows.GetWindowDC could not be converted

com.ms.wfc.win32.Windows.HDCHANDLE could not be converted

com.ms.wfc.win32.Windows.ICONHANDLE could not be converted

com.ms.wfc.win32.Windows.InvalidateRect could not be converted

com.ms.wfc.win32.Windows.KERNELHANDLE could not be converted

com.ms.wfc.win32.Windows.MENUHANDLE could not be converted

com.ms.wfc.win32.Windows.OleCreatePictureIndirect could not be converted

com.ms.wfc.win32.Windows.PathToRegion could not be converted

com.ms.wfc.win32.Windows.RegisterDragDrop could not be converted

com.ms.wfc.win32.Windows.ReleaseDC could not be converted

com.ms.wfc.win32.Windows.SetEnhMetaFileBits could not be converted

com.ms.wfc.win32.Windows.SetMetaFileBitsEx could not be converted

com.ms.wfc.win32.Windows.SetWindowRgn could not be converted

com.ms.wfc.win32.Windows.SetWinMetaFileBits could not be converted

com.ms.wfc.win32.Windows.StgCreateDocfileOnILockBytes could not be converted

com.ms.wfc.win32.Windows.StgOpenStorageOnILockBytes could not be converted

com.ms.wfc.win32.Windows.WINDOWHANDLE could not be converted

Com.ms.win32 Error Messages

com.ms.win32.Kernel32.CreateFiber could not be converted

com.ms.win32.Ole32.CoCreateInstanceEx could not be converted

com.ms.win32.Ole32.CoGetInstanceFromFile could not be converted

com.ms.win32.Ole32.CoGetInstanceFromIStorage could not be converted

com.ms.win32.Ole32.CreateILockBytesOnHGlobal could not be converted

com.ms.win32.Ole32.GetHGlobalFromILockBytes could not be converted

com.ms.win32.Ole32.StgCreateDocfile could not be converted

com.ms.win32.Ole32.StgCreateStorageEx could not be converted

com.ms.win32.Ole32.StgOpenLayoutDocfile could not be converted

com.ms.win32.Ole32.StgOpenStorage could not be converted

com.ms.win32.Spoolss.EnumPrinters could not be converted

Java.applet Error Messages

java.applet.Applet.getAccessibleContext could not be converted

java.applet.Applet.getAppletContext could not be converted

java.applet.Applet.getAppletInfo could not be converted

java.applet.Applet.getAudioClip could not be converted

java.applet.Applet.getCodeBase could not be converted

java.applet.Applet.getDocumentBase could not be converted

java.applet.Applet.getImage could not be converted

java.applet.Applet.getLocale could not be converted

java.applet.Applet.getParameter could not be converted

java.applet.Applet.getParameterInfo could not be converted

java.applet.Applet.init could not be converted

java.applet.Applet.isActive could not be converted

java.applet.Applet.newAudioClip could not be converted

java.applet.Applet.play could not be converted

java.applet.Applet.resize could not be converted

java.applet.Applet.setStub could not be converted

java.applet.Applet.showStatus could not be converted

java.applet.Applet.start could not be converted

java.applet.Applet.stop could not be converted

java.applet.AppletContext could not be converted

java.applet.AppletStub could not be converted

java.applet.AudioClip could not be converted

Java.awt Error Messages

java.awt.Adjustable could not be converted

java.awt.AWTError could not be converted

java.awt.AWTEvent could not be converted

java.awt.AWTEventMulticaster could not be converted

java.awt.BorderLayout could not be converted

java.awt.Button.getActionCommand could not be converted

java.awt.Button.setActionCommand could not be converted

java.awt.Canvas could not be converted

java.awt.CardLayout could not be converted

java.awt.CheckboxGroup.getCurrent could not be converted

java.awt.CheckboxGroup.getSelectedCheckbox could not be converted

java.awt.<ClassName>.addNotify could not be converted

java.awt.<ClassName>.getPeer could not be converted

java.awt.<ClassName>.print* could not be converted

java.awt.<ClassName>.processActionEvent could not be converted

java.awt.<ClassName>.processEvent could not be converted

java.awt.<ClassName>.processItemEvent could not be converted

java.awt.<ClassName>.removeNotify could not be converted

java.awt.Color.decode could not be converted

java.awt.Color.getColor could not be converted

java.awt.Color.getHSBColor could not be converted

java.awt.Color.HSBtoRGB could not be converted

java.awt.Color.RGBtoHSB could not be converted

java.awt.Component.action could not be converted

java.awt.Component.BOTTOM_ALIGNMENT could not be converted

java.awt.Component.CENTER_ALIGNMENT could not be converted

java.awt.Component.checkImage could not be converted

java.awt.Component.createImage(ImageProducer) could not be converted

java.awt.Component.createImage(int, int) could not be converted

java.awt.Component.deliverEvent could not be converted

java.awt.Component.disableEvents could not be converted

java.awt.Component.dispatchEvent could not be converted

java.awt.Component.enableEvents could not be converted

java.awt.Component.getAlignmentX could not be converted

java.awt.Component.getAlignmentY could not be converted

java.awt.Component.getColorModel could not be converted

java.awt.Component.getLocale could not be converted

java.awt.Component.getMaximumSize could not be converted

java.awt.Component.getMinimumSize could not be converted

java.awt.Component.getPreferredSize could not be converted

java.awt.Component.getToolkit could not be converted

java.awt.Component.getTreeLock could not be converted

java.awt.Component.handleEvent could not be converted

java.awt.Component.imageUpdate could not be converted

java.awt.Component.isValid could not be converted

java.awt.Component.keyDown could not be converted

java.awt.Component.keyUp could not be converted

java.awt.Component.LEFT_ALIGNMENT could not be converted

java.awt.Component.lostFocus could not be converted

java.awt.Component.minimumSize could not be converted

java.awt.Component.mouseDown could not be converted

java.awt.Component.mouseDrag could not be converted

java.awt.Component.mouseEnter could not be converted

java.awt.Component.mouseExit could not be converted

java.awt.Component.mouseMove could not be converted

java.awt.Component.mouseUp could not be converted

java.awt.Component.move could not be converted

java.awt.Component.nextFocus could not be converted

java.awt.Component.postEvent could not be converted

java.awt.Component.preferredSize could not be converted

java.awt.Component.prepareImage could not be converted

java.awt.Component.process*Event could not be converted

java.awt.Component.remove could not be converted

java.awt.Component.RIGHT_ALIGNMENT could not be converted

java.awt.Component.setLocale could not be converted

java.awt.Component.TOP_ALIGNMENT could not be converted

java.awt.Component.transferFocus could not be converted

java.awt.Component.validate could not be converted

java.awt.Container.deliverEvent could not be converted

java.awt.Container.getLayout could not be converted

java.awt.Container.paint* could not be converted

java.awt.Container.processContainerEvent could not be converted

java.awt.Container.setLayout could not be converted

java.awt.Container.validateTree could not be converted

java.awt.Cursor.predefined could not be converted

java.awt.datatransfer.Clipboard.Clipboard could not be converted

java.awt.datatransfer.Clipboard.contents could not be converted

java.awt.datatransfer.Clipboard.getName could not be converted

java.awt.datatransfer.Clipboard.owner could not be converted

java.awt.datatransfer.ClipboardOwner could not be converted

java.awt.datatransfer.DataFlavor.DataFlavor(class, string) could not be converted

java.awt.datatransfer.DataFlavor.DataFlavor(string, string) could not be converted

java.awt.datatransfer.DataFlavor.getHumanPresentableName could not be converted

java.awt.datatransfer.DataFlavor.getMimeType could not be converted

java.awt.datatransfer.DataFlavor.getRepresentationClass could not be converted

java.awt.datatransfer.DataFlavor.isMimeTypeEqual could not be converted

java.awt.datatransfer.DataFlavor.normalizeMimeType could not be converted

java.awt.datatransfer.DataFlavor.normalizeMimeTypeParameter could not be converted

java.awt.datatransfer.DataFlavor.setHumanPresentableName could not be converted

java.awt.datatransfer.StringSelection.isDataFlavorSupported could not be converted

java.awt.datatransfer.StringSelection.lostOwnership could not be converted

java.awt.datatransfer.Transferable.isDataFlavorSupported could not be converted

java.awt.Dialog.setModal could not be converted

java.awt.Event could not be converted

java.awt.event.ActionEvent.ACTION_PERFORMED could not be converted

java.awt.event.ActionEvent.getActionCommand could not be converted

java.awt.event.ActionEvent.<KeyName>_MASK could not be converted

java.awt.event.<ClassName>.<EventType>_FIRST could not be converted

java.awt.event.<ClassName>.<EventType>_LAST could not be converted

java.awt.event.ComponentEvent.COMPONENT_<EventType> could not be converted

java.awt.event.FocusEvent.FOCUS_<EventType> could not be converted

java.awt.event.FocusEvent.isTemporary could not be converted

java.awt.event.InputEvent could not be converted

java.awt.event.ItemEvent.<EventType> could not be converted

java.awt.event.ItemEvent.getStateChange could not be converted

java.awt.event.KeyEvent.CHAR_UNDEFINED could not be converted

java.awt.event.KeyEvent.getKeyCode could not be converted

java.awt.event.KeyEvent.KEY_<EventType> could not be converted

java.awt.event.KeyEvent.setKeyChar could not be converted

java.awt.event.KeyEvent.setKeyCode could not be converted

java.awt.event.KeyEvent.setModifiers could not be converted

java.awt.event.KeyEvent.VK_<CharacterName> could not be converted

java.awt.event.KeyEvent.VK_UNDEFINED could not be converted

java.awt.event.MouseEvent.isPopupTrigger could not be converted

java.awt.event.MouseEvent.MOUSE_<EventType> could not be converted

java.awt.event.MouseEvent.translatePoint could not be converted

java.awt.event.PaintEvent.<EventType> could not be converted

java.awt.event.PaintEvent.setUpdateRect could not be converted

java.awt.event.TextEvent.TEXT_VALUE_CHANGED could not be converted

java.awt.event.WindowEvent.WINDOW_<EventType> could not be converted

java.awt.EventQueue could not be converted

java.awt.FileDialog.FileDialog could not be converted

java.awt.FileDialog.*FilenameFilter could not be converted

java.awt.FileDialog.setMode could not be converted

java.awt.FlowLayout could not be converted

java.awt.Font.getFont could not be converted

java.awt.Font.PLAIN could not be converted

java.awt.FontMetrics.FontMetrics could not be converted

java.awt.FontMetrics.getLeading could not be converted

java.awt.FontMetrics.getMaxAdvance could not be converted

java.awt.FontMetrics.getWidths could not be converted

java.awt.FontMetrics.*Width could not be converted

java.awt.Frame.getCursorType could not be converted

java.awt.Frame.getIconImage could not be converted

java.awt.Frame.setIconImage could not be converted

java.awt.Graphics.copyArea could not be converted

java.awt.Graphics.drawBytes could not be converted

java.awt.Graphics.drawChars could not be converted

java.awt.Graphics.drawRoundRect could not be converted

java.awt.Graphics.drawString could not be converted

java.awt.Graphics.fill3DRect could not be converted

java.awt.Graphics.fillRoundRect could not be converted

java.awt.Graphics.finalize could not be converted

java.awt.Graphics.getClip could not be converted

java.awt.Graphics.setClip could not be converted

java.awt.Graphics.setPaintMode could not be converted

java.awt.Graphics.setXORMode could not be converted

java.awt.GridBagConstraints could not be converted

java.awt.GridBagLayout could not be converted

java.awt.GridLayout could not be converted

java.awt.image.<ClassName>.*Consumer could not be converted

java.awt.image.<ClassName>.imageComplete could not be converted

java.awt.image.<ClassName>.setHints could not be converted

java.awt.image.<ClassName>.setPixels could not be converted

java.awt.image.<ClassName>.setProperties could not be converted

java.awt.image.<ClassName>.*TopDownLeftRight* could not be converted

java.awt.image.FilteredImageSource.startProduction could not be converted

java.awt.Image.getProperty could not be converted

java.awt.image.ImageConsumer could not be converted

java.awt.image.ImageFilter.setColorModel could not be converted

java.awt.image.ImageObserver could not be converted

java.awt.image.ImageProducer.startProduction could not be converted

java.awt.image.MemoryImageSource could not be converted

java.awt.image.PixelGrabber.abortGrabbing could not be converted

java.awt.image.PixelGrabber.getStatus could not be converted

java.awt.image.PixelGrabber.setColorModel could not be converted

java.awt.image.PixelGrabber.startGrabbing could not be converted

java.awt.image.PixelGrabber.status could not be converted

java.awt.image.ReplicateScaleFilter.outpixbuf could not be converted

java.awt.image.ReplicateScaleFilter.src* could not be converted

java.awt.image.RGBImageFilter could not be converted

java.awt.Image.SCALE_<Algorithm> could not be converted

java.awt.Image.UndefinedProperty could not be converted

java.awt.ItemSelectable could not be converted

java.awt.LayoutManager could not be converted

java.awt.LayoutManager2 could not be converted

java.awt.List.delItems could not be converted

java.awt.MediaTracker could not be converted

java.awt.Menu.isTearOff could not be converted

java.awt.MenuBar.deleteShortcut could not be converted

java.awt.MenuBar.getHelpMenu could not be converted

java.awt.MenuBar.getShortcutMenuItem could not be converted

java.awt.MenuBar.setHelpMenu could not be converted

java.awt.MenuBar.shortcuts could not be converted

java.awt.MenuComponent.dispatchEvent could not be converted

java.awt.MenuComponent.MenuComponent could not be converted

java.awt.MenuComponent.postEvent could not be converted

java.awt.MenuComponent.setFont could not be converted

java.awt.MenuContainer.postEvent could not be converted

java.awt.MenuItem.disableEvents could not be converted

java.awt.MenuItem.enableEvents could not be converted

java.awt.MenuItem.getActionCommand could not be converted

java.awt.MenuItem.setActionCommand could not be converted

java.awt.MenuShortcut.usesShiftModifier could not be converted

java.awt.Panel.Panel could not be converted

java.awt.Polygon.addPoint could not be converted

java.awt.Polygon.Polygon could not be converted

java.awt.PrintGraphics could not be converted

java.awt.PrintJob could not be converted

java.awt.Scrollbar could not be converted

java.awt.Scrollbar.addAdjustmentListener could not be converted

java.awt.Scrollbar.Scrollbar could not be converted

java.awt.ScrollPane.get*Adjustable could not be converted

java.awt.ScrollPane.printComponents could not be converted

java.awt.ScrollPane.ScrollPane could not be converted

java.awt.ScrollPane.setLayout could not be converted

java.awt.Shape could not be converted

java.awt.SystemColor.NUM_COLORS could not be converted

java.awt.SystemColor.TEXT could not be converted

java.awt.SystemColor.TEXT_HIGHLIGHT could not be converted

java.awt.SystemColor.TEXT_HIGHLIGHT_TEXT could not be converted

java.awt.SystemColor.TEXT_INACTIVE_TEXT could not be converted

java.awt.SystemColor.TEXT_TEXT could not be converted

java.awt.SystemColor.textHighlight could not be converted

java.awt.SystemColor.textHighlightText could not be converted

java.awt.SystemColor.textInactiveText could not be converted

java.awt.SystemColor.textText could not be converted

java.awt.TextArea.*Columns could not be converted

java.awt.TextArea.*Rows could not be converted

java.awt.TextComponent.processTextEvent could not be converted

java.awt.TextComponent.textListener could not be converted

java.awt.TextField.*Columns could not be converted

java.awt.TextField.getColumns could not be converted

java.awt.TextField.getPreferredSize could not be converted

java.awt.TextField.preferredSize could not be converted

java.awt.TextField.setColumns could not be converted

java.awt.Toolkit could not be converted

java.awt.Window.getToolkit could not be converted

java.awt.Window.getWarningString could not be converted

java.awt.Window.pack could not be converted

java.awt.Window.postEvent could not be converted

java.awt.Window.processWindowEvent could not be converted

Java.beans Error Messages

java.beans.BeanDescriptor.BeanDescriptor could not be converted

java.beans.BeanDescriptor.getCustomizerClass could not be converted

java.beans.BeanInfo.ICON_COLOR_16x16 could not be converted

java.beans.BeanInfo.ICON_COLOR_32x32 could not be converted

java.beans.BeanInfo.ICON_MONO_16x16 could not be converted

java.beans.BeanInfo.ICON_MONO_32x32 could not be converted

Java.io Error Messages

java.io.BufferedInputStream.available could not be converted

java.io.BufferedInputStream.buf could not be converted

java.io.BufferedInputStream.marklimit could not be converted

java.io.BufferedInputStream.markpos could not be converted

java.io.BufferedOutputStream.buf could not be converted

java.io.BufferedReader.BufferedReader could not be converted

java.io.BufferedReader.mark could not be converted

java.io.BufferedReader.reset could not be converted

java.io.BufferedWriter.BufferedWriter could not be converted

java.io.BufferedWriter.write could not be converted

java.io.ByteArrayInputStream.buf could not be converted

java.io.ByteArrayInputStream.mark~ could not be converted

java.io.ByteArrayOutputStream.buf could not be converted

java.io.ByteArrayOutputStream.reset could not be converted

java.io.ByteArrayOutputStream.toString could not be converted

java.io.CharArrayReader.buf could not be converted

java.io.CharArrayReader.count could not be converted

java.io.CharArrayReader.markedPos could not be converted

java.io.CharArrayReader.mark could not be converted

java.io.CharArrayReader.pos could not be converted

java.io.CharArrayReader.reset could not be converted

java.io.CharArrayReader.skip could not be converted

java.io.CharArrayWriter.reset could not be converted

java.io.CharArrayWriter.size could not be converted

java.io.CharArrayWriter.toCharArray could not be converted

java.io.CharArrayWriter.toString could not be converted

java.io.CharArrayWriter.writeTo could not be converted

java.io.DataInput.readLine could not be converted

java.io.DataInput.readUTF could not be converted

java.io.DataInputStream.readLine could not be converted

java.io.DataInputStream.readUTF could not be converted

java.io.DataOutput.writeBytes could not be converted

java.io.DataOutput.writeChars could not be converted

java.io.DataOutput.writeUTF could not be converted

java.io.DataOutputStream.writeBytes could not be converted

java.io.DataOutputStream.writeChars could not be converted

java.io.DataOutputStream.writeUTF could not be converted

java.io.DataOutputStream.written could not be converted

java.io.Externalizable could not be converted

java.io.File.canRead could not be converted

java.io.File.isAbsolute could not be converted

java.io.File.list could not be converted

java.io.FileDescriptor could not be converted

java.io.FileInputStream.available could not be converted

java.io.FileInputStream.FileInputStream could not be converted

java.io.FileInputStream.getFD could not be converted

java.io.FilenameFilter could not be converted

java.io.FileOutputStream.FileOutputStream could not be converted

java.io.FileOutputStream.getFD could not be converted

java.io.FileOutputStream.write could not be converted

java.io.FileReader.FileReader could not be converted

java.io.FileWriter.FileWriter could not be converted

java.io.FilterInputStream.available could not be converted

java.io.FilterInputStream.close could not be converted

java.io.FilterInputStream.mark could not be converted

java.io.FilterInputStream.markSupported could not be converted

java.io.FilterInputStream.reset could not be converted

java.io.FilterReader.in could not be converted

java.io.FilterReader.mark could not be converted

java.io.FilterReader.markSupported could not be converted

java.io.FilterReader.reset could not be converted

java.io.InputStream.available could not be converted

java.io.InputStream.mark could not be converted

java.io.InputStream.markSupported could not be converted

java.io.InputStream.read could not be converted

java.io.InputStream.reset could not be converted

java.io.InterruptedIOException.bytesTransferred could not be converted

java.io.InvalidClassException.classname could not be converted

java.io.InvalidClassException.InvalidClassException could not be converted

java.io.LineNumberInputStream.available could not be converted

java.io.LineNumberInputStream.getLineNumber could not be converted

java.io.LineNumberInputStream.mark could not be converted

java.io.LineNumberInputStream.reset could not be converted

java.io.LineNumberInputStream.setLineNumber could not be converted

java.io.LineNumberReader.getLineNumber could not be converted

java.io.LineNumberReader.LineNumberReader could not be converted

java.io.LineNumberReader.mark could not be converted

java.io.LineNumberReader.reset could not be converted

java.io.LineNumberReader.setLineNumber could not be converted

java.io.ObjectInput.available could not be converted

java.io.ObjectInput.readObject could not be converted

java.io.ObjectInputStream.available could not be converted

java.io.ObjectInputStream.defaultReadObject could not be converted

java.io.ObjectInputStream.enableResolveObject could not be converted

java.io.ObjectInputStream.readLine could not be converted

java.io.ObjectInputStream.readStreamHeader could not be converted

java.io.ObjectInputStream.readUTF could not be converted

java.io.ObjectInputStream.registerValidation could not be converted

java.io.ObjectInputStream.resolveClass could not be converted

java.io.ObjectInputStream.resolveObject could not be converted

java.io.ObjectInputValidation could not be converted

java.io.ObjectOutput.writeObject could not be converted

java.io.ObjectOutputStream.annotateClass could not be converted

java.io.ObjectOutputStream.baseWireHandle could not be converted

java.io.ObjectOutputStream.defaultWriteObject could not be converted

java.io.ObjectOutputStream.enableReplaceObject could not be converted

java.io.ObjectOutputStream.replaceObject could not be converted

java.io.ObjectOutputStream.reset could not be converted

java.io.ObjectOutputStream.SC_EXTERNALIZABLE could not be converted

java.io.ObjectOutputStream.SC_SERIALIZABLE could not be converted

java.io.ObjectOutputStream.SC_WRITE_METHOD could not be converted

java.io.ObjectOutputStream.writeBytes could not be converted

java.io.ObjectOutputStream.writeChars could not be converted

java.io.ObjectOutputStream.writeStreamHeader could not be converted

java.io.ObjectOutputStream.writeUTF could not be converted

java.io.ObjectStreamClass could not be converted

java.io.OptionalDataException could not be converted

java.io.OutputStreamWriter.getEncoding could not be converted

java.io.PipedInputStream.available could not be converted

java.io.PipedInputStream.buffer could not be converted

java.io.PipedInputStream.connect could not be converted

java.io.PipedInputStream.in could not be converted

java.io.PipedInputStream.out could not be converted

java.io.PipedInputStream.PIPE_SIZE could not be converted

java.io.PipedInputStream.PipedInputStream could not be converted

java.io.PipedInputStream.read could not be converted

java.io.PipedInputStream.receive could not be converted

java.io.PipedOutputStream.connect could not be converted

java.io.PipedOutputStream.PipedOutputStream could not be converted

java.io.PipedOutputStream.write could not be converted

java.io.PipedReader.connect could not be converted

java.io.PipedWriter.connect could not be converted

java.io.PrintStream.checkError could not be converted

java.io.PrintStream.setError could not be converted

java.io.PrintWriter.PrintWriter could not be converted

java.io.PrintWriter.setError could not be converted

java.io.PushbackInputStream.available could not be converted

java.io.RandomAccessFile.getFD could not be converted

java.io.RandomAccessFile.readUTF could not be converted

java.io.RandomAccessFile.writeUTF could not be converted

java.io.Reader.lock could not be converted

java.io.Reader.mark could not be converted

java.io.Reader.markSupported could not be converted

java.io.Reader.read could not be converted

java.io.Reader.Reader could not be converted

java.io.Reader.reset could not be converted

java.io.SequenceInputStream.available could not be converted

java.io.SequenceInputStream.SequenceInputStream could not be converted

java.io.StreamTokenizer could not be converted

java.io.StringBufferInputStream.available could not be converted

java.io.StringBufferInputStream.buffer could not be converted

java.io.StringBufferInputStream.count could not be converted

java.io.StringBufferInputStream.pos could not be converted

java.io.StringBufferInputStream.read could not be converted

java.io.StringBufferInputStream.reset could not be converted

java.io.StringBufferInputStream.skip could not be converted

java.io.StringReader.mark could not be converted

java.io.StringReader.markSupported could not be converted

java.io.StringReader.ready could not be converted

java.io.StringReader.reset could not be converted

java.io.StringReader.skip could not be converted

java.io.StringWriter.getBuffer could not be converted

java.io.Writer.lock could not be converted

Java.lang Error Messages

java.lang.Boolean.getBoolean could not be converted

java.lang.Character.forDigit could not be converted

java.lang.Character.isDefined could not be converted

java.lang.Character.isIdentifierIgnorable could not be converted

java.lang.Character.isJavaIdentifierPart could not be converted

java.lang.Character.isUnicodeIdentifierPart could not be converted

java.lang.Character.isUnicodeIdentifierStart could not be converted

java.lang.Class.getClassLoader could not be converted

java.lang.Class.getModifiers could not be converted

java.lang.Class.getResource could not be converted

java.lang.Class.getResourceAsStream could not be converted

java.lang.Class.getSigners could not be converted

java.lang.ClassLoader could not be converted

java.lang.Compiler could not be converted

java.lang.Double.doubleToLongBits could not be converted

java.lang.Double.longBitsToDouble could not be converted

java.lang.ExceptionInInitializerError.getException could not be converted

java.lang.Float.floatToIntBits could not be converted

java.lang.Float.intBitsToFloat could not be converted

java.lang.Integer.getInteger could not be converted

java.lang.Integer.TYPE could not be converted

java.lang.Long.getLong could not be converted

java.lang.Number could not be converted

java.lang.reflect.Constructor.getExceptionTypes could not be converted

java.lang.reflect.Constructor.newInstance could not be converted

java.lang.reflect.Field.getModifiers could not be converted

java.lang.reflect.Field.setByte could not be converted

java.lang.reflect.Field.setChar could not be converted

java.lang.reflect.Field.setShort could not be converted

java.lang.reflect.InvocationTargetException.InvocationTargetException could not be converted

java.lang.reflect.Member.DECLARED could not be converted

java.lang.reflect.Member.getModifiers could not be converted

java.lang.reflect.Member.PUBLIC could not be converted

java.lang.reflect.Method.getExceptionTypes could not be converted

java.lang.reflect.Method.getModifiers could not be converted

java.lang.reflect.Modifier could not be converted

java.lang.Runtime.exec could not be converted

java.lang.Runtime.getLocalizedInputStream could not be converted

java.lang.Runtime.getLocalizedOutputStream could not be converted

java.lang.Runtime.load could not be converted

java.lang.Runtime.loadLibrary could not be converted

java.lang.Runtime.runFinalizersOnExit could not be converted

java.lang.Runtime.traceInstructions could not be converted

java.lang.Runtime.traceMethodCalls could not be converted

java.lang.SecurityManager.checkAccess could not be converted

java.lang.SecurityManager.checkAwtEventQueueAccess could not be converted

java.lang.SecurityManager.checkConnect could not be converted

java.lang.SecurityManager.checkCreateClassLoader could not be converted

java.lang.SecurityManager.checkExec could not be converted

java.lang.SecurityManager.checkExit could not be converted

java.lang.SecurityManager.checkLink could not be converted

java.lang.SecurityManager.checkListen could not be converted

java.lang.SecurityManager.checkMemberAccess could not be converted

java.lang.SecurityManager.checkMulticast could not be converted

java.lang.SecurityManager.checkPackageAccess could not be converted

java.lang.SecurityManager.checkPackageDefinition could not be converted

java.lang.SecurityManager.checkPrintJobAccess could not be converted

java.lang.SecurityManager.checkPropertiesAccess could not be converted

java.lang.SecurityManager.checkPropertyAccess could not be converted

java.lang.SecurityManager.checkRead could not be converted

java.lang.SecurityManager.checkSecurityAccess could not be converted

java.lang.SecurityManager.checkSetFactory could not be converted

java.lang.SecurityManager.checkTopLevelWindow could not be converted

java.lang.SecurityManager.checkWrite could not be converted

java.lang.SecurityManager.classDepth could not be converted

java.lang.SecurityManager.classLoaderDepth could not be converted

java.lang.SecurityManager.currentClassLoader could not be converted

java.lang.SecurityManager.currentLoadedClass could not be converted

java.lang.SecurityManager.getClassContext could not be converted

java.lang.SecurityManager.getInCheck could not be converted

java.lang.SecurityManager.getSecurityContext could not be converted

java.lang.SecurityManager.getThreadGroup could not be converted

java.lang.SecurityManager.inCheck could not be converted

java.lang.SecurityManager.inClass could not be converted

java.lang.SecurityManager.inClassLoader could not be converted

java.lang.SecurityManager.SecurityManager could not be converted

java.lang.StackOverflowError could not be converted

java.lang.String.getBytes could not be converted

java.lang.String.String(byte[], string) could not be converted

java.lang.String.String(byte[]) could not be converted

java.lang.String.String(byte[]), int, int, int) could not be converted

java.lang.String.String(byte[], int, int, string) could not be converted

java.lang.System.currentTimeMillis could not be converted

java.lang.System.getProperties could not be converted

java.lang.System.getProperty could not be converted

java.lang.System.getSecurityManager could not be converted

java.lang.System.load could not be converted

java.lang.System.loadLibrary could not be converted

java.lang.System.runFinalization could not be converted

java.lang.System.runFinalizersOnExit could not be converted

java.lang.System.setProperties could not be converted

java.lang.System.setSecurityManager could not be converted

java.lang.Thread.activeCount could not be converted

java.lang.Thread.checkAccess could not be converted

java.lang.Thread.countStackFrames could not be converted

java.lang.Thread.destroy could not be converted

java.lang.Thread.enumerate could not be converted

java.lang.Thread.getThreadGroup could not be converted

java.lang.Thread.interrupted could not be converted

java.lang.Thread.isInterrupted could not be converted

java.lang.Thread.run could not be converted

java.lang.Thread.Thread could not be converted

java.lang.Thread.yield could not be converted

java.lang.ThreadGroup could not be converted

java.lang.Throwable.fillInStackTrace could not be converted

Java.math Error Messages

java.math.BigDecimal.BigDecimal could not be converted

java.math.BigDecimal.divide could not be converted

java.math.BigDecimal.movePointLeft could not be converted

java.math.BigDecimal.movePointRight could not be converted

java.math.BigDecimal.ROUND_CEILING could not be converted

java.math.BigDecimal.ROUND_DOWN could not be converted

java.math.BigDecimal.ROUND_FLOOR could not be converted

java.math.BigDecimal.ROUND_HALF_DOWN could not be converted

java.math.BigDecimal.ROUND_HALF_EVEN could not be converted

java.math.BigDecimal.ROUND_HALF_UP could not be converted

java.math.BigDecimal.ROUND_UNNECESSARY could not be converted

java.math.BigDecimal.ROUND_UP could not be converted

java.math.BigDecimal.scale could not be converted

java.math.BigDecimal.setScale could not be converted

java.math.BigDecimal.toBigInteger could not be converted

java.math.BigDecimal.valueOf could not be converted

java.math.BigInteger.and could not be converted

java.math.BigInteger.andNot could not be converted

java.math.BigInteger.BigInteger(byte[]) could not be converted

java.math.BigInteger.BigInteger(int, byte[]) could not be converted

java.math.BigInteger.BigInteger(int, int, Random) could not be converted

java.math.BigInteger.BigInteger(int, Random) could not be converted

java.math.BigInteger.BigInteger(string, int) could not be converted

java.math.BigInteger.bitCount could not be converted

java.math.BigInteger.bitLength could not be converted

java.math.BigInteger.clearBit could not be converted

java.math.BigInteger.flipBit could not be converted

java.math.BigInteger.gcd could not be converted

java.math.BigInteger.getLowestSetBit could not be converted

java.math.BigInteger.isProbablePrime could not be converted

java.math.BigInteger.modInverse could not be converted

java.math.BigInteger.modPow could not be converted

java.math.BigInteger.not could not be converted

java.math.BigInteger.or could not be converted

java.math.BigInteger.pow could not be converted

java.math.BigInteger.setBit could not be converted

java.math.BigInteger.shiftLeft could not be converted

java.math.BigInteger.shiftRight could not be converted

java.math.BigInteger.testBit could not be converted

java.math.BigInteger.toByteArray could not be converted

java.math.BigInteger.toString could not be converted

java.math.BigInteger.xor could not be converted

Java.net Error Messages

java.net.ContentHandler could not be converted

java.net.ContentHandlerFactory could not be converted

java.net.DatagramSocket.getLocalAddress could not be converted

java.net.DatagramSocket.getLocalPort could not be converted

java.net.DatagramSocket.getSoTimeout could not be converted

java.net.DatagramSocket.setSoTimeout could not be converted

java.net.DatagramSocketImpl could not be converted

java.net.FileNameMap could not be converted

java.net.HttpURLConnection.getFollowRedirects could not be converted

java.net.HttpURLConnection.getResponseCode could not be converted

java.net.HttpURLConnection.getResponseMessage could not be converted

java.net.HttpURLConnection.HTTP_MOVED_TEMP could not be converted

java.net.HttpURLConnection.responseCode could not be converted

java.net.HttpURLConnection.responseMessage could not be converted

java.net.HttpURLConnection.usingProxy could not be converted

java.net.InetAddress.getAddress could not be converted

java.net.InetAddress.getAllByName could not be converted

java.net.InetAddress.getByName could not be converted

java.net.InetAddress.getLocalHost could not be converted

java.net.InetAddress.isMulticastAddress could not be converted

java.net.MulticastSocket.getInterface could not be converted

java.net.MulticastSocket.getTTL could not be converted

java.net.MulticastSocket.send could not be converted

java.net.MulticastSocket.setInterface could not be converted

java.net.MulticastSocket.setTTL could not be converted

java.net.NoRouteToHostException.NoRouteToHostException could not be converted

java.net.PlainDatagramSocketImpl could not be converted

java.net.PlainSocketImpl could not be converted

java.net.ServerSocket.getSoTimeout could not be converted

java.net.ServerSocket.implAccept could not be converted

java.net.ServerSocket.ServerSocket could not be converted

java.net.ServerSocket.setSocketFactory could not be converted

java.net.ServerSocket.setSoTimeout could not be converted

java.net.Socket.getInetAddress could not be converted

java.net.Socket.getLocalAddress could not be converted

java.net.Socket.getLocalPort could not be converted

java.net.Socket.getPort could not be converted

java.net.Socket.setSocketImplFactory could not be converted

java.net.Socket.Socket could not be converted

java.net.SocketException.SocketException could not be converted

java.net.SocketImpl could not be converted

java.net.SocketImplFactory could not be converted

java.net.SocketInputStream could not be converted

java.net.SocketOptions could not be converted

java.net.SocketOutputStream could not be converted

java.net.UnknownContentHandler could not be converted

java.net.URL.getContent could not be converted

java.net.URL.openStream could not be converted

java.net.URL.setURLStreamHandlerFactory could not be converted

java.net.URL.URL could not be converted

java.net.URL.URL could not be converted

java.net.URLConnection.allowUserInteraction could not be converted

java.net.URLConnection.connect could not be converted

java.net.URLConnection.doInput could not be converted

java.net.URLConnection.doOutput could not be converted

java.net.URLConnection.fileNameMap could not be converted

java.net.URLConnection.getAllowUserInteraction could not be converted

java.net.URLConnection.getContent could not be converted

java.net.URLConnection.getDefaultAllowUserInteraction could not be converted

java.net.URLConnection.getDefaultUseCaches could not be converted

java.net.URLConnection.getDoInput could not be converted

java.net.URLConnection.getDoOutput could not be converted

java.net.URLConnection.getUseCaches could not be converted

java.net.URLConnection.guessContentTypeFromName could not be converted

java.net.URLConnection.guessContentTypeFromStream could not be converted

java.net.URLConnection.setAllowUserInteraction could not be converted

java.net.URLConnection.setContentHandlerFactory could not be converted

java.net.URLConnection.setDefaultAllowUserInteraction could not be converted

java.net.URLConnection.setDefaultRequestProperty could not be converted

java.net.URLConnection.setDefaultUseCaches could not be converted

java.net.URLConnection.setDoInput could not be converted

java.net.URLConnection.setDoOutput could not be converted

java.net.URLConnection.setUseCaches could not be converted

java.net.URLConnection.toString could not be converted

java.net.URLConnection.url could not be converted

java.net.URLConnection.useCaches could not be converted

java.net.URLEncoder could not be converted

java.net.URLStreamHandler could not be converted

java.net.URLStreamHandlerFactory could not be converted

Java.security Error Messages

java.security.acl could not be converted

java.security.Certificate could not be converted

java.security.Identity could not be converted

java.security.IdentityScope could not be converted

java.security.interfaces.DSAKeyPairGenerator could not be converted

java.security.interfaces.DSAParams.getG could not be converted

java.security.interfaces.DSAParams.getP could not be converted

java.security.interfaces.DSAParams.getQ could not be converted

java.security.interfaces.DSAPrivateKey could not be converted

java.security.interfaces.DSAPublicKey could not be converted

java.security.Key.getFormat could not be converted

java.security.KeyPairGenerator could not be converted

java.security.MessageDigest.clone could not be converted

java.security.MessageDigest.getInstance could not be converted

java.security.Provider could not be converted

java.security.SecureRandom.next could not be converted

java.security.Security could not be converted

java.security.Signature.clone could not be converted

java.security.Signature.engineGetParameter could not be converted

java.security.Signature.engineSetParameter could not be converted

java.security.Signature.engineSign could not be converted

java.security.Signature.engineVerify could not be converted

java.security.Signature.getInstance could not be converted

java.security.Signature.getParameter could not be converted

java.security.Signature.setParameter could not be converted

java.security.Signature.sign could not be converted

java.security.Signature.SIGN~ could not be converted

java.security.Signature.state could not be converted

java.security.Signature.UNINITIALIZED could not be converted

java.security.Signature.verify could not be converted

java.security.Signature.VERIFY~ could not be converted

java.security.Signer could not be converted

Java.sql Error Messages

java.sql.CallableStatement.getBigDecimal could not be converted

java.sql.CallableStatement.getTimestamp could not be converted

java.sql.Connection.clearWarnings could not be converted

java.sql.Connection.getWarnings could not be converted

java.sql.Connection.isReadOnly could not be converted

java.sql.Connection.nativeSQL could not be converted

java.sql.Connection.setReadOnly could not be converted

java.sql.DatabaseMetaData.allProceduresAreCallable could not be converted

java.sql.DatabaseMetaData.allTablesAreSelectable could not be converted

java.sql.DatabaseMetaData.bestRowNotPseudo could not be converted

java.sql.DatabaseMetaData.bestRowPseudo could not be converted

java.sql.DatabaseMetaData.bestRowSession could not be converted

java.sql.DatabaseMetaData.bestRowTemporary could not be converted

java.sql.DatabaseMetaData.bestRowTransaction could not be converted

java.sql.DatabaseMetaData.bestRowUnknown could not be converted

java.sql.DatabaseMetaData.columnNoNulls could not be converted

java.sql.DatabaseMetaData.columnNullable could not be converted

java.sql.DatabaseMetaData.columnNullableUnknown could not be converted

java.sql.DatabaseMetaData.dataDefinitionCausesTransactionCommit could not be converted

java.sql.DatabaseMetaData.dataDefinitionIgnoredInTransactions could not be converted

java.sql.DatabaseMetaData.doesMaxRowSizeIncludeBlobs could not be converted

java.sql.DatabaseMetaData.getBestRowIdentifier could not be converted

java.sql.DatabaseMetaData.getCatalogTerm could not be converted

java.sql.DatabaseMetaData.getCrossReference could not be converted

java.sql.DatabaseMetaData.getDatabaseProductName could not be converted

java.sql.DatabaseMetaData.getDriverMajorVersion could not be converted

java.sql.DatabaseMetaData.getDriverMinorVersion could not be converted

java.sql.DatabaseMetaData.getDriverName could not be converted

java.sql.DatabaseMetaData.getDriverVersion could not be converted

java.sql.DatabaseMetaData.getExtraNameCharacters could not be converted

java.sql.DatabaseMetaData.getIdentifierQuoteString could not be converted

java.sql.DatabaseMetaData.getIndexInfo could not be converted

java.sql.DatabaseMetaData.getMaxColumnsInGroupBy could not be converted

java.sql.DatabaseMetaData.getMaxColumnsInIndex could not be converted

java.sql.DatabaseMetaData.getMaxColumnsInOrderBy could not be converted

java.sql.DatabaseMetaData.getMaxColumnsInSelect could not be converted

java.sql.DatabaseMetaData.getMaxColumnsInTable could not be converted

java.sql.DatabaseMetaData.getMaxConnections could not be converted

java.sql.DatabaseMetaData.getMaxIndexLength could not be converted

java.sql.DatabaseMetaData.getMaxRowSize could not be converted

java.sql.DatabaseMetaData.getMaxStatementLength could not be converted

java.sql.DatabaseMetaData.getMaxStatements could not be converted

java.sql.DatabaseMetaData.getMaxTablesInSelect could not be converted

java.sql.DatabaseMetaData.getNumericFunctions could not be converted

java.sql.DatabaseMetaData.getProcedureTerm could not be converted

java.sql.DatabaseMetaData.getSchemaTerm could not be converted

java.sql.DatabaseMetaData.getSearchStringEscape could not be converted

java.sql.DatabaseMetaData.getSQLKeywords could not be converted

java.sql.DatabaseMetaData.getStringFunctions could not be converted

java.sql.DatabaseMetaData.getSystemFunctions could not be converted

java.sql.DatabaseMetaData.getTimeDateFunctions could not be converted

java.sql.DatabaseMetaData.getURL could not be converted

java.sql.DatabaseMetaData.getUserName could not be converted

java.sql.DatabaseMetaData.getVersionColumns could not be converted

java.sql.DatabaseMetaData.importedKeyCascade could not be converted

java.sql.DatabaseMetaData.importedKeyInitiallyDeferred could not be converted

java.sql.DatabaseMetaData.importedKeyInitiallyImmediate could not be converted

java.sql.DatabaseMetaData.importedKeyNoAction could not be converted

java.sql.DatabaseMetaData.importedKeyNotDeferrable could not be converted

java.sql.DatabaseMetaData.importedKeyRestrict could not be converted

java.sql.DatabaseMetaData.importedKeySetDefault could not be converted

java.sql.DatabaseMetaData.importedKeySetNull could not be converted

java.sql.DatabaseMetaData.isCatalogAtStart could not be converted

java.sql.DatabaseMetaData.isReadOnly could not be converted

java.sql.DatabaseMetaData.nullPlusNonNullIsNull could not be converted

java.sql.DatabaseMetaData.nullsAreSortedAtEnd could not be converted

java.sql.DatabaseMetaData.nullsAreSortedAtStart could not be converted

java.sql.DatabaseMetaData.nullsAreSortedHigh could not be converted

java.sql.DatabaseMetaData.nullsAreSortedLow could not be converted

java.sql.DatabaseMetaData.procedureColumnIn could not be converted

java.sql.DatabaseMetaData.procedureColumnInOut could not be converted

java.sql.DatabaseMetaData.procedureColumnOut could not be converted

java.sql.DatabaseMetaData.procedureColumnResult could not be converted

java.sql.DatabaseMetaData.procedureColumnReturn could not be converted

java.sql.DatabaseMetaData.procedureColumnUnknown could not be converted

java.sql.DatabaseMetaData.procedureNoNulls could not be converted

java.sql.DatabaseMetaData.procedureNoResult could not be converted

java.sql.DatabaseMetaData.procedureNullable could not be converted

java.sql.DatabaseMetaData.procedureNullableUnknown could not be converted

java.sql.DatabaseMetaData.procedureResultUnknown could not be converted

java.sql.DatabaseMetaData.procedureReturnsResult could not be converted

java.sql.DatabaseMetaData.storesLowerCaseIdentifiers could not be converted

java.sql.DatabaseMetaData.storesLowerCaseQuotedIdentifiers could not be converted

java.sql.DatabaseMetaData.storesMixedCaseIdentifiers could not be converted

java.sql.DatabaseMetaData.storesMixedCaseQuotedIdentifiers could not be converted

java.sql.DatabaseMetaData.storesUpperCaseIdentifiers could not be converted

java.sql.DatabaseMetaData.storesUpperCaseQuotedIdentifiers could not be converted

java.sql.DatabaseMetaData.supportsAlterTableWithAddColumn could not be converted

java.sql.DatabaseMetaData.supportsAlterTableWithDropColumn could not be converted

java.sql.DatabaseMetaData.supportsANSI92EntryLevelSQL could not be converted

java.sql.DatabaseMetaData.supportsANSI92FullSQL could not be converted

java.sql.DatabaseMetaData.supportsANSI92IntermediateSQL could not be converted

java.sql.DatabaseMetaData.supportsCatalogsInDataManipulation could not be converted

java.sql.DatabaseMetaData.supportsCatalogsInIndexDefinitions could not be converted

java.sql.DatabaseMetaData.supportsCatalogsInPrivilegeDefinitions could not be converted

java.sql.DatabaseMetaData.supportsCatalogsInProcedureCalls could not be converted

java.sql.DatabaseMetaData.supportsCatalogsInTableDefinitions could not be converted

java.sql.DatabaseMetaData.supportsColumnAliasing could not be converted

java.sql.DatabaseMetaData.supportsConvert could not be converted

java.sql.DatabaseMetaData.supportsCoreSQLGrammar could not be converted

java.sql.DatabaseMetaData.supportsCorrelatedSubqueries could not be converted

java.sql.DatabaseMetaData.supportsDataDefinitionAndDataManipulationTransactions could not be converted

java.sql.DatabaseMetaData.supportsDataManipulationTransactionsOnly could not be converted

java.sql.DatabaseMetaData.supportsDifferentTableCorrelationNames could not be converted

java.sql.DatabaseMetaData.supportsExpressionsInOrderBy could not be converted

java.sql.DatabaseMetaData.supportsExtendedSQLGrammar could not be converted

java.sql.DatabaseMetaData.supportsFullOuterJoins could not be converted

java.sql.DatabaseMetaData.supportsGroupBy could not be converted

java.sql.DatabaseMetaData.supportsGroupByBeyondSelect could not be converted

java.sql.DatabaseMetaData.supportsGroupByUnrelated could not be converted

java.sql.DatabaseMetaData.supportsIntegrityEnhancementFacility could not be converted

java.sql.DatabaseMetaData.supportsLikeEscapeClause could not be converted

java.sql.DatabaseMetaData.supportsLimitedOuterJoins could not be converted

java.sql.DatabaseMetaData.supportsMinimumSQLGrammar could not be converted

java.sql.DatabaseMetaData.supportsMixedCaseIdentifiers could not be converted

java.sql.DatabaseMetaData.supportsMixedCaseQuotedIdentifiers could not be converted

java.sql.DatabaseMetaData.supportsMultipleResultSets could not be converted

java.sql.DatabaseMetaData.supportsMultipleTransactions could not be converted

java.sql.DatabaseMetaData.supportsNonNullableColumns could not be converted

java.sql.DatabaseMetaData.supportsOpenCursorsAcrossCommit could not be converted

java.sql.DatabaseMetaData.supportsOpenCursorsAcrossRollback could not be converted

java.sql.DatabaseMetaData.supportsOpenStatementsAcrossCommit could not be converted

java.sql.DatabaseMetaData.supportsOpenStatementsAcrossRollback could not be converted

java.sql.DatabaseMetaData.supportsOrderByUnrelated could not be converted

java.sql.DatabaseMetaData.supportsOuterJoins could not be converted

java.sql.DatabaseMetaData.supportsPositionedDelete could not be converted

java.sql.DatabaseMetaData.supportsPositionedUpdate could not be converted

java.sql.DatabaseMetaData.supportsSchemasInDataManipulation could not be converted

java.sql.DatabaseMetaData.supportsSchemasInIndexDefinitions could not be converted

java.sql.DatabaseMetaData.supportsSchemasInPrivilegeDefinitions could not be converted

java.sql.DatabaseMetaData.supportsSchemasInProcedureCalls could not be converted

java.sql.DatabaseMetaData.supportsSchemasInTableDefinitions could not be converted

java.sql.DatabaseMetaData.supportsSelectForUpdate could not be converted

java.sql.DatabaseMetaData.supportsStoredProcedures could not be converted

java.sql.DatabaseMetaData.supportsSubqueriesInComparisons could not be converted

java.sql.DatabaseMetaData.supportsSubqueriesInExists could not be converted

java.sql.DatabaseMetaData.supportsSubqueriesInIns could not be converted

java.sql.DatabaseMetaData.supportsSubqueriesInQuantifieds could not be converted

java.sql.DatabaseMetaData.supportsTableCorrelationNames could not be converted

java.sql.DatabaseMetaData.supportsTransactionIsolationLevel could not be converted

java.sql.DatabaseMetaData.supportsTransactions could not be converted

java.sql.DatabaseMetaData.supportsUnion could not be converted

java.sql.DatabaseMetaData.supportsUnionAll could not be converted

java.sql.DatabaseMetaData.tableIndexClustered could not be converted

java.sql.DatabaseMetaData.tableIndexHashed could not be converted

java.sql.DatabaseMetaData.tableIndexOther could not be converted

java.sql.DatabaseMetaData.tableIndexStatistic could not be converted

java.sql.DatabaseMetaData.typeNoNulls could not be converted

java.sql.DatabaseMetaData.typeNullable could not be converted

java.sql.DatabaseMetaData.typeNullableUnknown could not be converted

java.sql.DatabaseMetaData.typePredBasic could not be converted

java.sql.DatabaseMetaData.typePredChar could not be converted

java.sql.DatabaseMetaData.typePredNone could not be converted

java.sql.DatabaseMetaData.typeSearchable could not be converted

java.sql.DatabaseMetaData.usesLocalFilePerTable could not be converted

java.sql.DatabaseMetaData.usesLocalFiles could not be converted

java.sql.DatabaseMetaData.versionColumnNotPseudo could not be converted

java.sql.DatabaseMetaData.versionColumnPseudo could not be converted

java.sql.DatabaseMetaData.versionColumnUnknown could not be converted

java.sql.DataTruncation could not be converted

java.sql.Date.getHours could not be converted

java.sql.Date.getMinutes could not be converted

java.sql.Date.getSeconds could not be converted

java.sql.Date.setHours could not be converted

java.sql.Date.setMinutes could not be converted

java.sql.Date.setSeconds could not be converted

java.sql.DriverInfo could not be converted

java.sql.DriverManager could not be converted

java.sql.DriverPropertyInfo could not be converted

java.sql.ResultSet could not be converted

java.sql.ResultSet.clearWarnings could not be converted

java.sql.ResultSet.getAsciiStream could not be converted

java.sql.ResultSet.getBigDecimal could not be converted

java.sql.ResultSet.getBinaryStream could not be converted

java.sql.ResultSet.getBytes could not be converted

java.sql.ResultSet.getCursorName could not be converted

java.sql.ResultSetMetaData.getPrecision could not be converted

java.sql.ResultSet.getUnicodeStream could not be converted

java.sql.ResultSet.getWarnings could not be converted

java.sql.ResultSet.wasNull could not be converted

java.sql.ResultSetMetaData.columnNoNulls could not be converted

java.sql.ResultSetMetaData.columnNullable could not be converted

java.sql.ResultSetMetaData.columnNullableUnknown could not be converted

java.sql.ResultSetMetaData.isCaseSensitive could not be converted

java.sql.ResultSetMetaData.isCurrency could not be converted

java.sql.ResultSetMetaData.isDefinitelyWritable could not be converted

java.sql.ResultSetMetaData.isSearchable could not be converted

java.sql.ResultSetMetaData.isSigned could not be converted

java.sql.ResultSetMetaData.isWritable could not be converted

java.sql.SQLException.getNextException could not be converted

java.sql.SQLException.setNextException could not be converted

java.sql.SQLException.SQLException could not be converted

java.sql.SQLWarning could not be converted

java.sql.Statement.clearWarnings could not be converted

java.sql.Statement.close could not be converted

java.sql.Statement.getMaxFieldSize could not be converted

java.sql.Statement.getMaxRows could not be converted

java.sql.Statement.getMoreResults could not be converted

java.sql.Statement.getResultSet could not be converted

java.sql.Statement.getUpdateCount could not be converted

java.sql.Statement.getWarnings could not be converted

java.sql.Statement.setCursorName could not be converted

java.sql.Statement.setEscapeProcessing could not be converted

java.sql.Statement.setMaxFieldSize could not be converted

java.sql.Statement.setMaxRows could not be converted

java.sql.Timestamp.getNanos could not be converted

java.sql.Timestamp.setNanos could not be converted

Java.text Error Messages

java.text.BreakIterator could not be converted

java.text.CharacterIterator.clone could not be converted

java.text.ChoiceFormat could not be converted

java.text.CollationElementIterator could not be converted

java.text.Collator.CANONICAL_DECOMPOSITION could not be converted

java.text.Collator.Collator could not be converted

java.text.Collator.compare could not be converted

java.text.Collator.FULL_DECOMPOSITION could not be converted

java.text.Collator.getDecomposition could not be converted

java.text.Collator.getStrength could not be converted

java.text.Collator.IDENTICAL could not be converted

java.text.Collator.NO_DECOMPOSITION could not be converted

java.text.Collator.PRIMARY could not be converted

java.text.Collator.SECONDARY could not be converted

java.text.Collator.setDecomposition could not be converted

java.text.Collator.setStrength could not be converted

java.text.Collator.TERTIARY could not be converted

java.text.DateFormat.format could not be converted

java.text.DateFormat.getNumberFormat could not be converted

java.text.DateFormat.getTimeZone could not be converted

java.text.DateFormat.isLenient could not be converted

java.text.DateFormat.numberFormat could not be converted

java.text.DateFormat.parse could not be converted

java.text.DateFormat.parseObject could not be converted

java.text.DateFormat.setLenient could not be converted

java.text.DateFormat.setNumberFormat could not be converted

java.text.DateFormat.setTimeZone could not be converted

java.text.DateFormatSymbols.getEras could not be converted

java.text.DateFormatSymbols.getLocalPatternChars could not be converted

java.text.DateFormatSymbols.getZoneStrings could not be converted

java.text.DateFormatSymbols.setEras could not be converted

java.text.DateFormatSymbols.setLocalPatternChars could not be converted

java.text.DateFormatSymbols.setZoneStrings could not be converted

java.text.DecimalFormat could not be converted

java.text.DecimalFormatSymbols.DecimalFormatSymbols could not be converted

java.text.DecimalFormatSymbols.getDigit could not be converted

java.text.DecimalFormatSymbols.getPatternSeparator could not be converted

java.text.DecimalFormatSymbols.getZeroDigit could not be converted

java.text.DecimalFormatSymbols.setDigit could not be converted

java.text.DecimalFormatSymbols.setPatternSeparator could not be converted

java.text.DecimalFormatSymbols.setZeroDigit could not be converted

java.text.Format could not be converted

java.text.MessageFormat could not be converted

java.text.NumberFormat.format could not be converted

java.text.NumberFormat.FRACTION_FIELD could not be converted

java.text.NumberFormat.INTEGER_FIELD could not be converted

java.text.NumberFormat.isParseIntegerOnly could not be converted

java.text.NumberFormat.parse could not be converted

java.text.NumberFormat.parseObject could not be converted

java.text.NumberFormat.setParseIntegerOnly could not be converted

java.text.ParseException.getErrorOffset could not be converted

java.text.RuleBasedCollator could not be converted

java.text.SimpleDateFormat could not be converted

java.text.StringCharacterIterator.clone could not be converted

java.text.StringCharacterIterator.setText could not be converted

Java.text.resources Error Messages

java.text.resources.DateFormatZoneData.DateFormatZoneData could not be converted

java.text.resources.DateFormatZoneData.getContents could not be converted

java.text.resources.DateFormatZoneData_ar.DateFormatZoneData_ar could not be converted

java.text.resources.DateFormatZoneData_ar.getContents could not be converted

java.text.resources.DateFormatZoneData_be.DateFormatZoneData_be could not be converted

java.text.resources.DateFormatZoneData_be.getContents could not be converted

java.text.resources.DateFormatZoneData_bg.DateFormatZoneData_bg could not be converted

java.text.resources.DateFormatZoneData_bg.getContents could not be converted

java.text.resources.DateFormatZoneData_ca.DateFormatZoneData_ca could not be converted

java.text.resources.DateFormatZoneData_ca.getContents could not be converted

java.text.resources.DateFormatZoneData_cs.DateFormatZoneData_cs could not be converted

java.text.resources.DateFormatZoneData_cs.getContents could not be converted

java.text.resources.DateFormatZoneData_da.DateFormatZoneData_da could not be converted

java.text.resources.DateFormatZoneData_da.getContents could not be converted

java.text.resources.DateFormatZoneData_de.DateFormatZoneData_de could not be converted

java.text.resources.DateFormatZoneData_de.getContents could not be converted

java.text.resources.DateFormatZoneData_de_AT.DateFormatZoneData_de_AT could not be converted

java.text.resources.DateFormatZoneData_de_AT.getContents could not be converted

java.text.resources.DateFormatZoneData_de_CH.DateFormatZoneData_de_CH could not be converted

java.text.resources.DateFormatZoneData_de_CH.getContents could not be converted

java.text.resources.DateFormatZoneData_el.DateFormatZoneData_el could not be converted

java.text.resources.DateFormatZoneData_el.getContents could not be converted

java.text.resources.DateFormatZoneData_en.DateFormatZoneData_en could not be converted

java.text.resources.DateFormatZoneData_en.getContents could not be converted

java.text.resources.DateFormatZoneData_en_CA.DateFormatZoneData_en_CA could not be converted

java.text.resources.DateFormatZoneData_en_CA.getContents could not be converted

java.text.resources.DateFormatZoneData_en_GB.DateFormatZoneData_en_GB could not be converted

java.text.resources.DateFormatZoneData_en_GB.getContents could not be converted

java.text.resources.DateFormatZoneData_en_IE.DateFormatZoneData_en_IE could not be converted

java.text.resources.DateFormatZoneData_en_IE.getContents could not be converted

java.text.resources.DateFormatZoneData_es.DateFormatZoneData_es could not be converted

java.text.resources.DateFormatZoneData_es.getContents could not be converted

java.text.resources.DateFormatZoneData_et.DateFormatZoneData_et could not be converted

java.text.resources.DateFormatZoneData_et.getContents could not be converted

java.text.resources.DateFormatZoneData_fi.DateFormatZoneData_fi could not be converted

java.text.resources.DateFormatZoneData_fi.getContents could not be converted

java.text.resources.DateFormatZoneData_fr.DateFormatZoneData_fr could not be converted

java.text.resources.DateFormatZoneData_fr.getContents could not be converted

java.text.resources.DateFormatZoneData_fr_BE.DateFormatZoneData_fr_BE could not be converted

java.text.resources.DateFormatZoneData_fr_BE.getContents could not be converted

java.text.resources.DateFormatZoneData_fr_CA.DateFormatZoneData_fr_CA could not be converted

java.text.resources.DateFormatZoneData_fr_CA.getContents could not be converted

java.text.resources.DateFormatZoneData_fr_CH.DateFormatZoneData_fr_CH could not be converted

java.text.resources.DateFormatZoneData_fr_CH.getContents could not be converted

java.text.resources.DateFormatZoneData_hr.DateFormatZoneData_hr could not be converted

java.text.resources.DateFormatZoneData_hr.getContents could not be converted

java.text.resources.DateFormatZoneData_hu.DateFormatZoneData_hu could not be converted

java.text.resources.DateFormatZoneData_hu.getContents could not be converted

java.text.resources.DateFormatZoneData_is.DateFormatZoneData_is could not be converted

java.text.resources.DateFormatZoneData_is.getContents could not be converted

java.text.resources.DateFormatZoneData_it.DateFormatZoneData_it could not be converted

java.text.resources.DateFormatZoneData_it.getContents could not be converted

java.text.resources.DateFormatZoneData_it_CH.DateFormatZoneData_it_CH could not be converted

java.text.resources.DateFormatZoneData_it_CH.getContents could not be converted

java.text.resources.DateFormatZoneData_iw.DateFormatZoneData_iw could not be converted

java.text.resources.DateFormatZoneData_iw.getContents could not be converted

java.text.resources.DateFormatZoneData_ja.DateFormatZoneData_ja could not be converted

java.text.resources.DateFormatZoneData_ja.getContents could not be converted

java.text.resources.DateFormatZoneData_ko.DateFormatZoneData_ko could not be converted

java.text.resources.DateFormatZoneData_ko.getContents could not be converted

java.text.resources.DateFormatZoneData_lt.DateFormatZoneData_lt could not be converted

java.text.resources.DateFormatZoneData_lt.getContents could not be converted

java.text.resources.DateFormatZoneData_lv.DateFormatZoneData_lv could not be converted

java.text.resources.DateFormatZoneData_lv.getContents could not be converted

java.text.resources.DateFormatZoneData_mk.DateFormatZoneData_mk could not be converted

java.text.resources.DateFormatZoneData_mk.getContents could not be converted

java.text.resources.DateFormatZoneData_nl.DateFormatZoneData_nl could not be converted

java.text.resources.DateFormatZoneData_nl.getContents could not be converted

java.text.resources.DateFormatZoneData_nl_BE.DateFormatZoneData_nl_BE could not be converted

java.text.resources.DateFormatZoneData_nl_BE.getContents could not be converted

java.text.resources.DateFormatZoneData_no.DateFormatZoneData_no could not be converted

java.text.resources.DateFormatZoneData_no.getContents could not be converted

java.text.resources.DateFormatZoneData_no_NO_NY.DateFormatZoneData_no_NO_NY could not be converted

java.text.resources.DateFormatZoneData_no_NO_NY.getContents could not be converted

java.text.resources.DateFormatZoneData_pl.DateFormatZoneData_pl could not be converted

java.text.resources.DateFormatZoneData_pl.getContents could not be converted

java.text.resources.DateFormatZoneData_pt.DateFormatZoneData_pt could not be converted

java.text.resources.DateFormatZoneData_pt.getContents could not be converted

java.text.resources.DateFormatZoneData_ro.DateFormatZoneData_ro could not be converted

java.text.resources.DateFormatZoneData_ro.getContents could not be converted

java.text.resources.DateFormatZoneData_ru.DateFormatZoneData_ru could not be converted

java.text.resources.DateFormatZoneData_ru.getContents could not be converted

java.text.resources.DateFormatZoneData_sh.DateFormatZoneData_sh could not be converted

java.text.resources.DateFormatZoneData_sh.getContents could not be converted

java.text.resources.DateFormatZoneData_sk.DateFormatZoneData_sk could not be converted

java.text.resources.DateFormatZoneData_sk.getContents could not be converted

java.text.resources.DateFormatZoneData_sl.DateFormatZoneData_sl could not be converted

java.text.resources.DateFormatZoneData_sl.getContents could not be converted

java.text.resources.DateFormatZoneData_sq.DateFormatZoneData_sq could not be converted

java.text.resources.DateFormatZoneData_sq.getContents could not be converted

java.text.resources.DateFormatZoneData_sr.DateFormatZoneData_sr could not be converted

java.text.resources.DateFormatZoneData_sr.getContents could not be converted

java.text.resources.DateFormatZoneData_sv.DateFormatZoneData_sv could not be converted

java.text.resources.DateFormatZoneData_sv.getContents could not be converted

java.text.resources.DateFormatZoneData_th.getContents could not be converted

java.text.resources.DateFormatZoneData_tr could not be converted

java.text.resources.DateFormatZoneData_tr.DateFormatZoneData_tr could not be converted

java.text.resources.DateFormatZoneData_tr.getContents could not be converted

java.text.resources.DateFormatZoneData_uk.DateFormatZoneData_uk could not be converted

java.text.resources.DateFormatZoneData_uk.getContents could not be converted

java.text.resources.DateFormatZoneData_zh.DateFormatZoneData_zh could not be converted

java.text.resources.DateFormatZoneData_zh.getContents could not be converted

java.text.resources.DateFormatZoneData_zh_HK.getContents could not be converted

java.text.resources.DateFormatZoneData_zh_TW.DateFormatZoneData_zh_TW could not be converted

java.text.resources.DateFormatZoneData_zh_TW.getContents could not be converted

java.text.resources.LocaleData.getKeys could not be converted

java.text.resources.LocaleData.init could not be converted

java.text.resources.LocaleData.LocaleData could not be converted

java.text.resources.LocaleElements.LocaleElements could not be converted

java.text.resources.LocaleElements_ar.LocaleElements_ar could not be converted

java.text.resources.LocaleElements_ar_AE.getContents could not be converted

java.text.resources.LocaleElements_ar_AE.LocaleElements_ar_AE could not be converted

java.text.resources.LocaleElements_ar_BH.getContents could not be converted

java.text.resources.LocaleElements_ar_BH.LocaleElements_ar_BH could not be converted

java.text.resources.LocaleElements_ar_DZ.getContents could not be converted

java.text.resources.LocaleElements_ar_DZ.LocaleElements_ar_DZ could not be converted

java.text.resources.LocaleElements_ar_EG.getContents could not be converted

java.text.resources.LocaleElements_ar_EG.LocaleElements_ar_EG could not be converted

java.text.resources.LocaleElements_ar_IQ.getContents could not be converted

java.text.resources.LocaleElements_ar_IQ.LocaleElements_ar_IQ could not be converted

java.text.resources.LocaleElements_ar_JO.getContents could not be converted

java.text.resources.LocaleElements_ar_JO.LocaleElements_ar_JO could not be converted

java.text.resources.LocaleElements_ar_KW.getContents could not be converted

java.text.resources.LocaleElements_ar_KW.LocaleElements_ar_KW could not be converted

java.text.resources.LocaleElements_ar_LB.getContents could not be converted

java.text.resources.LocaleElements_ar_LB.LocaleElements_ar_LB could not be converted

java.text.resources.LocaleElements_ar_LY.getContents could not be converted

java.text.resources.LocaleElements_ar_LY.LocaleElements_ar_LY could not be converted

java.text.resources.LocaleElements_ar_MA.getContents could not be converted

java.text.resources.LocaleElements_ar_MA.LocaleElements_ar_MA could not be converted

java.text.resources.LocaleElements_ar_OM.getContents could not be converted

java.text.resources.LocaleElements_ar_OM.LocaleElements_ar_OM could not be converted

java.text.resources.LocaleElements_ar_QA.getContents could not be converted

java.text.resources.LocaleElements_ar_QA.LocaleElements_ar_QA could not be converted

java.text.resources.LocaleElements_ar_SA.getContents could not be converted

java.text.resources.LocaleElements_ar_SA.LocaleElements_ar_SA could not be converted

java.text.resources.LocaleElements_ar_SD.getContents could not be converted

java.text.resources.LocaleElements_ar_SD.LocaleElements_ar_SD could not be converted

java.text.resources.LocaleElements_ar_SY.getContents could not be converted

java.text.resources.LocaleElements_ar_SY.LocaleElements_ar_SY could not be converted

java.text.resources.LocaleElements_ar_TN.getContents could not be converted

java.text.resources.LocaleElements_ar_TN.LocaleElements_ar_TN could not be converted

java.text.resources.LocaleElements_ar_YE.getContents could not be converted

java.text.resources.LocaleElements_ar_YE.LocaleElements_ar_YE could not be converted

java.text.resources.LocaleElements_be.LocaleElements_be could not be converted

java.text.resources.LocaleElements_be_BY.getContents could not be converted

java.text.resources.LocaleElements_be_BY.LocaleElements_be_BY could not be converted

java.text.resources.LocaleElements_bg.LocaleElements_bg could not be converted

java.text.resources.LocaleElements_bg_BG.getContents could not be converted

java.text.resources.LocaleElements_bg_BG.LocaleElements_bg_BG could not be converted

java.text.resources.LocaleElements_ca.LocaleElements_ca could not be converted

java.text.resources.LocaleElements_ca_ES.getContents could not be converted

java.text.resources.LocaleElements_ca_ES.LocaleElements_ca_ES could not be converted

java.text.resources.LocaleElements_ca_ES_EURO.getContents could not be converted

java.text.resources.LocaleElements_ca_ES_EURO.LocaleElements_ca_ES_EURO could not be converted

java.text.resources.LocaleElements_cs.LocaleElements_cs could not be converted

java.text.resources.LocaleElements_cs_CZ.getContents could not be converted

java.text.resources.LocaleElements_cs_CZ.LocaleElements_cs_CZ could not be converted

java.text.resources.LocaleElements_da.LocaleElements_da could not be converted

java.text.resources.LocaleElements_da_DK.getContents could not be converted

java.text.resources.LocaleElements_da_DK.LocaleElements_da_DK could not be converted

java.text.resources.LocaleElements_de.LocaleElements_de could not be converted

java.text.resources.LocaleElements_de_AT.LocaleElements_de_AT could not be converted

java.text.resources.LocaleElements_de_AT_EURO.getContents could not be converted

java.text.resources.LocaleElements_de_AT_EURO.LocaleElements_de_AT_EURO could not be converted

java.text.resources.LocaleElements_de_CH.LocaleElements_de_CH could not be converted

java.text.resources.LocaleElements_de_DE.getContents could not be converted

java.text.resources.LocaleElements_de_DE.LocaleElements_de_DE could not be converted

java.text.resources.LocaleElements_de_DE_EURO.getContents could not be converted

java.text.resources.LocaleElements_de_DE_EURO.LocaleElements_de_DE_EURO could not be converted

java.text.resources.LocaleElements_de_LU.getContents could not be converted

java.text.resources.LocaleElements_de_LU.LocaleElements_de_LU could not be converted

java.text.resources.LocaleElements_de_LU_EURO.getContents could not be converted

java.text.resources.LocaleElements_de_LU_EURO.LocaleElements_de_LU_EURO could not be converted

java.text.resources.LocaleElements_el.LocaleElements_el could not be converted

java.text.resources.LocaleElements_el_GR.getContents could not be converted

java.text.resources.LocaleElements_el_GR.LocaleElements_el_GR could not be converted

java.text.resources.LocaleElements_en.LocaleElements_en could not be converted

java.text.resources.LocaleElements_en_AU.LocaleElements_en_AU could not be converted

java.text.resources.LocaleElements_en_CA.LocaleElements_en_CA could not be converted

java.text.resources.LocaleElements_en_GB.LocaleElements_en_GB could not be converted

java.text.resources.LocaleElements_en_IE.LocaleElements_en_IE could not be converted

java.text.resources.LocaleElements_en_IE_EURO.getContents could not be converted

java.text.resources.LocaleElements_en_IE_EURO.LocaleElements_en_IE_EURO could not be converted

java.text.resources.LocaleElements_en_IR.LocaleElements_en_IR could not be converted

java.text.resources.LocaleElements_en_NZ.LocaleElements_en_NZ could not be converted

java.text.resources.LocaleElements_en_US.getContents could not be converted

java.text.resources.LocaleElements_en_US.LocaleElements_en_US could not be converted

java.text.resources.LocaleElements_en_ZA.LocaleElements_en_ZA could not be converted

java.text.resources.LocaleElements_es.LocaleElements_es could not be converted

java.text.resources.LocaleElements_es_AR.getContents could not be converted

java.text.resources.LocaleElements_es_AR.LocaleElements_es_AR could not be converted

java.text.resources.LocaleElements_es_BO.getContents could not be converted

java.text.resources.LocaleElements_es_BO.LocaleElements_es_BO could not be converted

java.text.resources.LocaleElements_es_CL.getContents could not be converted

java.text.resources.LocaleElements_es_CL.LocaleElements_es_CL could not be converted

java.text.resources.LocaleElements_es_CO.getContents could not be converted

java.text.resources.LocaleElements_es_CO.LocaleElements_es_CO could not be converted

java.text.resources.LocaleElements_es_CR.getContents could not be converted

java.text.resources.LocaleElements_es_CR.LocaleElements_es_CR could not be converted

java.text.resources.LocaleElements_es_DO.getContents could not be converted

java.text.resources.LocaleElements_es_DO.LocaleElements_es_DO could not be converted

java.text.resources.LocaleElements_es_EC.getContents could not be converted

java.text.resources.LocaleElements_es_EC.LocaleElements_es_EC could not be converted

java.text.resources.LocaleElements_es_ES.getContents could not be converted

java.text.resources.LocaleElements_es_ES.LocaleElements_es_ES could not be converted

java.text.resources.LocaleElements_es_ES_EURO.getContents could not be converted

java.text.resources.LocaleElements_es_ES_EURO.LocaleElements_es_ES_EURO could not be converted

java.text.resources.LocaleElements_es_GT.getContents could not be converted

java.text.resources.LocaleElements_es_GT.LocaleElements_es_GT could not be converted

java.text.resources.LocaleElements_es_HN.getContents could not be converted

java.text.resources.LocaleElements_es_HN.LocaleElements_es_HN could not be converted

java.text.resources.LocaleElements_es_MX.LocaleElements_es_MX could not be converted

java.text.resources.LocaleElements_es_NI.getContents could not be converted

java.text.resources.LocaleElements_es_NI.LocaleElements_es_NI could not be converted

java.text.resources.LocaleElements_es_PA.getContents could not be converted

java.text.resources.LocaleElements_es_PA.LocaleElements_es_PA could not be converted

java.text.resources.LocaleElements_es_PE.getContents could not be converted

java.text.resources.LocaleElements_es_PE.LocaleElements_es_PE could not be converted

java.text.resources.LocaleElements_es_PR.getContents could not be converted

java.text.resources.LocaleElements_es_PR.LocaleElements_es_PR could not be converted

java.text.resources.LocaleElements_es_PY.getContents could not be converted

java.text.resources.LocaleElements_es_SV.getContents could not be converted

java.text.resources.LocaleElements_es_SV.LocaleElements_es_SV could not be converted

java.text.resources.LocaleElements_es_UY.getContents could not be converted

java.text.resources.LocaleElements_es_UY.LocaleElements_es_UY could not be converted

java.text.resources.LocaleElements_es_VE.getContents could not be converted

java.text.resources.LocaleElements_es_VE.LocaleElements_es_VE could not be converted

java.text.resources.LocaleElements_et.LocaleElements_et could not be converted

java.text.resources.LocaleElements_et_EE.getContents could not be converted

java.text.resources.LocaleElements_et_EE.LocaleElements_et_EE could not be converted

java.text.resources.LocaleElements_fi.LocaleElements_fi could not be converted

java.text.resources.LocaleElements_fi_FI.getContents could not be converted

java.text.resources.LocaleElements_fi_FI.LocaleElements_fi_FI could not be converted

java.text.resources.LocaleElements_fi_FI_EURO.getContents could not be converted

java.text.resources.LocaleElements_fi_FI_EURO.LocaleElements_fi_FI_EURO could not be converted

java.text.resources.LocaleElements_fr.LocaleElements_fr could not be converted

java.text.resources.LocaleElements_fr_BE.LocaleElements_fr_BE could not be converted

java.text.resources.LocaleElements_fr_BE_EURO.getContents could not be converted

java.text.resources.LocaleElements_fr_BE_EURO.LocaleElements_fr_BE_EURO could not be converted

java.text.resources.LocaleElements_fr_CA.LocaleElements_fr_CA could not be converted

java.text.resources.LocaleElements_fr_CH.LocaleElements_fr_CH could not be converted

java.text.resources.LocaleElements_fr_FR.getContents could not be converted

java.text.resources.LocaleElements_fr_FR.LocaleElements_fr_FR could not be converted

java.text.resources.LocaleElements_fr_FR_EURO.getContents could not be converted

java.text.resources.LocaleElements_fr_FR_EURO.LocaleElements_fr_FR_EURO could not be converted

java.text.resources.LocaleElements_fr_LU.getContents could not be converted

java.text.resources.LocaleElements_fr_LU.LocaleElements_fr_LU could not be converted

java.text.resources.LocaleElements_fr_LU_EURO.getContents could not be converted

java.text.resources.LocaleElements_fr_LU_EURO.LocaleElements_fr_LU_EURO could not be converted

java.text.resources.LocaleElements_he.LocaleElements_he could not be converted

java.text.resources.LocaleElements_hr.LocaleElements_hr could not be converted

java.text.resources.LocaleElements_hr_HR.getContents could not be converted

java.text.resources.LocaleElements_hr_HR.LocaleElements_hr_HR could not be converted

java.text.resources.LocaleElements_hu.LocaleElements_hu could not be converted

java.text.resources.LocaleElements_hu_HU.getContents could not be converted

java.text.resources.LocaleElements_hu_HU.LocaleElements_hu_HU could not be converted

java.text.resources.LocaleElements_is.LocaleElements_is could not be converted

java.text.resources.LocaleElements_is_IS.getContents could not be converted

java.text.resources.LocaleElements_is_IS.LocaleElements_is_IS could not be converted

java.text.resources.LocaleElements_it.LocaleElements_it could not be converted

java.text.resources.LocaleElements_it_CH.LocaleElements_it_CH could not be converted

java.text.resources.LocaleElements_it_IT.LocaleElements_it_IT could not be converted

java.text.resources.LocaleElements_it_IT_EURO.getContents could not be converted

java.text.resources.LocaleElements_it_IT_EURO.LocaleElements_it_IT_EURO could not be converted

java.text.resources.LocaleElements_iw.LocaleElements_iw could not be converted

java.text.resources.LocaleElements_iw_IL.getContents could not be converted

java.text.resources.LocaleElements_iw_IL.LocaleElements_iw_IL could not be converted

java.text.resources.LocaleElements_ja.LocaleElements_ja could not be converted

java.text.resources.LocaleElements_ja_JP.getContents could not be converted

java.text.resources.LocaleElements_ja_JP.LocaleElements_ja_JP could not be converted

java.text.resources.LocaleElements_ko.LocaleElements_ko could not be converted

java.text.resources.LocaleElements_ko_KR.getContents could not be converted

java.text.resources.LocaleElements_ko_KR.LocaleElements_ko_KR could not be converted

java.text.resources.LocaleElements_lt.LocaleElements_lt could not be converted

java.text.resources.LocaleElements_lt_LT.getContents could not be converted

java.text.resources.LocaleElements_lt_LT.LocaleElements_lt_LT could not be converted

java.text.resources.LocaleElements_lv.LocaleElements_lv could not be converted

java.text.resources.LocaleElements_lv_LV.getContents could not be converted

java.text.resources.LocaleElements_lv_LV.LocaleElements_lv_LV could not be converted

java.text.resources.LocaleElements_mk.LocaleElements_mk could not be converted

java.text.resources.LocaleElements_mk_MK.getContents could not be converted

java.text.resources.LocaleElements_mk_MK.LocaleElements_mk_MK could not be converted

java.text.resources.LocaleElements_nl.LocaleElements_nl could not be converted

java.text.resources.LocaleElements_nl_BE.LocaleElements_nl_BE could not be converted

java.text.resources.LocaleElements_nl_BE_EURO.getContents could not be converted

java.text.resources.LocaleElements_nl_BE_EURO.LocaleElements_nl_BE_EURO could not be converted

java.text.resources.LocaleElements_nl_NL.getContents could not be converted

java.text.resources.LocaleElements_nl_NL.LocaleElements_nl_NL could not be converted

java.text.resources.LocaleElements_nl_NL_EURO.getContents could not be converted

java.text.resources.LocaleElements_nl_NL_EURO.LocaleElements_nl_NL_EURO could not be converted

java.text.resources.LocaleElements_no.LocaleElements_no could not be converted

java.text.resources.LocaleElements_no_NO.getContents could not be converted

java.text.resources.LocaleElements_no_NO.LocaleElements_no_NO could not be converted

java.text.resources.LocaleElements_no_NO_NY.LocaleElements_no_NO_NY could not be converted

java.text.resources.LocaleElements_pl.LocaleElements_pl could not be converted

java.text.resources.LocaleElements_pl_PL.getContents could not be converted

java.text.resources.LocaleElements_pl_PL.LocaleElements_pl_PL could not be converted

java.text.resources.LocaleElements_pt.LocaleElements_pt could not be converted

java.text.resources.LocaleElements_pt_BR.getContents could not be converted

java.text.resources.LocaleElements_pt_BR.LocaleElements_pt_BR could not be converted

java.text.resources.LocaleElements_pt_BZ.LocaleElements_pt_BZ could not be converted

java.text.resources.LocaleElements_pt_PT.getContents could not be converted

java.text.resources.LocaleElements_pt_PT.LocaleElements_pt_PT could not be converted

java.text.resources.LocaleElements_pt_PT_EURO.getContents could not be converted

java.text.resources.LocaleElements_pt_PT_EURO.LocaleElements_pt_PT_EURO could not be converted

java.text.resources.LocaleElements_ro.LocaleElements_ro could not be converted

java.text.resources.LocaleElements_ro_RO.getContents could not be converted

java.text.resources.LocaleElements_ro_RO.LocaleElements_ro_RO could not be converted

java.text.resources.LocaleElements_ru.LocaleElements_ru could not be converted

java.text.resources.LocaleElements_ru_RU.getContents could not be converted

java.text.resources.LocaleElements_ru_RU.LocaleElements_ru_RU could not be converted

java.text.resources.LocaleElements_sh.LocaleElements_sh could not be converted

java.text.resources.LocaleElements_sh_YU.getContents could not be converted

java.text.resources.LocaleElements_sh_YU.LocaleElements_sh_YU could not be converted

java.text.resources.LocaleElements_sk.LocaleElements_sk could not be converted

java.text.resources.LocaleElements_sk_SK.getContents could not be converted

java.text.resources.LocaleElements_sk_SK.LocaleElements_sk_SK could not be converted

java.text.resources.LocaleElements_sl.LocaleElements_sl could not be converted

java.text.resources.LocaleElements_sl_SI.getContents could not be converted

java.text.resources.LocaleElements_sl_SI.LocaleElements_sl_SI could not be converted

java.text.resources.LocaleElements_sq.LocaleElements_sq could not be converted

java.text.resources.LocaleElements_sq_AL.getContents could not be converted

java.text.resources.LocaleElements_sq_AL.LocaleElements_sq_AL could not be converted

java.text.resources.LocaleElements_sr.LocaleElements_sr could not be converted

java.text.resources.LocaleElements_sr_YU.getContents could not be converted

java.text.resources.LocaleElements_sr_YU.LocaleElements_sr_YU could not be converted

java.text.resources.LocaleElements_sv.LocaleElements_sv could not be converted

java.text.resources.LocaleElements_sv_SE.getContents could not be converted

java.text.resources.LocaleElements_sv_SE.LocaleElements_sv_SE could not be converted

java.text.resources.LocaleElements_th.getContents could not be converted

java.text.resources.LocaleElements_th.LocaleElements_th could not be converted

java.text.resources.LocaleElements_th_TH.getContents could not be converted

java.text.resources.LocaleElements_th_TH.LocaleElements_th_TH could not be converted

java.text.resources.LocaleElements_tr.LocaleElements_tr could not be converted

java.text.resources.LocaleElements_tr_TR.getContents could not be converted

java.text.resources.LocaleElements_tr_TR.LocaleElements_tr_TR could not be converted

java.text.resources.LocaleElements_uk.LocaleElements_uk could not be converted

java.text.resources.LocaleElements_uk_UA.getContents could not be converted

java.text.resources.LocaleElements_uk_UA.LocaleElements_uk_UA could not be converted

java.text.resources.LocaleElements_zh.LocaleElements_zh could not be converted

java.text.resources.LocaleElements_zh_CN.getContents could not be converted

java.text.resources.LocaleElements_zh_CN.LocaleElements_zh_CN could not be converted

java.text.resources.LocaleElements_zh_HK.getContents could not be converted

java.text.resources.LocaleElements_zh_HK.LocaleElements_zh_HK could not be converted

java.text.resources.LocaleElements_zh_SG.LocaleElements_zh_SG could not be converted

java.text.resources.LocaleElements_zh_TW.LocaleElements_zh_TW could not be converted

Java.util Error Messages

java.util.AbstractCollection.add could not be converted

java.util.AbstractCollection.addAll could not be converted

java.util.AbstractCollection.contains could not be converted

java.util.AbstractCollection.containsAll could not be converted

java.util.AbstractCollection.iterator could not be converted

java.util.AbstractCollection.remove could not be converted

java.util.AbstractCollection.removeAll could not be converted

java.util.AbstractCollection.retainAll could not be converted

java.util.AbstractCollection.toArray could not be converted

java.util.AbstractCollection.toArray(javalangObject[]) could not be converted

java.util.AbstractList.AbstractList could not be converted

java.util.AbstractList.addAll could not be converted

java.util.AbstractList.lastIndexOf could not be converted

java.util.AbstractList.listIterator could not be converted

java.util.AbstractList.listIterator(int) could not be converted

java.util.AbstractList.modCount could not be converted

java.util.AbstractList.removeRange could not be converted

java.util.AbstractList.subList could not be converted

java.util.AbstractMap could not be converted

java.util.AbstractMap.AbstractMap could not be converted

java.util.AbstractMap.clear could not be converted

java.util.AbstractMap.containsKey could not be converted

java.util.AbstractMap.containsValue could not be converted

java.util.AbstractMap.entrySet could not be converted

java.util.AbstractMap.equals could not be converted

java.util.AbstractMap.get could not be converted

java.util.AbstractMap.hashCode could not be converted

java.util.AbstractMap.keySet could not be converted

java.util.AbstractMap.put could not be converted

java.util.AbstractMap.putAll could not be converted

java.util.AbstractMap.remove could not be converted

java.util.AbstractMap.values could not be converted

java.util.AbstractSequentialList.AbstractSequentialList could not be converted

java.util.AbstractSequentialList.addAll could not be converted

java.util.AbstractSequentialList.listIterator could not be converted

java.util.AbstractSequentialList.set could not be converted

java.util.AbstractSet.removeAll could not be converted

java.util.ArrayList.toArray could not be converted

java.util.Arrays.sort could not be converted

java.util.BitSet.andNot could not be converted

java.util.Calendar.after could not be converted

java.util.Calendar.AM could not be converted

java.util.Calendar.AM_PM could not be converted

java.util.Calendar.APRIL could not be converted

java.util.Calendar.areFieldsSet could not be converted

java.util.Calendar.AUGUST could not be converted

java.util.Calendar.before could not be converted

java.util.Calendar.Calendar could not be converted

java.util.Calendar.clone could not be converted

java.util.Calendar.complete could not be converted

java.util.Calendar.computeFields could not be converted

java.util.Calendar.computeTime could not be converted

java.util.Calendar.DAY_OF_MONTH could not be converted

java.util.Calendar.DAY_OF_WEEK could not be converted

java.util.Calendar.DAY_OF_WEEK_IN_MONTH could not be converted

java.util.Calendar.DAY_OF_YEAR could not be converted

java.util.Calendar.DECEMBER could not be converted

java.util.Calendar.DST_OFFSET could not be converted

java.util.Calendar.ERA could not be converted

java.util.Calendar.FEBRUARY could not be converted

java.util.Calendar.FIELD_COUNT could not be converted

java.util.Calendar.fields could not be converted

java.util.Calendar.get could not be converted

java.util.Calendar.getAvailableLocales could not be converted

java.util.Calendar.getGreatestMinimum could not be converted

java.util.Calendar.getInstance could not be converted

java.util.Calendar.getLeastMaximum could not be converted

java.util.Calendar.getMaximum could not be converted

java.util.Calendar.getMinimalDaysInFirstWeek could not be converted

java.util.Calendar.getMinimum could not be converted

java.util.Calendar.getTimeInMillis could not be converted

java.util.Calendar.getTimeZone could not be converted

java.util.Calendar.HOUR could not be converted

java.util.Calendar.HOUR_OF_DAY could not be converted

java.util.Calendar.internalGet could not be converted

java.util.Calendar.isLenient could not be converted

java.util.Calendar.isSet could not be converted

java.util.Calendar.isSet~ could not be converted

java.util.Calendar.isTimeSet could not be converted

java.util.Calendar.JANUARY could not be converted

java.util.Calendar.JULY could not be converted

java.util.Calendar.JUNE could not be converted

java.util.Calendar.MARCH could not be converted

java.util.Calendar.MAY could not be converted

java.util.Calendar.NOVEMBER could not be converted

java.util.Calendar.OCTOBER could not be converted

java.util.Calendar.PM could not be converted

java.util.Calendar.roll could not be converted

java.util.Calendar.SEPTEMBER could not be converted

java.util.Calendar.setLenient could not be converted

java.util.Calendar.setMinimalDaysInFirstWeek could not be converted

java.util.Calendar.setTimeInMillis could not be converted

java.util.Calendar.setTimeZone could not be converted

java.util.Calendar.time could not be converted

java.util.Calendar.UNDECIMBER could not be converted

java.util.Calendar.WEEK_OF_MONTH could not be converted

java.util.Calendar.WEEK_OF_YEAR could not be converted

java.util.Calendar.ZONE_OFFSET could not be converted

java.util.Collection.add could not be converted

java.util.Collection.addAll could not be converted

java.util.Collection.clear could not be converted

java.util.Collection.contains could not be converted

java.util.Collection.containsAll could not be converted

java.util.Collection.remove could not be converted

java.util.Collection.removeAll could not be converted

java.util.Collection.retainAll could not be converted

java.util.Collection.toArray could not be converted

java.util.Collections.reverseOrder could not be converted

java.util.Collections.synchronizedMap could not be converted

java.util.Collections.synchronizedSortedSet could not be converted

java.util.Collections.unmodifiableMap could not be converted

java.util.Collections.unmodifiableSortedMap could not be converted

java.util.Collections.unmodifiableSortedSet could not be converted

java.util.Date.clone could not be converted

java.util.Date.getTime could not be converted

java.util.Date.getTimezoneOffset could not be converted

java.util.Date.hashCode could not be converted

java.util.Date.parse could not be converted

java.util.Date.UTC could not be converted

java.util.Dictionary.Dictionary could not be converted

java.util.Dictionary.get could not be converted

java.util.Dictionary.isEmpty could not be converted

java.util.Dictionary.keys could not be converted

java.util.Dictionary.put could not be converted

java.util.Dictionary.remove could not be converted

java.util.Entry.setValue could not be converted

java.util.EventListener could not be converted

java.util.EventObject could not be converted

java.util.GregorianCalendar.add could not be converted

java.util.GregorianCalendar.after could not be converted

java.util.GregorianCalendar.before could not be converted

java.util.GregorianCalendar.computeFields could not be converted

java.util.GregorianCalendar.computeTime could not be converted

java.util.GregorianCalendar.getActualMaximum could not be converted

java.util.GregorianCalendar.getActualMinimum could not be converted

java.util.GregorianCalendar.getGreatestMinimum could not be converted

java.util.GregorianCalendar.getGregorianChange could not be converted

java.util.GregorianCalendar.getLeastMaximum could not be converted

java.util.GregorianCalendar.getMaximum could not be converted

java.util.GregorianCalendar.getMinimum could not be converted

java.util.GregorianCalendar.GregorianCalendar could not be converted

java.util.GregorianCalendar.roll could not be converted

java.util.GregorianCalendar.roll(int boolean) could not be converted

java.util.GregorianCalendar.setGregorianChange could not be converted

java.util.HashMap could not be converted

java.util.HashMap.get could not be converted

java.util.HashMap.putAll could not be converted

java.util.HashSet.add could not be converted

java.util.HashSet.clear could not be converted

java.util.HashSet.clone could not be converted

java.util.HashSet.contains could not be converted

java.util.HashSet.HashSet could not be converted

java.util.HashSet.HashSet(int) could not be converted

java.util.HashSet.HashSet(int float) could not be converted

java.util.HashSet.HashSet(javautilCollection) could not be converted

java.util.HashSet.remove could not be converted

java.util.Hashtable.putAll could not be converted

java.util.Hashtable.rehash could not be converted

java.util.HashtableEntry could not be converted

java.util.HashtableEnumerator could not be converted

java.util.Iterator.hasNext could not be converted

java.util.Iterator.next could not be converted

java.util.Iterator.remove could not be converted

java.util.LinkedList.listIterator could not be converted

java.util.List.addAll could not be converted

java.util.List.containsAll could not be converted

java.util.List.lastIndexOf could not be converted

java.util.List.listIterator could not be converted

java.util.List.remove could not be converted

java.util.List.removeAll could not be converted

java.util.List.retainAll could not be converted

java.util.List.subList could not be converted

java.util.List.toArray could not be converted

java.util.ListIterator.add could not be converted

java.util.ListIterator.hasNext could not be converted

java.util.ListIterator.next could not be converted

java.util.ListIterator.hasPrevious could not be converted

java.util.ListIterator.nextIndex could not be converted

java.util.ListIterator.previous could not be converted

java.util.ListIterator.previousIndex could not be converted

java.util.ListIterator.remove could not be converted

java.util.ListIterator.set could not be converted

java.util.ListResourceBundle.getContents could not be converted

java.util.ListResourceBundle.getKeys could not be converted

java.util.ListResourceBundle.ListResourceBundle could not be converted

java.util.Locale.CHINESE could not be converted

java.util.Locale.getDefault could not be converted

java.util.Locale.getDisplayVariant could not be converted

java.util.Locale.getVariant could not be converted

java.util.Locale.setDefault could not be converted

java.util.Map.containsValue could not be converted

java.util.Map.get could not be converted

java.util.Map.isEmpty could not be converted

java.util.Map.putAll could not be converted

java.util.MissingResourceException.getClassName could not be converted

java.util.MissingResourceException.getKey could not be converted

java.util.MissingResourceException.MissingResourceException could not be converted

java.util.Properties.defaults could not be converted

java.util.Properties.list could not be converted

java.util.Properties.load could not be converted

java.util.Properties.save could not be converted

java.util.PropertyResourceBundle.getKeys could not be converted

java.util.PropertyResourceBundle.PropertyResourceBundle could not be converted

java.util.Random.nextBoolean could not be converted

java.util.Random.nextGaussian could not be converted

java.util.ResourceBundle.getKeys could not be converted

java.util.ResourceBundle.getStringArray could not be converted

java.util.ResourceBundle.handleGetObject could not be converted

java.util.ResourceBundle.parent could not be converted

java.util.ResourceBundle.ResourceBundle could not be converted

java.util.ResourceBundle.setParent could not be converted

java.util.Set.add could not be converted

java.util.Set.addAll could not be converted

java.util.Set.clear could not be converted

java.util.Set.contains could not be converted

java.util.Set.containsAll could not be converted

java.util.Set.remove could not be converted

java.util.Set.removeAll could not be converted

java.util.Set.retainAll could not be converted

java.util.Set.toArray could not be converted

java.util.Set.toArray(javalangObject[]) could not be converted

java.util.SimpleTimeZone.clone could not be converted

java.util.SimpleTimeZone.getRawOffset could not be converted

java.util.SimpleTimeZone.setEndRule could not be converted

java.util.SimpleTimeZone.setRawOffset could not be converted

java.util.SimpleTimeZone.setStartRule could not be converted

java.util.SimpleTimeZone.setStartYear could not be converted

java.util.SimpleTimeZone.SimpleTimeZone could not be converted

java.util.SortedMap.comparator could not be converted

java.util.SortedMap.headMap could not be converted

java.util.SortedMap.subMap could not be converted

java.util.SortedMap.tailMap could not be converted

java.util.SortedSet.comparator could not be converted

java.util.Stack.elements could not be converted

java.util.StringTokenizer.StringTokenizer could not be converted

java.util.SystemClassLoader could not be converted

java.util.Timer.cancel could not be converted

java.util.Timer.schedule(javautilTimerTask javautilDate) could not be converted

java.util.Timer.schedule(javautilTimerTask javautilDate long) could not be converted

java.util.Timer.schedule(javautilTimerTask long) could not be converted

java.util.Timer.schedule(javautilTimerTask long long) could not be converted

java.util.Timer.scheduleAtFixedRate(javautilTimerTask javautilDate long) could not be converted

java.util.Timer.scheduleAtFixedRate(javautilTimerTask long long) could not be converted

java.util.Timer.Timer could not be converted

java.util.TimeZone.clone could not be converted

java.util.TimeZone.getAvailableIDs could not be converted

java.util.TimeZone.getDisplayName(boolean int) could not be converted

java.util.TimeZone.getDisplayName(boolean int javautilLocale) could not be converted

java.util.TimeZone.getDisplayName(javautilLocale) could not be converted

java.util.TimeZone.getRawOffset could not be converted

java.util.TimeZone.getTimeZone could not be converted

java.util.TimeZone.hasSameRules could not be converted

java.util.TimeZone.setDefault could not be converted

java.util.TimeZone.setID could not be converted

java.util.TimeZone.setRawOffset could not be converted

java.util.TimeZone.TimeZone could not be converted

java.util.TreeMap.comparator could not be converted

java.util.TreeMap.headMap could not be converted

java.util.TreeMap.putAll could not be converted

java.util.TreeMap.subMap could not be converted

java.util.TreeMap.tailMap could not be converted

java.util.TreeMap.TreeMap(javautilMap) could not be converted

java.util.TreeMap.TreeMap(javautilSorteMap) could not be converted

java.util.TreeMap.values could not be converted

java.util.TreeSet.comparator could not be converted

java.util.TreeSet.TreeSet(javautilComparator) could not be converted

java.util.TreeSet.TreeSet(javautilSortedMap) could not be converted

java.util.TreeSet.TreeSet(javautilSortedSet) could not be converted

java.util.Vector.capacityIncrement could not be converted

java.util.Vector.containsAll could not be converted

java.util.Vector.elementData could not be converted

java.util.Vector.removeAll could not be converted

java.util.Vector.toArray could not be converted

java.util.Vector.Vector could not be converted

java.util.VectorEnumerator could not be converted

java.util.WeakHashMap could not be converted

java.util.WeakHashMap.WeakHashMap could not be converted

Javax.servlet Error Messages

javax.servlet.FilterConfig could not be converted

javax.servlet.GenericServlet.destroy could not be converted

javax.servlet.GenericServlet.getInitParameter could not be converted

javax.servlet.GenericServlet.getInitParameterNames could not be converted

javax.servlet.GenericServlet.getServletConfig could not be converted

javax.servlet.GenericServlet.getServletContext could not be converted

javax.servlet.GenericServlet.getServletInfo could not be converted

javax.servlet.GenericServlet.getServletName could not be converted

javax.servlet.GenericServlet.log(javalangString) could not be converted

javax.servlet.GenericServlet.log(javalangString javalangThrowable) could not be converted

javax.servlet.GenericServlet.service could not be converted

javax.servlet.HttpDummyBase.Config could not be converted

javax.servlet.HttpDummyBase.HttpDummyBase could not be converted

javax.servlet.RequestDispatcher could not be converted

javax.servlet.RequestDispatcher.forward could not be converted

javax.servlet.RequestDispatcher.include could not be converted

javax.servlet.Servlet.destroy could not be converted

javax.servlet.Servlet.getServletConfig could not be converted

javax.servlet.Servlet.getServletInfo could not be converted

javax.servlet.Servlet.service could not be converted

javax.servlet.ServletConfig could not be converted

javax.servlet.ServletContext.getContext could not be converted

javax.servlet.ServletContext.getInitParameter could not be converted

javax.servlet.ServletContext.getInitParameterNames could not be converted

javax.servlet.ServletContext.getMajorVersion could not be converted

javax.servlet.ServletContext.getMimeType could not be converted

javax.servlet.ServletContext.getMinorVersion could not be converted

javax.servlet.ServletContext.getNamedDispatcher could not be converted

javax.servlet.ServletContext.getRequestDispatcher could not be converted

javax.servlet.ServletContext.getResource could not be converted

javax.servlet.ServletContext.getResourceAsStream could not be converted

javax.servlet.ServletContext.getResourcePaths could not be converted

javax.servlet.ServletContext.getServerInfo could not be converted

javax.servlet.ServletContext.getServlet could not be converted

javax.servlet.ServletContext.getServletContextName could not be converted

javax.servlet.ServletContext.getServletNames could not be converted

javax.servlet.ServletContext.getServlets could not be converted

javax.servlet.ServletContext.log could not be converted

javax.servlet.ServletContext.setAttribute could not be converted

javax.servlet.ServletContextAttributeEvent could not be converted

javax.servlet.ServletContextAttributeListener could not be converted

javax.servlet.ServletContextEvent could not be converted

javax.servlet.ServletContextListener could not be converted

javax.servlet.ServletInputStream could not be converted

javax.servlet.ServletRequest.getAttribute could not be converted

javax.servlet.ServletRequest.getAttributeNames could not be converted

javax.servlet.ServletRequest.getLocale could not be converted

javax.servlet.ServletRequest.getLocales could not be converted

javax.servlet.ServletRequest.getParameter could not be converted

javax.servlet.ServletRequest.getParameterMap could not be converted

javax.servlet.ServletRequest.getParameterValues could not be converted

javax.servlet.ServletRequest.getProtocol could not be converted

javax.servlet.ServletRequest.getReader could not be converted

javax.servlet.ServletRequest.getRealPath could not be converted

javax.servlet.ServletRequest.getRequestDispatcher could not be converted

javax.servlet.ServletRequest.getScheme could not be converted

javax.servlet.ServletRequest.getServerPort could not be converted

javax.servlet.ServletRequest.removeAttribute could not be converted

javax.servlet.ServletRequest.setAttribute could not be converted

javax.servlet.ServletRequest.setCharacterEncoding could not be converted

javax.servlet.ServletRequestWrapper.getAttribute could not be converted

javax.servlet.ServletRequestWrapper.getAttributeNames could not be converted

javax.servlet.ServletRequestWrapper.getLocale could not be converted

javax.servlet.ServletRequestWrapper.getLocales could not be converted

javax.servlet.ServletRequestWrapper.getParameter could not be converted

javax.servlet.ServletRequestWrapper.getParameterMap could not be converted

javax.servlet.ServletRequestWrapper.getReader could not be converted

javax.servlet.ServletRequestWrapper.getRequestDispatcher could not be converted

javax.servlet.ServletRequestWrapper.getScheme could not be converted

javax.servlet.ServletResponse could not be converted

javax.servlet.ServletResponse.flushBuffer could not be converted

javax.servlet.ServletResponse.getBufferSize could not be converted

javax.servlet.ServletResponse.getLocale could not be converted

javax.servlet.ServletResponse.isCommitted could not be converted

javax.servlet.ServletResponse.resetBuffer could not be converted

javax.servlet.ServletResponse.setBufferSize could not be converted

javax.servlet.ServletResponse.setContentLength could not be converted

javax.servlet.ServletResponseWrapper.flushBuffer could not be converted

javax.servlet.ServletResponseWrapper.getBufferSize could not be converted

javax.servlet.ServletResponseWrapper.getLocale could not be converted

javax.servlet.ServletResponseWrapper.isCommitted could not be converted

javax.servlet.ServletResponseWrapper.setBufferSize could not be converted

javax.servlet.ServletResponseWrapper.setContentLength could not be converted

javax.servlet.ServletResponseWrapper.setLocale could not be converted

javax.servlet.UnavailableException.getServlet could not be converted

javax.servlet.UnavailableException.getUnavailableSeconds could not be converted

javax.servlet.UnavailableException.isPermanent could not be converted

javax.servlet.UnavailableException.UnavailableException could not be converted

Javax.servlet.http Error Messages

javax.servlet.http.Cookie.clone could not be converted

javax.servlet.http.Cookie.getComment could not be converted

javax.servlet.http.Cookie.getMaxAge could not be converted

javax.servlet.http.Cookie.getSecure could not be converted

javax.servlet.http.Cookie.getVersion could not be converted

javax.servlet.http.Cookie.setComment could not be converted

javax.servlet.http.Cookie.setMaxAge could not be converted

javax.servlet.http.Cookie.setSecure could not be converted

javax.servlet.http.Cookie.setVersion could not be converted

javax.servlet.http.HttpServlet.getLastModified could not be converted

javax.servlet.http.HttpServlet.HttpServlet could not be converted

javax.servlet.http.HttpServletRequest.BASIC_AUTH could not be converted

javax.servlet.http.HttpServletRequest.CLIENT_CERT_AUTH could not be converted

javax.servlet.http.HttpServletRequest.DIGEST_AUTH could not be converted

javax.servlet.http.HttpServletRequest.FORM_AUTH could not be converted

javax.servlet.http.HttpServletRequest.getSession could not be converted

javax.servlet.http.HttpServletRequest.isRequestedSessionIdFromCookie could not be converted

javax.servlet.http.HttpServletRequest.isRequestedSessionIdFromURL could not be converted

javax.servlet.http.HttpServletRequest.isRequestedSessionIdValid could not be converted

javax.servlet.http.HttpServletRequestWrapper.getSession could not be converted

javax.servlet.http.HttpServletRequestWrapper.isRequestedSessionIdFromCookie could not be converted

javax.servlet.http.HttpServletRequestWrapper.isRequestedSessionIdFromURL could not be converted

javax.servlet.http.HttpServletRequestWrapper.isRequestedSessionIdValid could not be converted

javax.servlet.http.HttpServletResponse.addDateHeader could not be converted

javax.servlet.http.HttpServletResponse.containsHeader could not be converted

javax.servlet.http.HttpServletResponse.encodeRedirectURL could not be converted

javax.servlet.http.HttpServletResponse.encodeURL could not be converted

javax.servlet.http.HttpServletResponse.SC_TEMPORARY_REDIRECT could not be converted

javax.servlet.http.HttpServletResponse.setDateHeader could not be converted

javax.servlet.http.HttpServletResponseWrapper.containsHeader could not be converted

javax.servlet.http.HttpServletResponseWrapper.setDateHeader could not be converted

javax.servlet.http.HttpServletResponseWrapper.setHeader could not be converted

javax.servlet.http.HttpServletResponseWrapper.setIntHeader could not be converted

javax.servlet.http.HttpSession.getCreationTime could not be converted

javax.servlet.http.HttpSession.getLastAccessedTime could not be converted

javax.servlet.http.HttpSession.getMaxInactiveInterval could not be converted

javax.servlet.http.HttpSession.getSessionContext could not be converted

javax.servlet.http.HttpSession.getValue could not be converted

javax.servlet.http.HttpSession.invalidate could not be converted

javax.servlet.http.HttpSession.putValue could not be converted

javax.servlet.http.HttpSession.removeValue could not be converted

javax.servlet.http.HttpSession.setMaxInactiveInterval could not be converted

javax.servlet.http.HttpSessionActivationListener could not be converted

javax.servlet.http.HttpSessionAttributeListener could not be converted

javax.servlet.http.HttpSessionBindingEvent could not be converted

javax.servlet.http.HttpSessionBindingListener could not be converted

javax.servlet.http.HttpSessionContext could not be converted

javax.servlet.http.HttpSessionEvent could not be converted

javax.servlet.http.HttpSessionListener could not be converted

javax.servlet.http.HttpUtils could not be converted

Javax.servlet.jsp Error Messages

javax.servlet.jsp.HttpJspPage could not be converted

javax.servlet.jsp.JspEngineInfo could not be converted

javax.servlet.jsp.JspFactory could not be converted

javax.servlet.jsp.JspPage could not be converted

javax.servlet.jsp.JspWriter.bufferSize could not be converted

javax.servlet.jsp.JspWriter.DEFAULT_BUFFER could not be converted

javax.servlet.jsp.JspWriter.getBufferSize could not be converted

javax.servlet.jsp.JspWriter.getRemaining could not be converted

javax.servlet.jsp.JspWriter.NO_BUFFER could not be converted

javax.servlet.jsp.JspWriter.UNBOUNDED_BUFFER could not be converted

javax.servlet.jsp.PageContext.APPLICATION could not be converted

javax.servlet.jsp.PageContext.APPLICATION_SCOPE could not be converted

javax.servlet.jsp.PageContext.CONFIG could not be converted

javax.servlet.jsp.PageContext.EXCEPTION could not be converted

javax.servlet.jsp.PageContext.findAttribute could not be converted

javax.servlet.jsp.PageContext.getAttribute could not be converted

javax.servlet.jsp.PageContext.getAttributeNamesInScope could not be converted

javax.servlet.jsp.PageContext.getAttributesScope could not be converted

javax.servlet.jsp.PageContext.getServletConfig could not be converted

javax.servlet.jsp.PageContext.handlePageException could not be converted

javax.servlet.jsp.PageContext.initialize could not be converted

javax.servlet.jsp.PageContext.OUT could not be converted

javax.servlet.jsp.PageContext.PAGE could not be converted

javax.servlet.jsp.PageContext.PageContext could not be converted

javax.servlet.jsp.PageContext.PAGECONTEXT could not be converted

javax.servlet.jsp.PageContext.PAGE_SCOPE could not be converted

javax.servlet.jsp.PageContext.popBody could not be converted

javax.servlet.jsp.PageContext.pushBody could not be converted

javax.servlet.jsp.PageContext.release could not be converted

javax.servlet.jsp.PageContext.removeAttribute could not be converted

javax.servlet.jsp.PageContext.REQUEST could not be converted

javax.servlet.jsp.PageContext.REQUEST_SCOPE could not be converted

javax.servlet.jsp.PageContext.RESPONSE could not be converted

javax.servlet.jsp.PageContext.SESSION could not be converted

javax.servlet.jsp.PageContext.SESSION_SCOPE could not be converted

javax.servlet.jsp.PageContext.setAttribute could not be converted

javax.servlet.jsp.tagext.BodyContent.flush could not be converted

javax.servlet.jsp.tagext.PageData could not be converted

javax.servlet.jsp.tagext.TagAttributeInfo could not be converted

javax.servlet.jsp.tagext.TagData could not be converted

javax.servlet.jsp.tagext.TagExtraInfo could not be converted

javax.servlet.jsp.tagext.TagInfo could not be converted

javax.servlet.jsp.tagext.TagLibraryInfo could not be converted

javax.servlet.jsp.tagext.TagLibraryValidator could not be converted

javax.servlet.jsp.tagext.TagVariableInfo could not be converted

javax.servlet.jsp.tagext.TryCatchFinally could not be converted

javax.servlet.jsp.tagext.ValidationMessage could not be converted

javax.servlet.jsp.tagext.VariableInfo could not be converted

Javax.swing Error Messages

javax.swing.JComponent.addNotify could not be converted

javax.swing.JComponent.getAlignmentX could not be converted

javax.swing.JComponent.getAlignmentY could not be converted

javax.swing.JComponent.getMaximumSize could not be converted

javax.swing.JComponent.getMinimumSize could not be converted

javax.swing.JComponent.isMaximumSizeSet could not be converted

javax.swing.JComponent.isMinimumSizeSet could not be converted

javax.swing.JComponent.isPreferredSizeSet could not be converted

javax.swing.JComponent.print could not be converted

javax.swing.JComponent.printAll could not be converted

javax.swing.JComponent.setAlignmentX could not be converted

javax.swing.JComponent.setAlignmentY could not be converted

javax.swing.JComponent.setMaximumSize could not be converted

javax.swing.JComponent.setMinimumSize could not be converted

javax.swing.JPopupMenu.getInvoker could not be converted

javax.swing.JPopupMenu.getLabel could not be converted

javax.swing.JPopupMenu.isBorderPainted could not be converted

javax.swing.JPopupMenu.isVisible could not be converted

javax.swing.JPopupMenu.pack could not be converted

javax.swing.JPopupMenu.paintBorder could not be converted

javax.swing.JPopupMenu.setBorderPainted could not be converted

javax.swing.JPopupMenu.setLabel could not be converted

Legacy ActiveX Control Reference

The following links apply to ActiveX controls that are available to use with Visual Basic .NET.

> **Note** There are performance limitations for Windows Forms when ActiveX controls are added to them.

Masked Edit Control
: Provides reference information about this control, which provides restricted data input as well as formatted data output. This control supplies visual cues about the type of data being entered or displayed.

MSChart Control
: Provides reference information about this control, which graphically displays data.

Objects, Properties, Methods, and Events for ActiveX Controls

Masked Edit Control Properties, Methods, and Events

MSChart Control Properties, Methods, and Events

Objects (ActiveX Controls) (alphabetical list)

Properties (ActiveX Controls) (alphabetical list)

Methods (ActiveX Controls) (alphabetical list)

Events (ActiveX Controls) (alphabetical list)

See Also

Masked Edit Control | MSChart control

Objects (ActiveX Controls)

Following are the objects related to the ActiveX controls.

- AmbientProperties Object (ActiveX Controls)
- DataBinding Object (ActiveX Controls)
- Masked Edit Control
- MSChart control
    - Axis Object
    - AxisGrid Object
    - AxisScale Object
    - AxisTitle Object
    - Backdrop Object
    - Brush Object
    - CategoryScale Object
    - Coor Object
    - DataGrid Object
    - DataPoint Object
    - DataPointLabel Object
    - DataPoints Collection
    - Fill Object
    - Footnote Object
    - Frame Object
    - Intersection Object
    - Label Object
    - Labels Collection
    - LCoor Object
    - Legend Object
    - Light Object
    - LightSource Object
    - LightSources Collection
    - Location Object
    - Marker Object
    - Pen Object
    - Plot Object
    - PlotBase Object
    - Rect Object
    - Series Collection
    - Series Object
    - SeriesCollection Collection
    - SeriesMarker Object
    - SeriesPosition Object
    - Shadow Object
    - StatLine Object
    - TextLayout Object
    - Tick Object
    - Title Object
    - ValueScale Object
    - View3D Object
    - VtColor Object
    - VtFont Object
    - Wall Object

- Weighting Object

AmbientProperties Object (ActiveX Controls)

An **AmbientProperties** object holds ambient information from a container to suggest behavior to controls contained within the container.

Remarks

Containers provide ambient properties in order to suggest behavior to controls. As an example, **BackColor** is one of the standard ambient properties; the container is suggesting what the well-behaved control should set its **BackColor** property to.

The **AmbientProperties** object's properties are the ambient properties of the container. These properties are read-only.

Some ambient properties are standard, while others are specific to certain containers. A control may access non-standard ambient properties, but this will make the control container-specific. The control should handle the case where an ambient property is not present in the current container.

When the control is compiled, Visual Basic has no way of knowing what container-specific ambient properties may be available when the control is run; therefore references to container-specific ambient properties will always be late bound.

The **AmbientProperties** object is not available when the Initialize event is raised; but is available when the InitProperties event or ReadProperties event is raised.

The **AmbientProperties** object has several standard properties:

- The **BackColor** property, a Color that contains the suggested interior color of the contained control. The Visual Basic supplied default if the container does not support this property is 0x80000005: the system color for a window background.
- The **DisplayAsDefault** property, a Boolean that specifies if the control is the default control. The Visual Basic supplied default if the container does not support this property is **False**.
- The **DisplayName** property, a String containing the name that the control should display for itself. The Visual Basic supplied default if the container does not support this property is an empty string: "".
- The **Font** property, a **Font** object that contains the suggested font information of the contained control. The Visual Basic supplied default if the container does not support this property is MS Sans Serif 8.
- The **ForeColor** property, a Color that contains the suggested foreground color of the contained control. The Visual Basic supplied default if the container does not support this property is 0x80000008: the system color for window text.
- The **LocaleID** property, a Long that specifies the language and country/region of the user. The Visual Basic supplied default if the container does not support this property is the current system locale ID.
- The **MessageReflect** property, a Boolean that specifies if the container supports message reflection. The Visual Basic supplied default if the container does not support this property is **False**.
- The **Palette** property, a **Picture** object whose palette specifies the suggested palette for the contained control.
- The **ScaleUnits** property, a String containing the name of the coordinate units being used by the container. The Visual Basic supplied default if the container does not support this property is an empty string: "".
- The **ShowGrabHandles** property, a Boolean that specifies if the container handles the showing of grab handles. The Visual Basic supplied default if the container does not support this property is **True**.
- The **ShowHatching** property, a Boolean that specifies if the container handles the showing of hatching. The Visual Basic supplied default if the container does not support this property is **True**.
- The **SupportsMnemonics** property, a Boolean that specifies if the container handles access keys for the control. The Visual Basic supplied default if the container does not support this property is **False**.
- The **TextAlign** property, an enumeration that specifies how text is to be aligned. The Visual Basic supplied default if the container does not support this property is **0 - General Align**.
- The **UserMode** property, a Boolean that specifies if the environment is in design mode or end user mode. The Visual Basic supplied default if the container does not support this property is **True**.
- The **UIDead** property, a Boolean that specifies if the User Interface is unresponsive. The Visual Basic supplied default if the container does not support this property is **False**.

> **Note** When you access the Font property of the AmbientProperties object, you do not get a reference to the container's font. Rather, you get a copy (clone) of the font. The reason for this is that AmbientProperties.Font is commonly used to initialize a control's font whenever an instance of the control is added to a container. If a reference to the font itself were supplied, then changing the control's font would change the container's font as well. If for some reason your control requires a reference to the container's font, you can obtain it by accessing the container through the Parent property of the UserControl object.

See Also

Properties, Methods, and Events | Masked Edit Control | MSChart Control

AmbientProperties Object Properties, Methods, and Events

Properties

BackColor, ForeColor Properties (ActiveX Controls)

Font Property (ActiveX Controls)

See Also

Applies To: AmbientProperties Object (ActiveX Controls)

DataBinding Object (ActiveX Controls)

The **DataBinding** object represents a bindable property of a component.

Syntax

DataBinding

Remarks

There is one **DataBinding** object for each property of a component marked as Bindable in the **Procedure Attributes** dialog box.

Visual Basic version 4.0 supported binding only one property of a control to a database at a time. Later versions of Visual Basic give you the ability to bind multiple properties of a control to a database. This is used most commonly with **User** controls.

See Also

Properties, Methods, and Events | Masked Edit Control

DataBinding Object Properties, Methods, and Events

Properties

DataField Property (ActiveX Controls)

DataFormat Property (ActiveX Controls)

DataMember Property (ActiveX Controls)

DataSource Property (ActiveX Controls)

See Also

Applies To: DataBinding Object (ActiveX Controls)

Properties (ActiveX Controls)

Alignment Property (ActiveX Controls)

Ambient Property (ActiveX Controls)

Appearance Property (ActiveX Controls)

BorderStyle Property (ActiveX Controls)

CausesValidation Property (ActiveX Controls)

Container Property (ActiveX Controls)

Count Property (ActiveX Controls)

DataBindings Property (ActiveX Controls)

DataChanged Property (ActiveX Controls)

DataField Property (ActiveX Controls)

DataFormat Property (ActiveX Controls)

DataMember Property (ActiveX Controls)

DataSource Property (ActiveX Controls)

DragIcon Property (ActiveX Controls)

DragMode Property (ActiveX Controls)

Enabled Property (ActiveX Controls)

Font Property (ActiveX Controls)

HelpContextID Property (ActiveX Controls)

HideSelection Property (ActiveX Controls)

hWnd Property (ActiveX Controls)

Index Property (ActiveX Controls)

Key Property (ActiveX Controls)

MouseIcon Property (ActiveX Controls)

MousePointer Property (ActiveX Controls)

Name Property (ActiveX Controls)

Object Property (ActiveX Controls)

OLEDragMode Property (ActiveX Controls)

OLEDropMode Property (ActiveX Controls)

Parent Property (ActiveX Controls)

RecordSource Property (ActiveX Controls)

TabIndex Property (ActiveX Controls)

TabStop Property (ActiveX Controls)

Tag Property (ActiveX Controls)

Text Property (ActiveX Controls)

ToolTipText Property (ActiveX Controls)

Visible Property (ActiveX Controls)

WhatsThisHelpID Property (ActiveX Controls)

Alignment Property (ActiveX Controls)

Returns or sets a value that determines the alignment of text in a control.

Syntax

object.**Alignment** [= *number*]

The **Alignment** property syntax has these parts:

Part	Description
Object	An object expression that evaluates to an object in the Applies To list .
Number	An integer that specifies the type of alignment.

Remarks

By default, text is left-aligned.

See Also

Applies To: MSChart Control | Masked Edit Control

Ambient Property (ActiveX Controls)

Returns an **AmbientProperties** object holding the ambient properties of the container. The **Ambient** property is not available at the control's authoring time, and read-only at the control's run time.

Syntax

object.**Ambient**

The **Ambient** property syntax has this part:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.

See Also

Applies To: AmbientProperties Object (ActiveX Controls)

Appearance Property (ActiveX Controls)

Returns or sets a value that determines the appearance of the object.

Syntax

object.**Appearance**

The *object* placeholder represents an object expression that evaluates to an object in the Applies To list.

Settings

The **Appearance** property settings are:

Setting	Description
0	Flat. Paints controls and forms without visual effects.
1	(Default) 3D. Paints control or object with three-dimensional effects.

See Also

Applies To: Masked Edit Control

BackColor, ForeColor Properties (ActiveX Controls)

BackColor — returns or sets the background color of an object.

ForeColor — returns or sets the foreground color used to display text and graphics in an object.

Syntax

object.**BackColor** [= *color*]

object.**ForeColor** [= *color*]

The **BackColor** and **ForeColor** property syntax's have these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
color	A value or constant that determines the background or foreground colors of an object, as described in Settings.

Settings

Visual Basic uses the Microsoft Windows operating environment red-green-blue (RGB) color scheme. The settings for *color* are:

Setting	Description
Normal RGB colors	Colors specified by using the Color palette or by using the **RGB** or **QBColor** functions in code.
System default colors	Colors specified by system color constants listed in the Visual Basic (VB) object library in the Object Browser. The Windows operating environment substitutes the user's choices as specified in the Control Panel settings.

For all forms and controls, the default settings at design time are:

- **BackColor** — set to the system default color specified by the constant **vbWindowBackground**.
- **ForeColor** — set to the system default color specified by the constant **vbWindowText**.

Remarks

The valid range for a normal RGB color is 0 to 16,777,215 (&HFFFFFF). The high byte of a number in this range equals 0; the lower 3 bytes, from least to most significant byte, determine the amount of red, green, and blue, respectively. The red, green, and blue components are each represented by a number between 0 and 255 (&HFF). If the high byte isn't 0, Visual Basic uses the system colors, as defined in the user's Control Panel settings and by constants listed in the Visual Basic (VB) object library in the Object Browser.

To display text in the Windows operating environment, both the text and background colors must be solid. If the text or background colors you've selected aren't displayed, one of the selected colors may be dithered — that is, comprised of up to three different-colored pixels. If you choose a dithered color for either the text or background, the nearest solid color will be substituted.

> **Note** The **Animation** control displays only two types of AVI files, either uncompressed or compressed in RLE8 format. AVI files compressed with RLE8 display only 8-bit colors. The BackColor property for the **Animation** control is "rounded" to the closest 8-bit color in the standard palette.

See Also

Applies To: Masked Edit Control

BorderStyle Property (ActiveX Controls)

Returns or sets the border style for an object.

Syntax

object.**BorderStyle** = [*value*]

The **BorderStyle** property syntax has these parts:

Part	Description
Object	An object expression that evaluates to an object in the Applies To list.
Value	A value or constant that determines the border style, as described in Settings .

Settings

The **BorderStyle** property settings are:

Constant	Setting	Description
*xxx***None**	0	None (no border or border-related elements).
*xxx***FixedSingle**	1	Fixed Single.

> **Note** Constants for ActiveX controls are prefaced by three letters which are specific to the control. For example, the Masked Edit control uses **mskNone**. However, the description remains the same unless indicated.

See Also

Applies To: Masked Edit Control | MSChart Control

BorderStyle Constants (ActiveX Controls)

Following are the constants associated with the **BorderStyle** property:

Constant	Value	Description
*xxx***None**	0	(Default) No border or border-related elements.
*xxx***FixedSingle**	1	(Default for **ListView** control) Fixed single. There is a single line border around the control.

Note Constants for ActiveX controls are prefaced by three letters which are specific to the control. For example, the Masked Edit control uses **mskNone**. However, the description remains the same unless indicated.

See Also

Masked Edit Control | MSChart Control

CausesValidation Property (ActiveX Controls)

Returns or sets a value that determines whether the Validate event will occur on a second control from which the second control's focus is being shifted.

Syntax

object.**CausesValidation** [= *boolean*]

The **CausesValidation** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
boolean	A Boolean expression that specifies whether the control from which the focus is being shifted fires the Validate event.

Settings

The settings for *boolean* are:

Setting	Description
True	(Default) The control from which the focus has shifted fires its Validate event.
False	The control from which the focus has shifted does not fire its Validate event.

Remarks

The **CausesValidation** property works in tandem with the Validate event to limit when a control can lose focus.

Example

See Validate Event (ActiveX Controls).

See Also

Validate Event (ActiveX Controls)

Applies To: MSChart Control

Container Property (ActiveX Controls)

Returns or sets the container of a control on a **Form**. Not available at design time.

Syntax

Set *object*.**Container** [= *container*]

The **Container** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
container	An object expression that evaluates to an object that can serve as a container for other controls, as described in Remarks.

Remarks

The following controls can contain other controls:

- **Frame** control
- **PictureBox** control
- **SSTab** control

Example

This example demonstrates moving a **CommandButton** control from container to container on a **Form** object. To try this example, paste the code into the Declarations section of a form that contains a **Frame** control, a **PictureBox** control and a **CommandButton**, and then press F5.

```
Private Sub Form_Click()
   Static intX As Integer
   Select Case intX
      Case 0
         Set Command1.Container = Picture1
         Command1.Top= 0
         Command1.Left= 0
      Case 1
         Set Command1.Container = Frame1
         Command1.Top= 0
         Command1.Left= 0
      Case 2
         Set Command1.Container = Form1
         Command1.Top= 0
         Command1.Left= 0
   End Select
   intX = intX + 1
End Sub
```

See Also

Applies To: Masked Edit Control | MSChart Control

Count Property (ActiveX Controls)

Returns the number of objects in a collection.

Syntax

object.**Count**

The *object* placeholder is an object expression that evaluates to an object in the Applies To list.

Remarks

You can use this property with a **For...Next** statement to carry out an operation on the forms or controls in a collection. For example, the following code moves all controls on a form 0.5 inches to the right (**ScaleMode** property setting is 1 or **vbTwips**):

```
For I = 0 To Form1.Controls.Count - 1
   Form1.Controls(I).Left = Form1.Controls(I).Left + 720
Next I
```

You can also use this kind of structure to quickly enable or disable all controls on a form.

When used with the **If TypeOf** statement, you can cycle through all controls and change, for example, the **Enabled** property setting of only the text boxes or the **BackColor** property setting of only the option buttons.

See Also

Applies To: MSChart Control | DataPoints Collection | Labels Collection | LightSources Collection | SeriesCollection Collection

DataBindings Property (ActiveX Controls)

Returns the **DataBindings** collection object containing the bindable properties available to the developer.

Syntax

object.**DataBindings**

The *object* placeholder represents an object expression that evaluates to an object in the Applies To list.

See Also

Applies To: MSChart Control

DataChanged Property (ActiveX Controls)

Returns or sets a value indicating that the data in the bound control has been changed by some process other than that of retrieving data from the current record. Not available at design time.

Syntax

object.**DataChanged** [= *boolean*]

The **DataChanged** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
value	A Boolean expression that indicates whether data has changed, as described in Settings .

Settings

The settings for *boolean* are:

Setting	Description
True	The data currently in the control isn't the same as in the current record.
False	(Default) The data currently in the control, if any, is the same as the data in the current record .

Remarks

When a data control moves from record to record, it passes data from fields in the current record to controls bound to the specific field or the entire record. As data is displayed in the bound controls, the **DataChanged** property is set to **False**. If the user or any other operation changes the value in the bound control, the **DataChanged** property is set to **True**. Simply moving to another record doesn't affect the **DataChanged** property.

When the data control starts to move to a different record, the Validate event occurs. If **DataChanged** is **True** for any bound control, the data control automatically invokes the **Edit** and **Update** methods to post the changes to the database.

If you don't wish to save changes from a bound control to the database, you can set the **DataChanged** property to **False** in the Validate event.

Inspect the value of the **DataChanged** property in your code for a control's Change event to avoid a cascading event. This applies to both bound and unbound controls.

Data Type

Boolean

See Also

Applies To: Masked Edit Control

DataField Property (ActiveX Controls)

Returns or sets the name of a field that a data consumer will be bound to.

Syntax

object.**DataField** [= *string*]

The **DataField** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
string	A string expression that evaluates to the name of one of the fields in the **Recordset** object specified by a da ta source.

Remarks

Bound controls provide access to specific data in your database. Bound controls that manage a single field typically display the value of a specific field in the current record. The **DataSource** property of a bound control specifies a valid data source, and the **DataField** property specifies a valid field name in the **Recordset** object created by the data source. Together, these properties specify what data appears in the bound control.

When you use a **QueryDef** object or SQL statement that returns the results of an expression, the field name is automatically generated by the Microsoft Jet database engine. For example, when you code an SQL aggregate function or an expression in your SQL query, unless you alias the aggregate fields using an AS clause, the field names are automatically generated. Generally, the expression field name is Expr1 followed by a three-character number starting with 000. The first expression returned would be named Expr1000.

It's recommended that you code your SQL queries to alias expression columns as shown below:

```
Adodc1.RecordSource = "Select AVG(Sales)  " _
   & " AS AverageSales From SalesTable"
Text1.DataField = "AverageSales"
Adodc1.Refresh
```

> **Note** Make sure the **DataField** property setting is valid for each bound control. If you change the setting of a data control's **RecordSource** property and then use **Refresh**, the **Recordset** identifies the new object. This may invalidate the **DataField** settings of bound controls and produce a trappable error.

See Also

DataSource Property (ActiveX Controls)

Applies To: Masked Edit Control

DataFormat Property (ActiveX Controls)

Sets or returns the **StdDataFormat** object to which a bound object is attached. Read/write both at design time and run time.

Syntax

object.**DataFormat** = *formatobject*

The **DataFormat** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list .
formatobject	Required. A **StdDataFormat** object.

Remarks

The **DataFormat** property can also be set through a property page from the Properties window. If the setting in code differs from the property page, the property page setting is used for the first record fetched, and from then on the values in code will be used.

The **DataFormat** property of the **DataCombo** control is an **Extender** property. Therefore it is always visible on the property sheet and can be set in code. However the **DataCombo** control only formats the top item in its list. This can be disconcerting to the end user who sees a formatted top item, but is only given a list of unformatted items to select from. The formatted item may also mislead end users who assume the item will be entered in the database as formatted. For these reasons, it is advisable to not set the **DataFormat** property when using the **DataCombo** control.

See Also

Applies To: Masked Edit Control

DataMember Property (ActiveX Controls)

Returns or sets a specified data member from among several offered by the data provider.

Syntax

object.**DataMember** [= *string*]

Part	Description
object	Required. An object expression that evaluates to an object in the Applies To list .
string	The name of a data member.

Remarks

A data provider can have multiple sets of data that a data consumer can choose to bind to. Each set of data is called a "data member," and is identified by a unique string.

For example, when using a **Data Environment** that contains several **Command** objects as a **DataSource**, the **DataMember** specifies which **Command** object to use.

When using a class module or a user control as a data source, program the **GetDataMember** to return an appropriate data member. The event's *DataMember* argument contains the value of the **DataMember** property. By querying the argument, you can determine what data member is requested, and pass back the appropriate data through the *Data* argument.

See Also

Applies To: MSChart Control

DataSource Property (ActiveX Controls)

Returns or sets a data source through which a data consumer is bound to a database.

Syntax

object.**DataSource** [=*datasource*]

The **DataSource** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
datasource	An object reference that qualifies as a data source, including **ADO Recordset** objects, and classes or use r controls defined as data sources (**DataSourceBehavior** property = **vbDataSource**).

Remarks

Use the **Set** statement to set the **DataSource** property, as shown below:

```
Set DataGrid1.DataSource = ADODC1
```

See Also

Applies To: MSChart Control

DragIcon Property (ActiveX Controls)

Returns or sets the icon to be displayed as the pointer in a drag-and-drop operation.

Syntax

object.**DragIcon** [= *icon*]

The **DragIcon** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
icon	Any code reference that returns a valid icon, such as a reference to a form's icon (`Form1.Icon`), a reference to another control's **DragIcon** property (`Text1.DragIcon`), or the **LoadPicture** function.

Settings

The settings for *icon* are:

Setting	Description
(none)	(Default) An arrow pointer inside a rectangle.
Icon	A custom mouse pointer. You specify the icon by setting it using the Properties window at design time. You can also use the **LoadPicture** function at run time. The file you load must have the .ico filename extension and format.

Remarks

You can use the **DragIcon** property to provide visual feedback during a drag-and-drop operation — for example, to indicate that the source control is over an appropriate target. **DragIcon** takes effect when the user initiates a drag-and-drop operation. Typically, you set **DragIcon** as part of a MouseDown or DragOver event procedure.

> **Note** At run time, the **DragIcon** property can be set to any object's **DragIcon** or **Icon** property, or you can assign it an icon returned by the **LoadPicture** function.

When you set the **DragIcon** property at run time by assigning the **Picture** property of one control to the **DragIcon** property of another control, the **Picture** property must contain an .ico file, not a .bmp file.

Example

This example changes the **DragIcon** property setting each time you drag a **PictureBox** control. To try this example, paste the code into the Declarations section of a form that contains a **PictureBox** control. Set the **DragMode** property = 1, and then press F5 and click and drag the **PictureBox** control.

```
Private Sub Form_DragDrop (Source As Control, X As Single, Y As Single)
   Dim Pic   ' Declare variable.
   Source.Move X, Y   ' Set position of control.
   Pic = "ICONS\OFFICE\CRDFLE01.ICO"   ' Get name of icon file.
   If Source.DragIcon = False Then   ' If no picture loaded,
      Source.DragIcon = LoadPicture(Pic)   ' load picture.
   Else
      Source.DragIcon = LoadPicture()   ' Unload picture.
   End If
End Sub
```

See Also

Drag Method (ActiveX Controls) | DragDrop Event (ActiveX Controls) | DragMode Property (ActiveX Controls) | MousePointer Property (ActiveX Controls)

Applies To: MSChart Control | Masked Edit Control

DragMode Property (ActiveX Controls)

Returns or sets a value that determines whether manual or automatic drag mode is used for a drag-and-drop operation.

Syntax

object.**DragMode** [= *number*]

The **DragMode** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list .
number	An integer that specifies the drag mode, as described in Settings.

Settings

The settings for *number* are:

Constant	Setting	Description
VbManual	0	(Default) Manual — requires using the **Drag** method to initiate a drag-and-drop operation on the source control.
VbAutomatic	1	Automatic — clicking the source control automatically initiates a drag-and-drop operation. **OLE** container controls are automatically dragged only when they don't have the focus.

Remarks

When **DragMode** is set to 1 (Automatic), the control doesn't respond as usual to mouse events. Use the 0 (Manual) setting to determine when a drag-and-drop operation begins or ends; you can use this setting to initiate a drag-and-drop operation in response to a keyboard or menu command or to enable a source control to recognize a MouseDown event prior to a drag-and-drop operation.

Clicking while the mouse pointer is over a target object or form during a drag-and-drop operation generates a **DragDrop** event for the target object. This ends the drag-and-drop operation. A drag-and-drop operation may also generate a DragOver event.

> **Note** While a control is being dragged, it can't recognize other user-initiated mouse or keyboard events (**KeyDown**, **KeyPress** or **KeyUp**, **MouseDown**, **MouseMove**, or **MouseUp**). However, the control can receive events initiated by code.

Example

This example enables and disables the ability to drag a **CommandButton** control each time a form is clicked. To try this example, paste the code into the Declarations section of a form that contains a **CommandButton**, and then press F5 and click the form.

```
Private Sub Form_Click ()
 ' Check DragMode.
   If Command1.DragMode = vbManual Then
      ' Turn it on.
  Command1.DragMode = vbAutomatic
   Else
      ' Or turn it off.
     Command1.DragMode = vbManual
   End If
End Sub
```

See Also

DragDrop Event (ActiveX Controls) | DragIcon Property (ActiveX Controls) | DragOver Event (ActiveX Controls)

Applies To: MSChart Control | Masked Edit Control

Drag-and-Drop Constants (ActiveX Controls)

DragOver Event

Constant	Value	Description
vbEnter	0	Source control dragged into target
vbLeave	1	Source control dragged out of target
vbOver	2	Source control dragged from one position in target to another

Drag Method (Controls)

Constant	Value	Description
vbCancel	0	Cancel drag operation
vbBeginDrag	1	Begin dragging control
vbEndDrag	2	Drop control

DragMode Property

Constant	Value	Description
vbManual	0	Manual
vbAutomatic	1	Automatic

Enabled Property (ActiveX Controls)

Returns or sets a value that determines whether a form or control can respond to user-generated events.

Syntax

object.**Enabled** [= *boolean*]

The **Enabled** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list. If *object* is omitted, t he form associated with the active form module is assumed to be *object*.
boolean	A Boolean expression that specifies whether *object* can respond to user-generated events.

Settings

The settings for *boolean* are:

Setting	Description
True	(Default) Allows *object* to respond to events.
False	Prevents *object* from responding to events.

Remarks

The **Enabled** property allows objects to be enabled or disabled at run time. For example, you can disable objects that don't apply to the current state of the application. You can also disable a control used purely for display purposes, such as a text box that provides read-only information.

See Also

Applies To: Masked Edit Control | MSChart Control

Font Property (ActiveX Controls)

Returns a **Font** object.

Syntax

object.**Font**

The *object* placeholder represents an object expression that evaluates to an object in the Applies To list.

Remarks

Use the **Font** property of an object to identify a specific **Font** object whose properties you want to use. For example, the following code changes the **Bold** property setting of a **Font** object identified by the **Font** property of a **TextBox** object:

```
txtFirstName.Font.Bold = True
```

See Also

Applies To: AxisTitle Object | Masked Edit Control

Height, Width Properties (ActiveX Controls)

Return or set the dimensions of an object or the width of the **Columns** object of a **DataGrid** control. For the **Printer** and **Screen** objects, not available at design time.

Syntax

object.**Height** [= *number*]

object.**Width** [= *number*]

The **Height** and **Width** property syntaxes have these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
number	A numeric expression specifying the dimensions of an object, as described in Settings.

Settings

Measurements are calculated as follows:

- **Control** — measured from the center of the control's border so that controls with different border widths align correctly. These properties use the scale units of a control's container.

Remarks

Use the **Height**, **Width**, **Left**, and **Top** properties for operations or calculations based on an object's total area, such as sizing or moving the object. Use the **ScaleLeft**, **ScaleTop**, **ScaleHeight**, and **ScaleWidth** properties for operations or calculations based on an object's internal area, such as drawing or moving objects within another object.

Example

This example sets the size of a form to 75 percent of screen size and centers the form when it is loaded. To try this example, paste the code into the Declarations section of a form. Then press F5 and click the form.

```
Private Sub Form_Click ()
   Width = Screen.Width * .75   ' Set width of form.
   Height = Screen.Height * .75   ' Set height of form.
   Left = (Screen.Width - Width) / 2   ' Center form horizontally.
   Top = (Screen.Height - Height) / 2   ' Center form vertically.
End Sub
```

See Also

Left, Top Properties (ActiveX Controls) | Move Method (ActiveX Controls)

Applies To: Masked Edit Control | MSChart Control | Pen Object | StatLine Object | Wall Object

HelpContextID Property (ActiveX Controls)

Returns or sets an associated context number for an object. Used to provide context-sensitive Help for your application.

Syntax

object.**HelpContextID** [= *number*]

The **HelpContextID** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list. If *object* is omitted, the form associated with the active form module is assumed to be *object*.
number	A numeric expression that specifies the context number of the Help topic associated with *object*.

Settings

The settings for *number* are:

Setting	Description
0	(Default) No context number specified.
> 0	An integer specifying a valid context number.

Remarks

For context-sensitive Help on an object in your application, you must assign the same context number to both *object* and to the associated Help topic when you compile your Help file.

If you've created a Microsoft Windows operating environment Help file for your application and set the application's **HelpFile** property, when a user presses the F1 key, Visual Basic automatically calls Help and searches for the topic identified by the current context number.

The current context number is the value of **HelpContextID** for the object that has the focus. If **HelpContextID** is set to 0, then Visual Basic looks in the **HelpContextID** of the object's container, and then that object's container, and so on. If a nonzero current context number can't be found, the F1 key is ignored.

For a **Menu** control, **HelpContextID** is normally read/write at run time. But **HelpContextID** is read-only for menu items that are exposed or supplied by Visual Basic to add-ins, such as the Add-In Manager command on the Add-Ins menu.

See Also

Applies To: Masked Edit Control | MSChart Control

HideSelection Property (ActiveX Controls)

Returns a value that determines whether selected text appears highlighted when a control loses the focus.

Syntax

object.**HideSelection**

The *object* placeholder represents an object expression that evaluates to an object in the Applies To list.

Return Values

The **HideSelection** property return values are:

Value	Description
True	(Default) Selected text doesn't appear highlighted when the control loses the focus .
False	Selected text appears highlighted when the control loses the focus.

Remarks

You can use this property to indicate which text is highlighted while another form or a dialog box has the focus — for example, in a spell-checking routine.

See Also

Applies To: Masked Edit Control

hWnd Property (ActiveX Controls)

Returns a handle to a form or control.

> **Note** This property is not supported for the **OLE** container control.

Syntax

object.**hWnd**

The *object* placeholder represents an object expression that evaluates to an object in the Applies To list.

Remarks

The Microsoft Windows operating environment identifies each form and control in an application by assigning it a handle, or **hWnd**. The **hWnd** property is used with Windows API calls. Many Windows operating environment functions require the **hWnd** of the active window as an argument.

> **Note** Because the value of this property can change while a program is running, never store the **hWnd** value in a variable.

See Also

Applies To: Masked Edit Control | MSChart Control

Index Property (ActiveX Controls)

Returns or sets the number that uniquely identifies an object in a collection.

Syntax

object.**Index**

The object placeholder is an object expression that evaluates to an object in the Applies To list.

Remarks

The **Index** property is set by default to the order of the creation of objects in a collection. The index for the first object in a collection will always be one (1).

The value of the **Index** property of an object can change when objects in the collection are reordered, such as when you set the **Sorted** property to **True**. If you expect the **Index** property to change dynamically, it may be more useful to refer to objects in a collection by using the **Key** property.

See Also

Index Property (Brush) | Index Property (Intersection) | Name Property (ActiveX Controls) | Tag Property (ActiveX Controls)

Applies To: Brush Object | Intersection Object | Masked Edit Control | MSChart Control

Key Property (ActiveX Controls)

Returns or sets a string that uniquely identifies a member in a collection.

Syntax

object.**Key** [= *string*]

The **Key** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list .
string	A unique string identifying a member in a collection.

Remarks

If the string is not unique, an error will occur.

You can set the **Key** property when you use the **Add** method to add an object to a collection.

The value of the **Index** property of an object can change when objects in the collection are reordered, such as when you set the **Sorted** property to **True**. If you expect the **Index** property to change dynamically, refer to objects in a collection using the **Key** property.

In addition, you can use the **Key** property to make your Visual Basic project "self-documenting" by assigning meaningful names to the objects in a collection.

See Also

Applies To: LightSources Collection

Left, Top Properties (ActiveX Controls)

Left — returns or sets the distance between the internal left edge of an object and the left edge of its container.

Top — returns or sets the distance between the internal top edge of an object and the top edge of its container.

Syntax

object.**Left** [= *value*]

object.**Top** [= *value*]

The **Left** and **Top** property syntaxes have these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list .
value	A numeric expression specifying distance.

Remarks

For a form, the **Left** and **Top** properties are always expressed in twips; for a control, they are measured in units depending on the coordinate system of its container. The values for these properties change as the object is moved by the user or by code.

For either property, you can specify a single-precision number.

Use the **Left**, **Top**, **Height**, and **Width** properties for operations based on an object's external dimensions, such as moving or resizing. Use the **ScaleLeft**, **ScaleTop**, **ScaleHeight**, and **ScaleWidth** properties for operations based on an object's internal dimensions, such as drawing or moving objects that are contained within the object.

Example

This example sets the size of a form to 75 percent of screen size and centers the form when it is loaded. To try this example, paste the code into the Declarations section of a form, and then press F5 and click the form.

```
Private Sub Form_Click ()
   Width = Screen.Width * .75   ' Set width of form.
   Height = Screen.Height * .75   ' Set height of form.
   Left = (Screen.Width - Width) / 2   ' Center form horizontally.
   Top = (Screen.Height - Height) / 2   ' Center form vertically.
End Sub
```

See Also

Height, Width Properties (ActiveX Controls) | Move Method (ActiveX Controls)

Applies To: Masked Edit Control | MSChart Control

MouseIcon Property (ActiveX Controls)

Returns or sets a custom mouse icon.

Syntax

object.**MouseIcon = LoadPicture(***pathname***)**

object.**MouseIcon [=** *picture***]**

The **MouseIcon** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
pathname	A string expression specifying the path and filename of the file containing the custom icon .
picture	The **Picture** property of a **Form** object, **PictureBox** control, or **Image** control.

Remarks

The **MouseIcon** property provides a custom icon that is used when the **MousePointer** property is set to 99.

You can use the **MouseIcon** property to load either cursor or icon files. The **MouseIcon** property provides your program with easy access to custom cursors of any size, with any desired hot spot location. Visual Basic does not load animated cursor (.ani) files, even though 32-bit versions of Windows support these cursors.

See Also

Applies To: Masked Edit Control

MousePointer Property (ActiveX Controls)

Returns or sets a value indicating the type of mouse pointer displayed when the mouse is over a particular part of an object at run time.

Syntax

object.**MousePointer** [= *value*]

The **MousePointer** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
value	An integer specifying the type of mouse pointer displayed, as described in Settings .

Settings

The settings for *value* are:

Constant	Value	Description
*xxx***Default**	0	(Default) Shape determined by the object.
*xxx***Arrow**	1	Arrow.
*xxx***Crosshair**	2	Cross (crosshair pointer).
*xxx***IBeam**	3	I beam.
*xxx***IconPointer**	4	Icon (small square within a square).
*xxx***SizePointer**	5	Size (four-pointed arrow pointing north, south, east, and west) .
*xxx***SizeNESW**	6	Size NE SW (double arrow pointing northeast and southwest).
*xxx***SizeNS**	7	Size N S (double arrow pointing north and south).
*xxx***SizeNWSE**	8	Size NW SE (double arrow pointing northwest and southeast).
*xxx***SizeWE**	9	Size W E (double arrow pointing west and east).
*xxx***UpArrow**	10	Up Arrow.
*xxx***Hourglass**	11	Hourglass (wait).
*xxx***NoDrop**	12	No Drop.
*xxx***ArrowHourglass**	13	Arrow and hourglass.
*xxx***ArrowQuestion**	14	Arrow and question mark.
*xxx***SizeAll**	15	Size all.
*xxx***Custom**	99	Custom icon specified by the **MouseIcon** property.

> **Note** The *xxx* prefix refers to the ActiveX control in use. For example, the Masked Edit control uses **msk** for the prefix. Prefixes for the constants change with the specific control or group of controls. However, the description remains the same unless indicated.

Remarks

You can use this property when you want to indicate changes in functionality as the mouse pointer passes over controls on a form or dialog box. The Hourglass setting (11) is useful for indicating that the user should wait for a process or operation to finish.

> **Note** If your application calls DoEvents, the **MousePointer** property may temporarily change when over an ActiveX control.

See Also

Applies To: Masked Edit Control

MousePointer Constants (ActiveX Controls)

Following are the constants for mouse pointers:

Constant	Value	Description
*xxx***Default**	0	(Default) Shape determined by the object.
*xxx***Arrow**	1	Arrow.
*xxx***Cross**	2	Cross (cross-hair pointer).
*xxx***IBeam**	3	I Beam.
*xxx***Icon**	4	Icon (small square within a square).
*xxx***Size**	5	Size (four-pointed arrow pointing north, south, east, and west).
*xxx***SizeNESW**	6	Size NE SW (double arrow pointing northeast and southwest).
*xxx***SizeNS**	7	Size N S (double arrow pointing north and south).
*xxx***SizeNWSE**	8	Size NW, SE.
*xxx***SizeEW**	9	Size E W (double arrow pointing east and west).
*xxx***UpArrow**	10	Up Arrow.
*xxx***Hourglass**	11	Hourglass (wait).
*xxx***NoDrop**	12	No Drop.
*xxx***ArrowHourglass**	13	Arrow and hourglass.
*xxx***ArrowQuestion**	14	Arrow and question mark.
*xxx***SizeAll**	15	Size all.
*xxx***Custom**	99	Custom icon specified by the **MouseIcon** property.

Note The *xxx* prefix refers to the ActiveX control in use. For example, the Masked Edit control uses **msk** for the prefix. Prefixes for the constants change with the specific control or group of controls. However, the description remains the same unless indicated.

See Also

MousePointer Property (ActiveX Controls)

Name Property (ActiveX Controls)

Returns the name used in code to identify a form, control, or data access object. Read-only at run time.

Returns or sets the name of a font object.

Syntax

object.**Name**

The *object* placeholder represents an object expression that evaluates to an object in the Applies To list. If *object* is omitted, the form associated with the active form module is assumed to be *object*.

Remarks

The default name for new objects is the kind of object plus a unique integer. For example, the first new **Form** object is Form1, a new **MDIForm** object is MDIForm1, and the third **TextBox** control you create on a form is Text3.

An object's **Name** property must start with a letter and can be a maximum of 40 characters. It can include numbers and underline (_) characters but can't include punctuation or spaces. Forms can't have the same name as another public object such as **Clipboard**, **Screen**, or **App**. Although the **Name** property setting can be a keyword, property name, or the name of another object, this can create conflicts in your code.

You can use a form's **Name** property with the **Dim** statement at run time to create other instances of the form. You can't have two forms with the same name at design time.

You can create an array of controls of the same type by setting the **Name** property to the same value. For example, when you set the name of all option buttons in a group to MyOpt, Visual Basic assigns unique values to the **Index** property of each control to distinguish it from others in the array. Two controls of different types can't share the same name.

> **Note** Although Visual Basic often uses the **Name** property setting as the default value for the **Caption**, **LinkTopic**, and **Text** properties, changing one of these properties doesn't affect the others.

Changing the case of the **Name** property value for a Form or other module without otherwise changing the name itself, however, can cause a "Conflicting names" error message the next time the project containing the form or module is loaded. For example, changing "Form1" to "form1" will cause the error; changing "Form1" to "formX" will not.

The error is caused by the way module names are stored within the project file — the procedure for changing names within the project file isn't case sensitive, while the procedure for reading names on project load is.

See Also

Index Property (ActiveX Controls) | Text Property (ActiveX Controls)

Applies To: Masked Edit Control | MSChart Control

Object Property (ActiveX Controls)

Returns the object and/or a setting of an object's method or property.

Syntax

object.**Object**[.*property* | .*method*]

The **Object** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list .
property	Property that the object supports.
method	Method that the object supports.

Remarks

Use this property to specify an object you want to use in an Automation task.

You use the object returned by the **Object** property in an Automation task by using the properties and methods of that object. For information on which properties and methods an object supports, see the documentation for the application that created the object.

See Also

Applies To: Masked Edit Control | MSChart Control

OLEDragMode Property (ActiveX Controls)

Returns or sets whether the component or the programmer handles an OLE drag/drop operation.

Syntax

object.**OLEDragMode** = *mode*

The **OLEDragMode** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
mode	An integer which specifies the method with which an component handles OLE drag/drop operations, as de scribed in Settings.

Settings

The settings for *mode* are:

Constant	Value	Description
*xxx***OLEDragManual**	0	Manual. The programmer handles all OLE drag/drop operations.
*xxx***OLEDragAutomatic**	1	Automatic. The component handles all OLE drag/drop operations .

Remarks

When **OLEDragMode** is set to **Manual**, you must call the **OLEDrag** method to start dragging, which then triggers the **OLEStartDrag** event.

When **OLEDragMode** is set to **Automatic**, the source component fills the **DataObject** object with the data it contains and sets the *effects* parameter before initiating the OLEStartDrag event (as well as the OLESetData and other source-level OLE drag/drop events) when the user attempts to drag out of the control. This gives you control over the drag/drop operation and allows you to intercede by adding other formats, or by overriding or disabling the automatic data and formats using the **Clear** or **SetData** methods.

If the source's **OLEDragMode** property is set to **Automatic**, and no data is loaded in the **OLEStartDrag** event, or *aftereffects* is set to **0**, then the OLE drag/drop operation does not occur.

> **Note** If the **DragMode** property of a control is set to **Automatic**, the setting of **OLEDragMode** is ignored, because regular Visual Basic drag and drop events take precedence.

See Also

Applies To: Masked Edit Control | MSChart Control

OLEDropMode Property (ActiveX Controls)

Returns or sets how a target component handles drop operations.

Syntax

object.**OLEDropMode** [= *mode*]

The **OLEDropMode** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
mode	An enumerated integer which specifies the method which a component handles OLE drag/drop operations, as described in Settings.

Settings

The settings for *mode* are:

Constant	Value	Description
*xxx***OLEDropNone**	0	None. The target component does not accept OLE drops and displays the No Drop cursor.
*xxx***OLEDropManual**	1	Manual. The target component triggers the OLE drop events, allowing the programmer to handle the OLE drop operation in code.
*xxx***OLEDropAutomatic**	2	Automatic. The target component automatically accepts OLE drops if the **DataObject** object contains data in a format it recognizes. No mouse or OLE drag/drop events on the target will occur when **OLEDropMode** is set to *xxx***OLEDropAutomatic**.

Remarks

> **Note** The target component inspects what is being dragged over it in order to determine which events to trigger: the OLE drag/drop events, or the Visual Basic drag/drop events. There is no collision of components or confusion about which events are fired, because only one type of object can be dragged at a time.

See Also

Applies To: Masked Edit Control | MSChart Control

Parent Property (ActiveX Controls)

Returns the form, object, or collection that contains a control or another object or collection.

Syntax

object.**Parent**

The *object* placeholder represents an object expression that evaluates to an object in the Applies To list.

Remarks

Use the **Parent** property to access the properties, methods, or controls of an object's parent. For example:

```
MyButton.Parent.MousePointer = 4
```

The **Parent** property is useful in an application in which you pass objects as arguments. For example, you could pass a control variable to a general procedure in a module, and use the **Parent** property to access its parent form.

There is no relationship between the **Parent** property and the **MDIChild** property. There is, however, a parent-child relationship between an **MDIForm** object and any **Form** object that has its **MDIChild** property set to **True**.

Example

This example passes a control from a form that doesn't have the focus to a procedure in a module, and then displays the state of the control on the parent form. To try this example, create three forms: Form1, containing a **CommandButton** control, and Form2 and Form3, each containing a **CheckBox** control. You must also create a new module (click Add Module in the Project menu). Paste the code into the Declarations sections of the respective forms or module, and then press F5 to run the program.

```
' Enter this code into Form1.
Private Sub Form_Load ()
   Form2.Show   ' Display all forms.
   Form3.Show
   Form2.AutoRedraw = True
   Form3.AutoRedraw = True
End Sub

Private Sub Command1_Click ()
   ReadCheckBox Form2.Check1   ' Call procedure in other module
   ReadCheckBox Form3.Check1   ' and send control as argument.
End Sub

' Enter this code into Module1.
Sub ReadCheckBox (Source As Control)
   If Source.Value Then
      Source.Parent.Cls   ' Clear parent form.
      Source.Parent.Print "CheckBox is ON."  ' Display on parent form.
   Else
      Source.Parent.Cls   ' Clear parent form.
      Source.Parent.Print "CheckBox is OFF."  ' Display on parent form.
   End If
End Sub
```

See Also

Applies To: Masked Edit Control | MSChart Control

RecordSource Property (ActiveX Controls)

Returns or sets the underlying table, SQL statement, or stored procedure.

Syntax

object.**RecordSource** [= *value*]

The **RecordSource** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list .
value	A string expression specifying a name, as described in Settings.

Settings

The settings for *value* are:

Setting	Description
A table name	The name of one of the tables defined in the **Database** object's **TableDefs** collection.
An SQL query	A valid SQL string using syntax appropriate for the data source.
stored procedure	The name of a stored procedure in the database. When using DAO, the name of one of the **QueryDef** objects in the **Database** object's **QueryDefs** collection.

Remarks

The **RecordSource** property specifies the source of the records accessible through bound controls on your form.

If you set the **RecordSource** property to the name of an existing table in the database, all of the fields in that table are visible to the bound controls attached to the **Data** control. For table-type recordsets (**RecordsetType** = **vbRSTypeTable**), the order of the records retrieved is set by the **Index** object that you select using the **Index** property of the **Recordset**. For dynaset-type and snapshot-type **Recordset** objects, you can order the records by using a SQL statement with an Order By clause in the **RecordSource** property of the **Data** control. Otherwise, the data is returned in no particular order.

If you set the **RecordSource** property to the name of an existing **QueryDef** in the database, all fields returned by the **QueryDef** are visible to the bound controls attached to the **Data** control. The order of the records retrieved is set by the **QueryDef** object's query. For example, the **QueryDef** may include an ORDER BY clause to change the order of the records returned by the **Recordset** created by the **Data** control or a WHERE clause to filter the records. If the **QueryDef** doesn't specify an order, the data is returned in no particular order.

> **Note** At design-time, the **QueryDef** objects displayed in the Properties window for the **RecordSource** property are filtered out to display only **QueryDef** objects that are usable with the **Data** control. **QueryDef** objects which have parameters, and **QueryDef** objects which have the following types are not displayed: **dbQAction**, **dbQCrosstab**, **dbQSQLPassThrough** and **dbQSetOperation**.

If you set the **RecordSource** property to an SQL statement that returns records, all fields returned by the SQL query are visible to the bound controls attached to the **Data** control. This statement may include an ORDER BY clause to change the order of the records returned by the **Recordset** created by the **Data** control or a WHERE clause to filter the records. If the database you specify in the **Database** and **Connect** property isn't a Microsoft Jet engine database, and if the **dbSQLPassThrough** option is set in the **Options** property, your SQL query must use the syntax required by that database engine.

> **Note** Whenever your **QueryDef** or SQL statement returns a value from an expression, the field name of the expression is created automatically by the Microsoft Jet database engine. Generally, the name is Expr1 followed by a three-character number beginning with 000. For example, the first expression would be named: Expr1000.

In most cases you'll want to alias expressions so you know the name of the column to bind to the bound control. See the SQL SELECT statement AS clause for more information.

After changing the value of the **RecordSource** property at run time, you must use the **Refresh** method to enable the change and rebuild the **Recordset**.

At run time, if the **Recordset** specifies an invalid **Table** name, **QueryDef** name, or contains invalid SQL syntax, a trappable error will result. If this error occurs during the initial Form_Load procedure, the error is not trappable.

> **Note** Make sure each bound control has a valid setting for its **DataField** property. If you change the setting of a **Data** control's **RecordSource** property and then use **Refresh**, the **Recordset** identifies the new object. This may invalidate the **DataField** settings of bound controls and cause a trappable error.

Data Type

String

See Also

DataField Property (ActiveX Controls) | DataSource Property (ActiveX Controls) | Index Property (ActiveX Controls)

Applies To: MSChart Control

SelLength, SelStart, SelText Properties (ActiveX Controls)

SelLength — returns or sets the number of characters selected.

SelStart — returns or sets the starting point of text selected; indicates the position of the insertion point if no text is selected.

SelText — returns or sets the string containing the currently selected text; consists of a zero-length string ("") if no characters are selected.

These properties aren't available at design time.

Syntax

object.**SelLength** [= *number*]

object.**SelStart** [= *index*]

object.**SelText** [= *value*]

The **SelLength**, **SelStart**, and **SelText** property syntaxes have these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
number	A numeric expression specifying the number of characters selected. For **SelLength** and **SelStart**, the valid range of settings is 0 to text length — the total number of characters in the edit area of a **ComboBox** or **TextBox** control.
index	A numeric expression specifying the starting point of the selected text, as described in Settings.
value	A string expression containing the selected text.

Remarks

Use these properties for tasks such as setting the insertion point, establishing an insertion range, selecting substrings in a control, or clearing text. Used in conjunction with the **Clipboard** object, these properties are useful for copy, cut, and paste operations.

When working with these properties:

- Setting **SelLength** less than 0 causes a run-time error.
- Setting **SelStart** greater than the text length sets the property to the existing text length; changing **SelStart** changes the selection to an insertion point and sets **SelLength** to 0.
- Setting **SelText** to a new value sets **SelLength** to 0 and replaces the selected text with the new string.

Example

This example enables the user to specify some text to search for and then searches for the text and selects it, if found. To try this example, paste the code into the Declarations section of a form that contains a wide **TextBox** control, and then press F5 and click the form.

```
Private Sub Form_Load ()
   Text1.Text = "Two of the peak human experiences"
   Text1.Text = Text1.Text & " are good food and classical music."
End Sub
Private Sub Form_Click ()
   Dim Search, Where   ' Declare variables.
   ' Get search string from user.
   Search = InputBox("Enter text to be found:")
   Where = InStr(Text1.Text, Search)   ' Find string in text.
   If Where Then   ' If found,
      Text1.SetFocus
      Text1.SelStart = Where - 1   ' set selection start and
      Text1.SelLength = Len(Search)   ' set selection length.
   Else
      MsgBox "String not found."   ' Notify user.
   End If
End Sub
```

See Also

SelText Property (Masked Edit Control) | Text Property (ActiveX Controls)

Applies To: Masked Edit Control

TabIndex Property (ActiveX Controls)

Returns or sets the tab order of most objects within their parent form.

Syntax

object.**TabIndex** [= *index*]

The **TabIndex** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
index	An integer from 0 to (*n*–1), where *n* is the number of controls on the form that have a **TabIndex** property. A ssigning a **TabIndex** value of less than 0 generates an error.

Remarks

By default, Visual Basic assigns a tab order to controls as you draw them on a form, with the exception of the **Menu**, **Timer**, **Data**, **Image**, **Line** and **Shape** controls, which are not included in the tab order. At run time, invisible or disabled controls and controls that can't receive the focus (**Frame** and **Label** controls) remain in the tab order but are skipped during tabbing.

Each new control is placed last in the tab order. If you change the value of a control's **TabIndex** property to adjust the default tab order, Visual Basic automatically renumbers the **TabIndex** of other controls to reflect insertions and deletions. You can make changes at design time using the Properties window or at run time in code.

The **TabIndex** property isn't affected by the **ZOrder** method.

> **Note** A control's tab order doesn't affect its associated access key. If you press the access key for a **Frame** or **Label** control, the focus moves to the next control in the tab order that can receive the focus.

When loading forms saved as ASCII text, controls with a **TabIndex** property that aren't listed in the form description are automatically assigned a **TabIndex** value. In subsequently loaded controls, if existing **TabIndex** values conflict with earlier assigned values, the controls are automatically assigned new values.

When you delete one or more controls, you can use the **Undo** command to restore the controls and all their properties except for the **TabIndex** property, which can't be restored. **TabIndex** is reset to the end of the tab order when you use Undo.

Example

This example reverses the tab order of a group of buttons by changing the **TabIndex** property of a command button array. To try this example, paste the code into the Declarations section of a form that contains four **CommandButton** controls. Set the **Name** property to CommandX for each button to create the control array, and then press F5 and click the form to reverse the tab order of the buttons.

```
Private Sub Form_Click ()
   Dim I, X   ' Declare variables.
   ' Reverse tab order by setting start value of X.
   If Command1(0).TabIndex = 0 Then X = 4 Else X = 1
      For I = 0 To 3
         Command1(I).Caption = X   ' Set caption.
         Command1(I).TabIndex = X - 1   ' Set tab order.
         If Command1(0).TabIndex = 3 Then
            X = X - 1   ' Decrement X.
         Else
            X = X + 1   ' Increment X.
         End If
      Next I
End Sub
```

See Also

TabStop Property (ActiveX Controls) | ZOrder Method (ActiveX Controls)

Applies To: MSChart Control

TabStop Property (ActiveX Controls)

Returns or sets a value indicating whether a user can use the TAB key to give the focus to an object.

Syntax

object.**TabStop** [= *boolean*]

The **TabStop** property syntax has these parts:

Part	Description
Object	An object expression that evaluates to an object in the Applies To list.
Boolean	A Boolean expression specifying whether the object is a tab stop, as described in Settings .

Settings

The settings for *boolean* are:

Setting	Description
True	(Default) Designates the object as a tab stop.
False	Bypasses the object when the user is tabbing, although the object still holds its place in the actual tab order, as determined by the **TabIndex** property.

Remarks

This property enables you to add or remove a control from the tab order on a form.

See Also

TabIndex Property (ActiveX Controls)

Applies To: Masked Edit Control | MSChart Control

Tag Property (ActiveX Controls)

Returns or sets an expression that stores any extra data needed for your program. Unlike other properties, the value of the **Tag** property isn't used by Visual Basic; you can use this property to identify objects.

Syntax

object.**Tag** [= *expression*]

The **Tag** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
expression	A string expression identifying the object. The default is a zero-length string ("") .

Remarks

You can use this property to assign an identification string to an object without affecting any of its other property settings or causing side effects. The **Tag** property is useful when you need to check the identity of a control or **MDIForm** object that is passed as a variable to a procedure.

> **Tip** When you create a new instance of a form, assign a unique value to the **Tag** property.
>
> **Note** The **Tag** property is of type Variant for ActiveX control collections such as **Toolbar Button** objects, **TreeView Node** objects, **ListView ListItem** and **ColumnHeader** objects, **ImageList ListImage** objects, **TabStrip Tab** objects, and **StatusBar Panel** objects. You can use the **Tag** property to pass values, but it does not allow you to pass objects.

Example

This example displays a unique icon for each control being dragged. To try this example, paste the code into the Declarations section of a form that contains three **PictureBox** controls. Set the **DragMode** property to 1 for Picture1 and Picture2, and then press F5. Use the mouse to drag Picture1 or Picture2 over Picture3 controls.

```
Private Sub Form_Load ()
  Picture1.Tag = "ICONS\ARROWS\POINT03.ICO"
  Picture2.Tag = "ICONS\ARROWS\POINT04.ICO"
End Sub

Private Sub Picture3_DragOver (Source As Control, X As Single, Y As Single, State As Integer)
   If State = vbEnter Then
      ' Select based on each PictureBox's Name property.
      Select Case Source.Name
      Case "Picture1"
         ' Load icon for Picture1.
         Source.DragIcon = LoadPicture(Picture1.Tag)
        Case "Picture2"
         ' Load icon for Picture2.
         Source.DragIcon = LoadPicture(Picture2.Tag)
      End Select
   ElseIf State = vbLeave Then
      ' When source isn't over Picture3, unload icon.
      Source.DragIcon = LoadPicture ()
   End If
End Sub
```

See Also

Name Property (ActiveX Controls)

Applies To: Masked Edit Control | MSChart Control

Text Property (ActiveX Controls)

Returns or sets the text contained in an object.

Syntax

object.**Text** [= *string*]

The **Text** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list .
string	A string expression specifying the text appearing in the object.

Example

This example populates a **TreeView** control with the titles of files in a **ListBox** control. When an item in the **TreeView** control is clicked, the **Text** property is displayed in a **Label** on the form. To try the example, place **TreeView**, **Label**, and **ListBox** controls on a form and paste the code into the form's Declarations section. Run the example and click on any item to see its **Text** property.

```
Private Sub Form_Load()
   Dim nodX As Node  ' Declare an object variable for the Node.
   Dim i As Integer  ' Declare a variable for use as a counter.

   ' Add one Node to the TreeView control, and call it the first node
   Set nodX = TreeView1.Nodes.Add()
   nodX.Text = "First Node"

   'Populate the ListBox
      List1.AddItem "Node1"   ' Add each item to list.
      List1.AddItem "Node2"
      List1.AddItem "Node3"
      List1.AddItem "Node4"
      List1.AddItem "Node5"
      List1.AddItem "Node6"
      List1.AddItem "Node7"

   ' Add child nodes to the first Node object. Use the
   ' ListBox to populate the control.
   For i = 0 To List1.ListCount - 1
      Set nodX = TreeView1.Nodes.Add(1, tvwChild)
      nodX.Text = List1.List(i)
   Next i
   Treeview1.Nodes(1).Selected = True
   nodX.EnsureVisible   ' Make sure the node is visible.
End Sub

Private Sub TreeView1_NodeClick(ByVal Node As Node)
   ' Display the clicked Node object's Text property.
   Label1.Caption = Node.Text
End Sub
```

See Also

SelLength, SelStart, SelText Properties (ActiveX Controls)

Applies To: AxisTitle Object | DataPointLabel Object

ToolTipText Property (ActiveX Controls)

Returns or sets a ToolTip.

Syntax

object.**ToolTipText** [= *ToolTipText*]

The **ToolTipText** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
ToolTipText	The string that is displayed when the mouse pointer hovers over the object.

See Also

Applies To: MSChart Control

Visible Property (ActiveX Controls)

Returns or sets a value indicating whether an object is visible or hidden.

Syntax

object.**Visible** [= *boolean*]

The **Visible** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
boolean	A Boolean expression specifying whether the object is visible or hidden .

Settings

The settings for *boolean* are:

Setting	Description
True	(Default) Object is visible.
False	Object is hidden.

Remarks

To hide an object at startup, set the **Visible** property to **False** at design time. Setting this property in code enables you to hide and later redisplay a control at run time in response to a particular event.

> **Note** Using the **Show** or **Hide** method on a form is the same as setting the form's **Visible** property in code to **True** or **False**, respectively.

Example

This example creates animation using two **PictureBox** controls. To try this example, paste the code into the Declarations section of a form that contains two icon-sized **PictureBox** controls. Set the **Name** property to Picture1 for both **PictureBox** controls to create an array, and then press F5 and click the picture to view the animation.

```
Private Sub Form_Load ()
   Dim I   ' Declare variable.
   Picture1(0).BorderStyle = 0   ' Set BorderStyle.
   Picture1(1).BorderStyle = 0
   ' Load icons into picture boxes.
   Picture1(1).Picture = LoadPicture("ICONS\OFFICE\FILES03B.ICO")
   Picture1(0).Picture = LoadPicture("ICONS\OFFICE\FILES03A.ICO")
   For I = 0 To 1
      Picture1(I).Move 400, 400   ' Place graphics at same spot.
   Next I
   Picture1(1).Visible = False   ' Set to invisible.
   Picture1(0).Visible = True   ' Set to visible.
End Sub

Private Sub Picture1_Click (Index As Integer)
   Dim I   ' Declare variable.
   For I = 0 To 1
      ' Switch the visibility for both graphics.
      Picture1(I).Visible = Not Picture1(I).Visible
   Next I
End Sub
```

See Also

Applies To: Location Object | Marker Object | Masked Edit Control | MSChart Control

WhatsThisHelpID Property (ActiveX Controls)

Returns or sets an associated context number for an object. Use to provide context-sensitive Help for your application using the What's This pop-up in Help for 32-bit Windows operating systems.

Syntax

object.**WhatsThisHelpID** [= *number*]

The **WhatsThisHelpID** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
number	A numeric expression specifying a Help context number, as described in Settings.

Settings

The settings for *number* are:

Setting	Description
0	(Default) No context number specified.
>0	An integer specifying the valid context number for the What's This topic associated with the object.

Remarks

Thirty-two–bit Windows operating systems use the What's This button in the upper-right corner of the window to start Windows Help and load a topic identified by the **WhatsThisHelpID** property.

See Also

HelpContextID Property (ActiveX Controls) | ShowWhatsThis Method (ActiveX Controls)

Applies To: Masked Edit Control | MSChart Control

Methods (ActiveX Controls)

AboutBox Method

Clear Method (ActiveX Controls)

Drag Method (ActiveX Controls)

Move Method (ActiveX Controls)

OLEDrag Method (ActiveX Controls)

Refresh Method (ActiveX Controls)

Remove Method (ActiveX Controls)

SetFocus Method (ActiveX Controls)

ShowWhatsThis Method (ActiveX Controls)

ValidateControls Method (ActiveX Controls)

ZOrder Method (ActiveX Controls)

AboutBox Method

Displays the About box for the control.

Syntax

object.**AboutBox**

The *object* placeholder represents an object expression that evaluates to an object in the Applies To list.

Remarks

This is the same as clicking About in the Properties window.

See Also

Applies To: MSChart Control

Clear Method (ActiveX Controls)

Removes all objects in a collection.

Syntax

*object.***Clear**

The *object* placeholder represents an object expression that evaluates to an object in the Applies To list.

Remarks

To remove only one object from a collection, use the **Remove** method.

Example

This example adds six **Panel** objects to a **StatusBar** control, creating a total of seven **Panel** objects. A click on the form clears all **Panel** objects when their number reaches seven. If the number of **Panel** objects is less than seven, each click on the form will add a new **Panel** object to the control until the number seven is once again reached. To try the example, place a **StatusBar** control on a form and paste the code into the Declarations section. Run the example and click on the form to clear all **Panel** objects and subsequently add **Panel** objects.

```
Private Sub Form_Load()
   Dim pnlX As Panel   ' Declare object variable for Panel objects.
   Dim I As Integer

   ' Add 6 Panel objects to the single default Panel object,
   ' making 7 Panel objects.
   For I = 1 to 6
      Set pnlX = StatusBar1.Panels.Add
   Next I
End Sub

Private Sub Form_Click()
   ' If the Count of the collection is 7, then clear the collection.
   ' Otherwise, add one Panel and use the collection's Count property
   ' to set its Style.
   If StatusBar1.Panels.Count = 7 Then
      StatusBar1.Panels.Clear
   Else
      Dim pnlX As Panel
      Set pnlX = StatusBar1.Panels.Add( , , "simple", 0)
      ' The Style property is enumerated from 0 to 6. Use the Panels
      ' Count property -1 to set the Style property for the new Panel.
      ' Display all panels regardless of form width.
      pnlX.minwidth = TextWidth("simple")
      pnlX.AutoSize = sbrSpring
      pnlX.Style = Statusbar1.Panels.Count - 1
   End If
End Sub
```

See Also

Remove Method (ActiveX Controls)

Applies To: LightSources Collection

Drag Method (ActiveX Controls)

Begins, ends, or cancels a drag operation of any control except the **Line**, **Menu**, **Shape**, **Timer**, or **CommonDialog** controls. Doesn't support named arguments.

Syntax

object.**Drag** *action*

The **Drag** method syntax has these parts:

Part	Description
object	Required. An object expression that evaluates to an object in the Applies To list. If *object* is omitted, the object whose event procedure contains the **Drag** method is assumed.
action	Optional. A constant or value that specifies the action to perform, as described in Settings. If *action* is omitted, the default is to begin dragging the object.

Settings

The settings for *action* are:

Constant	Value	Description
*xxx***Cancel**	0	Cancels drag operation
*xxx***BeginDrag**	1	Begins dragging *object*
*xxx***EndDrag**	2	Ends dragging and drop *object*

Remarks

These constants are listed in the Visual Basic (VB) object library in the Object Browser.

Using the **Drag** method to control a drag-and-drop operation is required only when the **DragMode** property of the object is set to Manual (0). However, you can use **Drag** on an object whose **DragMode** property is set to Automatic (1 or **vbAutomatic**).

If you want the mouse pointer to change shape while the object is being dragged, use either the **DragIcon** or **MousePointer** property. The **MousePointer** property is only used if no **DragIcon** is specified.

The **Drag** method generally acts synchronously, meaning that subsequent statements aren't executed until the drag action is complete. It can, however, act asynchronously if the **DragMode** property for the control is set to Manual (0 or **vbManual**).

Example

This example uses the **Drag** method to drag the filename of a bitmap (.bmp) file to a picture box where the bitmap is displayed. To try this example, paste all of the code into the Declarations section of a form that contains **DriveListBox**, **DirListBox**, **FileListBox**, **PictureBox**, and **Label** controls. Use the default names for all of the controls. Size and position all controls so they can be easily seen and used. The size and position of the label is unimportant because it's changed at run time. When the program begins, you can browse your file system and load any bitmaps. Once you've located a bitmap that you want to display, click the filename of that bitmap, and drag it to the picture box.

```
Private Sub Form_Load ()
   Picture1.AutoSize = -1   ' Turn on AutoSize.
   Label1.Visible = 0   ' Make the label invisible.
   File1.Pattern = "*.BMP; *.ICO; *.WMF"   ' Set file patterns.
End Sub

Private Sub Dir1_Change ()   ' Any change in Dir1
   File1.Path = Dir1.Path   ' is reflected in File1.
End Sub

Private Sub Drive1_Change ()   ' Any change in Drive1
   Dir1.Path = Drive1.Drive   ' is reflected in Dir1.
End Sub

Private Sub File1_MouseDown (Button As Integer, Shift As Integer, X As Single, Y As Single)
```

```
   Dim DY   ' Declare variable.
   DY = TextHeight("A")   ' Get height of one line.
   Label1.Move File1.Left, File1.Top + Y - DY /2, File1.Width, DY
   Label1.Drag   ' Drag label outline.
End Sub

Private Sub Dir1_DragOver (Source As Control, X As Single, Y As Single, State As Integer)
   ' Change pointer to no drop.
   If State = 0 Then Source.MousePointer = 12
   ' Use default mouse pointer.
   If State = 1 Then Source.MousePointer = 0
End Sub

Private Sub Drive1_DragOver (Source As Control, X As Single, Y As Single, State As Integer)
   ' Change pointer to no drop.
   If State = 0 Then Source.MousePointer = 12
   ' Use default mouse pointer.
   If State = 1 Then Source.MousePointer = 0
End Sub

Private Sub Form_DragOver (Source As Control, X As Single, Y As Single, State As Integer)
   ' Change pointer to no drop.
   If State = 0 Then Source.MousePointer = 12
   ' Use default mouse pointer.
   If State = 1 Then Source.MousePointer = 0
End Sub

Private Sub File1_DragOver (Source As Control, X As Single, Y As Single, State As Integer)
   On Error Resume Next
   If State = 0 And Right$(File1.Filename,4) = ".ICO" Then
      Label1.DragIcon = LoadPicture(File1.Path + "\" + File1.Filename)
   If Err Then MsgBox "The icon file can't be loaded."
   ElseIf State = 1 Then
      Label1.DragIcon = LoadPicture ()   ' Use no drag icon.
   End If
End Sub

Private Sub Picture1_DragDrop (Source As Control, X As Single, Y As Single)
   On Error Resume Next
   Picture1.Picture = LoadPicture(File1.Path + "\" + File1.Filename)
   If Err Then MsgBox "The picture file can't be loaded."
End Sub
```

See Also

DragDrop Event (ActiveX Controls) | DragIcon Property (ActiveX Controls) | DragMode Property (ActiveX Controls) | DragOver Event (ActiveX Controls) | MousePointer Property (ActiveX Controls)

Applies To: Masked Edit Control | MSChart Control

Move Method (ActiveX Controls)

Moves a control. Doesn't support named arguments.

Syntax

object.**Move** *left*, *top*, *width*, *height*

The **Move** method syntax has these parts:

Part	Description
object	Optional. An object expression that evaluates to an object in the Applies To list. If *object* is omitted, the form with the focus is assumed to be *object*.
left	Required. Single-precision value indicating the horizontal coordinate (x-axis) for the left edge of *object*.
top	Optional. Single-precision value indicating the vertical coordinate (y-axis) for the top edge of *object*.
width	Optional. Single-precision value indicating the new width of *object*.
height	Optional. Single-precision value indicating the new height of *object*.

Remarks

Only the *left* argument is required. However, to specify any other arguments, you must specify all arguments that appear in the syntax before the argument you want to specify. For example, you can't specify *width* without specifying *left* and *top*. Any trailing arguments that are unspecified remain unchanged.

For forms and controls in a **Frame** control, the coordinate system is always in twips. Moving a form on the screen or moving a control in a **Frame** is always relative to the origin (0,0), which is the upper-left corner. When moving a control on a **Form** object or in a **PictureBox** (or an MDI child form on an **MDIForm** object), the coordinate system of the container object is used. The coordinate system or unit of measure is set with the **ScaleMode** property at design time. You can change the coordinate system at run time with the **Scale** method.

Example

This example uses the **Move** method to move a form around on the screen. To try this example, paste the code into the Declarations section of a form, and then press F5 and click the form.

```
Private Sub Form_Click ()
   Dim Inch, Msg   ' Declare variables.
   Msg = "Choose OK to resize and move this form by "
   Msg = Msg & "changing the value of properties."
   MsgBox Msg   ' Display message.
   Inch = 1440   ' Set inch in twips.
   Width = 4 * Inch   ' Set width.
   Height = 2 * Inch   ' Set height.
   Left = 0   ' Set left to origin.
   Top = 0   ' Set top to origin.
   Msg = "Now choose OK to resize and move this form "
   Msg = Msg & "using the Move method."
   MsgBox Msg   ' Display message.
   Move Screen.Width - 2 * Inch, Screen.Height - Inch, 2 * Inch, Inch
End Sub
```

See Also

Height, Width Properties (ActiveX Controls) | Left, Top Properties (ActiveX Controls)

Applies To: Masked Edit Control | MSChart Control

OLEDrag Method (ActiveX Controls)

Causes a component to initiate an OLE drag/drop operation.

Syntax

object.**OLEDrag**

The *object* placeholder represents an object expression that evaluates to an object in the Applies To list.

Remarks

When the **OLEDrag** method is called, the component's OLEStartDrag event occurs, allowing it to supply data to a target component.

See Also

Applies To: Masked Edit Control | MSChart Control

Refresh Method (ActiveX Controls)

Forces a complete repaint of a form or control.

Syntax

object.**Refresh**

The *object* placeholder represents an object expression that evaluates to an object in the Applies To list.

Remarks

Use the **Refresh** method when you want to:

- Completely display one form while another form loads.
- Update the contents of a file-system list box, such as a **FileListBox** control.
- Update the data structures of a **Data** control.

Refresh can't be used on MDI forms but can be used on MDI child forms. You can't use **Refresh** on **Menu** or **Timer** controls.

Generally, painting a form or control is handled automatically while no events are occurring. However, there may be situations where you want the form or control updated immediately. For example, if you use a file list box, a directory list box, or a drive list box to show the current status of the directory structure, you can use **Refresh** to update the list whenever a change is made to the directory structure.

You can use the **Refresh** method on a **Data** control to open or reopen the database (if the **DatabaseName**, **ReadOnly**, **Exclusive**, or **Connect** property settings have changed) and rebuild the dynaset in the control's **Recordset** property.

See Also

Applies To: Masked Edit Control | MSChart Control

Remove Method (ActiveX Controls)

Removes a specific member from a collection.

Syntax

object.**Remove** *index*

The **Remove** method syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
index	An integer or string that uniquely identifies the object within the collection. Use an integer to specify the value of the **Index** property; use a string to specify the value of the **Key** property.

Remarks

To remove all the members of a collection, use the **Clear** method.

Example

This example adds six **Panel** objects to a **StatusBar** control, creating a total of seven **Panel** objects. When you click on the form, the code checks to see how many **Panel** objects there are. If there is only one **Panel** object, the code adds six **Panel** objects. Otherwise, it removes the first panel. To try the example, place a **StatusBar** control on a form and paste the code into the Declarations section. Run the example and click on the form to remove one **Panel** object at a time, and subsequently add **Panel** objects.

```
Private Sub Form_Load()
   Dim pnlX As Panel  ' Declare object variable for Panel objects.
   Dim i As Integer

   ' Add 6 Panel objects to the single default Panel object,
   ' making 7 Panel objects.
   For i = 1 To 6
      Set pnlX = StatusBar1.Panels.Add(, , , i)
      pnlX.AutoSize = sbrSpring
   Next i
End Sub

Private Sub Form_Click()
   ' If the Count of the collection is 1, add 6 Panel objects.
   ' Otherwise, remove the first panel from the collection.
   If StatusBar1.Panels.Count = 1 Then
      Dim sbrX As Panel
      Dim i As Integer
      For i = 1 To 6 ' Each panel has its style set by i.
         Set sbrX = StatusBar1.Panels.Add(, , , i)
         sbrX.AutoSize = sbrSpring
      Next i
   Else ' Remove the first panel.
      StatusBar1.Panels.Remove 1
   End If
End Sub
```

See Also

Clear Method (ActiveX Controls) | Index Property (ActiveX Controls) | Key Property (ActiveX Controls)

Applies To: LightSources Collection

SetFocus Method (ActiveX Controls)

Moves the focus to the specified control or form.

Syntax

object.**SetFocus**

The *object* placeholder represents an object expression that evaluates to an object in the Applies To list.

Remarks

The object must be a **Form** object, **MDIForm** object, or control that can receive the focus. After invoking the **SetFocus** method, any user input is directed to the specified form or control.

You can only move the focus to a visible form or control. Because a form and controls on a form aren't visible until the form's Load event has finished, you can't use the **SetFocus** method to move the focus to the form being loaded in its own Load event unless you first use the **Show** method to show the form before the Form_Load event procedure is finished.

You also can't move the focus to a form or control if the **Enabled** property is set to **False**. If the **Enabled** property has been set to **False** at design time, you must first set it to **True** before it can receive the focus using the **SetFocus** method.

See Also

Enabled Property (ActiveX Controls)

Applies To: Masked Edit Control | MSChart Control

ShowWhatsThis Method (ActiveX Controls)

Displays a selected topic in a Help file using the What's This popup provided by Help in 32-bit Windows operating systems.

Syntax

object.**ShowWhatsThis**

The *object* placeholder represents an object expression that evaluates to an object in the Applies To list.

Remarks

The **ShowWhatsThis** method is very useful for providing context-sensitive Help from a context menu in your application. The method displays the topic identified by the **WhatsThisHelpID** property of the object specified in the syntax.

Example

This example displays the What's This Help topic for a **CommandButton** control by selecting a menu command from a context menu created for the button. Set the **WhatsThisHelp** property of the form to **True**. Place a **CommandButton** control on a form, create a menu using the Menu Editor with a top-level invisible item named mnuBtnContextMenu, and a sub-menu named `mnuBtnWhatsThis` with a caption of "What's This?".

```
Private ThisControl As Control

Private Sub Command1_MouseUp(Button As Integer, Shift As Integer, X As Single, Y As Single)
      If Button = vbRightButton Then
         Set ThisControl = Command1
         PopupMenu mnuBtnContextMenu
      End If
   Set ThisControl = Nothing
End Sub

Private Sub mnuBtnWhatsThis_Click()
   ThisControl.ShowWhatsThis
End Sub
```

See Also

WhatsThisHelpID Property (ActiveX Controls)

Applies To: Masked Edit Control | MSChart Control

ValidateControls Method (ActiveX Controls)

Ensures that the contents of the last control on the form are valid before exiting the form.

Syntax

object.**ValidateControls**

The *object* placeholder represents an object expression that evaluates to an object in the Applies To list. If *object* is omitted, the **Form** with the focus is assumed to be *object*.

See Also

Validate Event (ActiveX Controls) | ValidationError Event

Applies To: Masked Edit Control

ZOrder Method (ActiveX Controls)

Places a specified **MDIForm**, **Form**, or control at the front or back of the z-order within its graphical level. Doesn't support named arguments.

Syntax

object.**ZOrder** *position*

The **ZOrder** method syntax has these parts:

Part	Description
object	Optional. An object expression that evaluates to an object in the Applies To list. If *object* is omitted, the form with the focus is assumed to be *object*.
Position	Optional. Integer indicating the position of *object* relative to other instances of the same *object*. If position is 0 or omitted, *object* is positioned at the front of the z-order. If position is 1, *object* is positioned at the back of the z-order.

Remarks

The z-order of objects can be set at design time by choosing the Bring To Front or Send To Back menu command from the Edit menu.

Within an **MDIForm** object, **ZOrder** sends MDI child forms to either the front or the back of the MDI client area, depending on the value of *position*. For an **MDIForm** or **Form** object, **ZOrder** sends the form to either the front or the back of the screen, depending on the value of *position*. As a result, forms can be displayed in front of or behind other running applications.

Three graphical layers are associated with forms and containers. The back layer is the drawing space where the results of the graphics methods are displayed. Next is the middle layer where graphical objects and **Label** controls are displayed. The front layer is where all nongraphical controls like **CommandButton**, **CheckBox**, or **ListBox** are displayed. Anything contained in a layer closer to the front covers anything contained in the layer(s) behind it. **ZOrder** arranges objects only within the layer where the object is displayed.

See Also

Applies To: Masked Edit Control | MSChart Control

Events (ActiveX Controls)

Change Event (ActiveX Controls)

Click Event (ActiveX Controls)

DblClick Event (ActiveX Controls)

DragDrop Event (ActiveX Controls)

DragOver Event (ActiveX Controls)

Format Event (ActiveX Controls)

GotFocus Event (ActiveX Controls)

KeyDown, KeyUp Events (ActiveX Controls)

KeyPress Event (ActiveX Controls)

LostFocus Event (ActiveX Controls)

MouseDown, MouseUp Events (ActiveX Controls)

MouseMove Event (ActiveX Controls)

OLECompleteDrag Event (ActiveX Controls)

OLEDragDrop Event (ActiveX Controls)

OLEDragOver Event (ActiveX Controls)

OLEGiveFeedback Event (ActiveX Controls)

OLESetData Event (ActiveX Controls)

OLEStartDrag Event (ActiveX Controls)

Unformat Event (ActiveX Controls)

Validate Event (ActiveX Controls)

Change Event (ActiveX Controls)

Indicates the contents of a control or a property value have changed. How and when this event occurs varies with the control.

Syntax

Private Sub *object*_**Change(**[*index* **As Integer**]**)**

The Change event syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list .
index	An integer that uniquely identifies a control if it's in a control array.

Remarks

The Change event procedure can synchronize or coordinate data display among controls. For example, you can use a scroll bar's Change event procedure to update the scroll bar's **Value** property setting in a **TextBox** control. Or you can use a Change event procedure to display data and formulas in a work area and results in another area.

> **Note** A **Change** event procedure can sometimes cause a cascading event. This occurs when the control's Change event alters the control's contents, for example, by setting a property in code that determines the control's value, such as the **Text** property setting for a **TextBox** control.

To prevent a cascading event:

- If possible, avoid writing a Change event procedure for a control that alters that control's contents. If you do write such a procedure, be sure to set a flag that prevents further changes while the current change is in progress.
- Avoid creating two or more controls whose Change event procedures affect each other, for example, two **TextBox** controls that update each other during their Change events.
- Avoid using a **MsgBox** function or statement in this event for **HScrollBar** and **VScrollBar** controls.

See Also

Applies To: Masked Edit Control

Click Event (ActiveX Controls)

Occurs when the user presses and then releases a mouse button over an object. It can also occur when the value of a control is changed.

Syntax

Private Sub *object*_**Click(**[*index* **As Integer**]**)**

The Click event syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list .
index	An integer that uniquely identifies a control if it's in a control array.

Remarks

Typically, you attach a Click event procedure to a **CommandButton** control, **Menu** object, or **PictureBox** control to carry out commands and command-like actions. For the other applicable controls, use this event to trigger actions in response to a change in the control.

You can use a control's **Value** property to test the state of the control from code. Clicking a control generates MouseDown and MouseUp events in addition to the Click event. The order in which these three events occur varies from control to control. If the order of events is important in your application, test the control to determine the event order.

> **Note** To distinguish between the left, right, and middle mouse buttons, use the MouseDown and MouseUp events.

If there is code in the Click event, the DlbClick event will never trigger, because the Click event is the first event to trigger between the two. As a result, the mouse click is intercepted by the Click event, so the DblClick event doesn't occur.

See Also

Applies To: MSChart Control

DblClick Event (ActiveX Controls)

Occurs when the user presses and releases a mouse button and then presses and releases it again over an object.

Syntax

Private Sub *object*_**DblClick (***index* **As Integer)**

Part	Description
object	An object expression that evaluates to an object in the Applies To list .
index	Identifies the control if it's in a control array.

Remarks

The argument *Index* uniquely identifies a control if it's in a control array. You can use a DblClick event procedure for an implied action, such as double-clicking an icon to open a window or document. You can also use this type of procedure to carry out multiple steps with a single action, such as double-clicking to select an item in a list box and to close the dialog box.

To produce such shortcut effects in Visual Basic, you can use a DblClick event procedure for a list box or file list box in tandem with a default button — a **CommandButton** control with its **Default** property set to **True**. As part of the DblClick event procedure for the list box, you simply call the default button's Click event.

For those objects that receive Mouse events, the events occur in this order: MouseDown, MouseUp, Click, DblClick, and MouseUp.

If DblClick doesn't occur within the system's double-click time limit, the object recognizes another Click event. The double-click time limit may vary because the user can set the double-click speed in the Control Panel. When you're attaching procedures for these related events, be sure that their actions don't conflict. Controls that don't receive DblClick events may receive two clicks instead of a DblClick.

> **Note** To distinguish between the left, right, and middle mouse buttons, use the MouseDown and MouseUp events.

If there is code in the **Click** event, the **DlbClick** event will never be raised.

Example

This example displays a selected list item in a **TextBox** control when either a **CommandButton** control is clicked or a list item is double-clicked. To try this example, paste the code into the Declarations section of a **Form** object that contains a **ListBox** control, a **TextBox** control, and a **CommandButton** control. Then run the example and click the **CommandButton** control or double-click an item in the **ListBox** control.

```
Private Sub Form_Load ()
   List1.AddItem "John"   ' Add list box entries.
   List1.AddItem "Paul"
   List1.AddItem "George"
   List1.AddItem "Ringo"
End Sub

Private Sub List1_DblClick ()
   Command1.Value = True   ' Trigger Click event.
End Sub

Private Sub Command1_Click ()
   Text1.Text = List1.Text   ' Display selection.
End Sub
```

See Also

Click Event (ActiveX Controls) | MouseDown, MouseUp Events (ActiveX Controls)

Applies To: MSChart Control

DragDrop Event (ActiveX Controls)

Occurs when a drag-and-drop operation is completed as a result of dragging a control over an object and releasing the mouse button or using the **Drag** method with its *action* argument set to 2 (Drop).

Syntax

Private Sub Form_DragDrop(*source* **As Control**, *x* **As Single**, *y* **As Single)**

Private Sub MDIForm_DragDrop(*source* **As Control**, *x* **As Single**, *y* **As Single)**

Private Sub *object*_**DragDrop(**[*index* **As Integer**,]*source* **As Control**, *x* **As Single**, *y* **As Single)**

The DragDrop event syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
index	An integer that uniquely identifies a control if it's in a control array.
source	The control being dragged. You can include properties and methods in the event procedure with this argument — for example, `Source.Visible = 0`.
x, *y*	A number that specifies the current horizontal (*x*) and vertical (*y*) position of the mouse pointer within the target form or control. These coordinates are always expressed in terms of the target's coordinate system as set by the **ScaleHeight**, **ScaleWidth**, **ScaleLeft**, and **ScaleTop** properties.

Remarks

Use a DragDrop event procedure to control what happens after a drag operation is completed. For example, you can move the source control to a new location or copy a file from one location to another.

When multiple controls can potentially be used in a *source* argument:

- Use the **TypeOf** keyword with the **If** statement to determine the type of control used with *source*.
- Use the control's **Tag** property to identify a control, and then use a DragDrop event procedure.

> **Note** Use the **DragMode** property and **Drag** method to specify the way dragging is initiated. Once dragging has been initiated, you can handle events that precede a DragDrop event with a DragOver event procedure.

Example

This example demonstrates the visual effect of dropping a **PictureBox** control onto another **PictureBox** control. To try this example, paste the code into the Declarations section of a form that contains three **PictureBox** controls. Set the **DragMode** property for Picture1 and Picture2 to 1 (Automatic). Use the **Picture** property to assign bitmaps to Picture1 and Picture2, and then press F5 and drag Picture1 or Picture2 over Picture3.

```
Private Sub Picture3_DragDrop (Source As Control, X as Single, Y As Single)
   If TypeOf Source Is PictureBox Then
      ' Set Picture3 bitmap to same as source control.
      Picture3.Picture = Source.Picture
   End If
End Sub
```

See Also

Drag Method (ActiveX Controls) | DragIcon Property (ActiveX Controls) | DragMode Property (ActiveX Controls) | DragOver Event (ActiveX Controls) | MouseDown, MouseUp Events (ActiveX Controls) | MouseMove Event (ActiveX Controls)

Applies To: Masked Edit Control | MSChart Control

DragOver Event (ActiveX Controls)

Occurs when a drag-and-drop operation is in progress. You can use this event to monitor the mouse pointer as it enters, leaves, or rests directly over a valid target. The mouse pointer position determines the target object that receives this event.

Syntax

Private Sub Form_DragOver(*source* **As Control**, *x* **As Single**, *y* **As Single**, *state* **As Integer)**

Private Sub MDIForm_DragOver(*source* **As Control**, *x* **As Single**, *y* **As Single**, *state* **As Integer)**

Private Sub *object*_**DragOver(**[*index* **As Integer**,]*source* **As Control**, *x* **As Single**, *y* **As Single**, *state* **As Integer)**

The DragOver event syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
index	An integer that uniquely identifies a control if it's in a control array.
source	The control being dragged. You can refer to properties and methods in the event procedure with this argument — for example, `Source.Visible = False`.
x, *y*	A number that specifies the current horizontal (*x*) and vertical (*y*) position of the mouse pointer within the target form or control. These coordinates are always expressed in terms of the target's coordinate system as set by the **ScaleHeight**, **ScaleWidth**, **ScaleLeft**, and **ScaleTop** properties.
state	An integer that corresponds to the transition state of the control being dragged in relation to a target form or control:
	0 = Enter (source control is being dragged within the range of a target).
	1 = Leave (source control is being dragged out of the range of a target).
	2 = Over (source control has moved from one position in the target to another).

Remarks

Use a DragOver event procedure to determine what happens after dragging is initiated and before a control drops onto a target. For example, you can verify a valid target range by highlighting the target (set the **BackColor** or **ForeColor** property from code) or by displaying a special drag pointer (set the **DragIcon** or **MousePointer** property from code).

Use the *state* argument to determine actions at key transition points. For example, you might highlight a possible target when *state* is set to 0 (Enter) and restore the object's previous appearance when *state* is set to 1 (Leave).

When an object receives a DragOver event while *state* is set to 0 (Enter):

- If the source control is dropped on the object, that object receives a DragDrop event.
- If the source control isn't dropped on the object, that object receives another DragOver event when *state* is set to 1 (Leave).

> **Note** Use the **DragMode** property and **Drag** method to specify the way dragging is initiated. For suggested techniques with the *source* argument, see Remarks for the DragDrop event topic.

Example

This example demonstrates one way to indicate a valid drop target. The pointer changes from the default arrow to a special icon when a **TextBox** control is dragged over a **PictureBox** control. The pointer returns to the default when the source is dragged elsewhere. To try this example, paste the code into the Declarations section of a form that contains a small **TextBox** and a **PictureBox**. Set the **TextBox** control's **DragMode** property to 1, and then press F5 and drag the **TextBox** over the **PictureBox**.

```
Private Sub Picture1_DragOver (Source As Control, X As Single, Y As Single, State As Integer)
   Select Case State
      Case vbEnter
   ' Load icon.
         Source.DragIcon = LoadPicture("ICONS\ARROWS\POINT03.ICO")
      Case vbLeave
         Source.DragIcon = LoadPicture()   ' Unload icon.
   End Select
End Sub
```

```
Private Sub Picture1_DragDrop (Source As Control, X As Single, Y As Single)
   Source.DragIcon = LoadPicture()   ' Unload icon.
End Sub
```

See Also

Drag Method (ActiveX Controls) | DragDrop Event (ActiveX Controls) | DragIcon Property (ActiveX Controls) | DragMode Property (ActiveX Controls) | MouseDown, MouseUp Events (ActiveX Controls) | MouseMove Event (ActiveX Controls)

Applies To: Masked Edit Control | MSChart Control

Format Event (ActiveX Controls)

Occurs after the **StdDataFormat** object formats the value.

Syntax

Sub *object*_**Format**(**ByRef** *datavalue* **As StdDataValue**)

The Format event syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
datavalue	**StdDataValue** object.

Remarks

The Format event allows you to do formatting that the standard settings of the **StdDataFormat** object cannot accomplish.

See Also

Unformat Event (ActiveX Controls)

Applies To: Masked Edit Control

GotFocus Event (ActiveX Controls)

Occurs when an object receives the focus, either by user action, such as tabbing to or clicking the object, or by changing the focus in code using the **SetFocus** method. A form receives the focus only when all visible controls are disabled.

Syntax

Private Sub Form_GotFocus( )

Private Sub *object*_**GotFocus(**[*index* **As Integer**]**)**

The GotFocus event syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list .
index	An integer that uniquely identifies a control if it's in a control array.

Remarks

Typically, you use a GotFocus event procedure to specify the actions that occur when a control or form first receives the focus. For example, by attaching a GotFocus event procedure to each control on a form, you can guide the user by displaying brief instructions or status bar messages. You can also provide visual cues by enabling, disabling, or showing other controls that depend on the control that has the focus.

> **Note** An object can receive the focus only if its **Enabled** and **Visible** properties are set to **True**. To customize the keyboard interface in Visual Basic for moving the focus, set the tab order or specify access keys for controls on a form.

Example

This example displays a status bar message when a button in an **OptionButton** group gets the focus. To try this example, paste the code into the Declarations section of a form that contains two **OptionButton** controls and a **Label** control. Set the **Name** property for both **OptionButton** controls to OptionGroup, and then press F5 and click the **OptionButton** controls.

```
Private Sub Form_Load ()
   Label1.AutoSize = True
End Sub

Private Sub OptionGroup_GotFocus (Index As Integer)
   Select Case Index
      Case 0
         Label1.Caption = "Option 1 has the focus."
      Case 1
         Label1.Caption = "Option 2 has the focus."
   End Select
End Sub

Private Sub OptionGroup_LostFocus (Index As Integer)
   Label1.Caption = ""
End Sub
```

See Also

LostFocus Event (ActiveX Controls) | SetFocus Method (ActiveX Controls) | TabIndex Property (ActiveX Controls) | TabStop Property (ActiveX Controls)

Applies To: Masked Edit Control | MSChart Control

KeyDown, KeyUp Events (ActiveX Controls)

Occur when the user presses (KeyDown) or releases (KeyUp) a key while an object has the focus. (To interpret ANSI characters, use the **KeyPress** event.)

Syntax

Private Sub Form_KeyDown(*keycode* **As Integer**, *shift* **As Integer)**

Private Sub *object*_**KeyDown(**[*index* **As Integer**,]*keycode* **As Integer**, *shift* **As Integer)**

Private Sub Form_KeyUp(*keycode* **As Integer**, *shift* **As Integer)**

Private Sub *object*_**KeyUp(**[*index* **As Integer**,]*keycode* **As Integer**, *shift* **As Integer)**

The KeyDown and KeyUp event syntaxes have these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
index	An integer that uniquely identifies a control if it's in a control array.
keycode	A key code, such as **vbKeyF1** (the F1 key) or **vbKeyHome** (the HOME key). To specify key codes, use the constants in the Visual Basic (VB) object library in the Object Browser.
shift	An integer that corresponds to the state of the SHIFT, CTRL, and ALT keys at the time of the event. The *shift* argument is a bit field with the least-significant bits corresponding to the SHIFT key (bit 0), the CTRL key (bit 1), and the ALT key (bit 2). These bits correspond to the values 1, 2, and 4, respectively. Some, all, or none of the bits can be set, indicating that some, all, or none of the keys are pressed. For example, if both CTRL and ALT are pressed, the value of *shift* is 6.

Remarks

For both events, the object with the focus receives all keystrokes. A form can have the focus only if it has no visible and enabled controls. Although the KeyDown and KeyUp events can apply to most keys, they're most often used for:

- Extended character keys such as function keys.
- Navigation keys.
- Combinations of keys with standard keyboard modifiers.
- Distinguishing between the numeric keypad and regular number keys.

Use KeyDown and KeyUp event procedures if you need to respond to both the pressing and releasing of a key.

KeyDown and KeyUp aren't invoked for:

- The ENTER key if the form has a **CommandButton** control with the **Default** property set to **True**.
- The ESC key if the form has a **CommandButton** control with the **Cancel** property set to **True**.
- The TAB key.

KeyDown and KeyUp interpret the uppercase and lowercase of each character by means of two arguments: *keycode*, which indicates the physical key (thus returning A and a as the same key) and *shift*, which indicates the state of *shift*+*key* and therefore returns either A or a.

If you need to test for the *shift* argument, you can use the *shift* constants which define the bits within the argument. The constants have the following values:

Constant	Value	Description
vbShiftMask	1	SHIFT key bit mask.
VbCtrlMask	2	CTRL key bit mask.
VbAltMask	4	ALT key bit mask.

The constants act as bit masks that you can use to test for any combination of keys.

You test for a condition by first assigning each result to a temporary integer variable and then comparing *shift* to a bit mask. Use the **And** operator with the *shift* argument to test whether the condition is greater than 0, indicating that the modifier was pressed, as in this example:

```
ShiftDown = (Shift And vbShiftMask) > 0
```

In a procedure, you can test for any combination of conditions, as in this example:

```
If ShiftDown And CtrlDown Then
```

> **Note** If the **KeyPreview** property is set to **True**, a form receives these events before controls on the form receive the events. Use the **KeyPreview** property to create global keyboard-handling routines.

See Also

Applies To: Masked Edit Control | MSChart Control

KeyPress Event (ActiveX Controls)

Occurs when the user presses and releases an ANSI key.

Syntax

Private Sub Form_KeyPress(*keyascii* **As Integer)**

Private Sub *object*_**KeyPress(**[*index* **As Integer**,]*keyascii* **As Integer)**

The KeyPress event syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
index	An integer that uniquely identifies a control if it's in a control array.
keyascii	An integer that returns a standard numeric ANSI keycode. *Keyascii* is passed by reference; changing it send s a different character to the object. Changing *keyascii* to 0 cancels the keystroke so the object receives no c haracter.

Remarks

The object with the focus receives the event. A form can receive the event only if it has no visible and enabled controls or if the **KeyPreview** property is set to **True**. A KeyPress event can involve any printable keyboard character, the CTRL key combined with a character from the standard alphabet or one of a few special characters, and the ENTER or BACKSPACE key. A KeyPress event procedure is useful for intercepting keystrokes entered in a **TextBox** or **ComboBox** control. It enables you to immediately test keystrokes for validity or to format characters as they're typed. Changing the value of the *keyascii* argument changes the character displayed.

You can convert the *keyascii* argument into a character by using the expression:

```
Chr(KeyAscii)
```

You can then perform string operations and translate the character back to an ANSI number that the control can interpret by using the expression:

```
KeyAscii = Asc(char)
```

Use KeyDown and KeyUp event procedures to handle any keystroke not recognized by KeyPress, such as function keys, editing keys, navigation keys, and any combinations of these with keyboard modifiers. Unlike the KeyDown and KeyUp events, KeyPress doesn't indicate the physical state of the keyboard; instead, it passes a character.

KeyPress interprets the uppercase and lowercase of each character as separate key codes and, therefore, as two separate characters. KeyDown and KeyUp interpret the uppercase and lowercase of each character by means of two arguments: *keycode*, which indicates the physical key (thus returning A and a as the same key), and *shift*, which indicates the state of *shift+key* and therefore returns either A or a.

If the **KeyPreview** property is set to **True**, a form receives the event before controls on the form receive the event. Use the **KeyPreview** property to create global keyboard-handling routines.

> **Note** The ANSI number for the keyboard combination of CTRL+@ is 0. Because Visual Basic recognizes a *keyascii* value of 0 as a zero-length string (""), avoid using CTRL+@ in your applications.

See Also

Applies To: Masked Edit Control | MSChart Control

Key Code Constants (ActiveX Controls)

Key Codes

Constant	Value	Description
vbKeyLButton	&H1	Left mouse button
vbKeyRButton	&H2	Right mouse button
vbKeyCancel	&H3	CANCEL key
vbKeyMButton	&H4	Middle mouse button
vbKeyBack	&H8	BACKSPACE key
vbKeyTab	&H9	TAB key
vbKeyClear	&HC	CLEAR key
vbKeyReturn	&HD	ENTER key
vbKeyShift	&H10	SHIFT key
vbKeyControl	&H11	CTRL key
vbKeyMenu	&H12	MENU key
vbKeyPause	&H13	PAUSE key
vbKeyCapital	&H14	CAPS LOCK key
vbKeyEscape	&H1B	ESC key
vbKeySpace	&H20	SPACEBAR key
vbKeyPageUp	&H21	PAGE UP key
vbKeyPageDown	&H22	PAGE DOWN key
vbKeyEnd	&H23	END key
vbKeyHome	&H24	HOME key
vbKeyLeft	&H25	LEFT ARROW key
vbKeyUp	&H26	UP ARROW key
vbKeyRight	&H27	RIGHT ARROW key
vbKeyDown	&H28	DOWN ARROW key
vbKeySelect	&H29	SELECT key
vbKeyPrint	&H2A	PRINT SCREEN key
vbKeyExecute	&H2B	EXECUTE key
vbKeySnapshot	&H2C	SNAPSHOT key
vbKeyInsert	&H2D	INS key
vbKeyDelete	&H2E	DEL key
vbKeyHelp	&H2F	HELP key
vbKeyNumlock	&H90	NUM LOCK key

KeyA Through KeyZ Are the Same as Their ASCII Equivalents: 'A' Through 'Z'

Constant	Value	Description
vbKeyA	65	A key
vbKeyB	66	B key
vbKeyC	67	C key
vbKeyD	68	D key
vbKeyE	69	E key
vbKeyF	70	F key
vbKeyG	71	G key
vbKeyH	72	H key
vbKeyI	73	I key
vbKeyJ	74	J key
vbKeyK	75	K key
vbKeyL	76	L key
vbKeyM	77	M key

vbKeyN	78	N key
vbKeyO	79	O key
vbKeyP	80	P key
vbKeyQ	81	Q key
vbKeyR	82	R key
vbKeyS	83	S key
vbKeyT	84	T key
vbKeyU	85	U key
vbKeyV	86	V key
vbKeyW	87	W key
vbKeyX	88	X key
vbKeyY	89	Y key
vbKeyZ	90	Z key

Key0 Through Key9 Are the Same as Their ASCII Equivalents: '0' Through '9

Constant	Value	Description
vbKey0	48	0 key
vbKey1	49	1 key
vbKey2	50	2 key
vbKey3	51	3 key
vbKey4	52	4 key
vbKey5	53	5 key
vbKey6	54	6 key
vbKey7	55	7 key
vbKey8	56	8 key
vbKey9	57	9 key

Keys on the Numeric Keypad

Constant	Value	Description
vbKeyNumpad0	&H60	0 key
vbKeyNumpad1	&H61	1 key
vbKeyNumpad2	&H62	2 key
vbKeyNumpad3	&H63	3 key
vbKeyNumpad4	&H64	4 key
vbKeyNumpad5	&H65	5 key
vbKeyNumpad6	&H66	6 key
vbKeyNumpad7	&H67	7 key
vbKeyNumpad8	&H68	8 key
vbKeyNumpad9	&H69	9 key
vbKeyMultiply	&H6A	MULTIPLICATION SIGN (*) key
vbKeyAdd	&H6B	PLUS SIGN (+) key
vbKeySeparator	&H6C	ENTER (keypad) key
vbKeySubtract	&H6D	MINUS SIGN (-) key
vbKeyDecimal	&H6E	DECIMAL POINT(.) key
vbKeyDivide	&H6F	DIVISION SIGN (/) key

Function Keys

Constant	Value	Description
vbKeyF1	&H70	F1 key
vbKeyF2	&H71	F2 key
vbKeyF3	&H72	F3 key

vbKeyF4	&H73	F4 key
vbKeyF5	&H74	F5 key
vbKeyF6	&H75	F6 key
vbKeyF7	&H76	F7 key
vbKeyF8	&H77	F8 key
vbKeyF9	&H78	F9 key
vbKeyF10	&H79	F10 key
vbKeyF11	&H7A	F11 key
vbKeyF12	&H7B	F12 key
vbKeyF13	&H7C	F13 key
vbKeyF14	&H7D	F14 key
vbKeyF15	&H7E	F15 key
vbKeyF16	&H7F	F16 key

LostFocus Event (ActiveX Controls)

Occurs when an object loses the focus, either by user action, such as tabbing to or clicking another object, or by changing the focus in code using the **SetFocus** method.

Syntax

Private Sub Form_LostFocus( )

Private Sub *object*_**LostFocus(**[*index* **As Integer**]**)**

The LostFocus event syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list .
index	An integer that uniquely identifies a control if it's in a control array.

Remarks

A LostFocus event procedure is primarily useful for verification and validation updates. Using LostFocus can cause validation to take place as the user moves the focus from the control. Another use for this type of event procedure is enabling, disabling, hiding, and displaying other objects as in a GotFocus event procedure. You can also reverse or change conditions that you set up in the object's GotFocus event procedure.

If an .exe file built by Visual Basic displays a dialog box created by a .dll file also built in Visual Basic, the .exe file's form will get Deactivate and LostFocus events. This may be unexpected, because you should not get the Deactivate event:

- If the object is an out-of-process component.
- If the object isn't written in Visual Basic.
- In the development environment when calling a DLL built in Visual Basic.

Example

This example changes the color of a **TextBox** control when it receives or loses the focus (selected with the mouse or TAB key) and displays the appropriate text in the **Label** control. To try this example, paste the code into the Declarations section of a form that contains two **TextBox** controls and a **Label** control, and then press F5 and move the focus between Text1 and Text2.

```
Private Sub Text1_GotFocus ()
   ' Show focus with red.
   Text1.BackColor = RGB(255, 0, 0)
   Label1.Caption = "Text1 has the focus."
End Sub

Private Sub Text1_LostFocus ()
   ' Show loss of focus with blue.
   Text1.BackColor = RGB(0, 0, 255)
   Label1.Caption = "Text1 doesn't have the focus."
End Sub
```

See Also

GotFocus Event (ActiveX Controls) | SetFocus Method (ActiveX Controls) | TabIndex Property (ActiveX Controls) | TabStop Property (ActiveX Controls)

Applies To: Masked Edit Control | MSChart Control

LostFocus Event (ActiveX Controls)

Occurs when an object loses the focus, either by user action, such as tabbing to or clicking another object, or by changing the focus in code using the **SetFocus** method.

Syntax

Private Sub Form_LostFocus( )

Private Sub *object*_**LostFocus(**[*index* **As Integer**]**)**

The LostFocus event syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list .
index	An integer that uniquely identifies a control if it's in a control array.

Remarks

A LostFocus event procedure is primarily useful for verification and validation updates. Using LostFocus can cause validation to take place as the user moves the focus from the control. Another use for this type of event procedure is enabling, disabling, hiding, and displaying other objects as in a GotFocus event procedure. You can also reverse or change conditions that you set up in the object's GotFocus event procedure.

If an .exe file built by Visual Basic displays a dialog box created by a .dll file also built in Visual Basic, the .exe file's form will get Deactivate and LostFocus events. This may be unexpected, because you should not get the Deactivate event:

- If the object is an out-of-process component.
- If the object isn't written in Visual Basic.
- In the development environment when calling a DLL built in Visual Basic.

Example

This example changes the color of a **TextBox** control when it receives or loses the focus (selected with the mouse or TAB key) and displays the appropriate text in the **Label** control. To try this example, paste the code into the Declarations section of a form that contains two **TextBox** controls and a **Label** control, and then press F5 and move the focus between Text1 and Text2.

```
Private Sub Text1_GotFocus ()
   ' Show focus with red.
   Text1.BackColor = RGB(255, 0, 0)
   Label1.Caption = "Text1 has the focus."
End Sub

Private Sub Text1_LostFocus ()
   ' Show loss of focus with blue.
   Text1.BackColor = RGB(0, 0, 255)
   Label1.Caption = "Text1 doesn't have the focus."
End Sub
```

See Also

GotFocus Event (ActiveX Controls) | SetFocus Method (ActiveX Controls) | TabIndex Property (ActiveX Controls) | TabStop Property (ActiveX Controls)

Applies To: Masked Edit Control | MSChart Control

MouseDown, MouseUp Events (ActiveX Controls)

Occur when the user presses (MouseDown) or releases (MouseUp) a mouse button.

Syntax

Private Sub Form_MouseDown(*button* **As Integer**, *shift* **As Integer**, *x* **As Single**, *y* **As Single)**

Private Sub MDIForm_MouseDown(*button* **As Integer**, *shift* **As Integer**, *x* **As Single**, *y* **As Single)**

Private Sub *object*_**MouseDown(**[*index* **As Integer**,]*button* **As Integer**, *shift* **As Integer**, *x* **As Single**, *y* **As Single)**

Private Sub Form_MouseUp(*button* **As Integer**, *shift* **As Integer**, *x* **As Single**, *y* **As Single)**

Private Sub MDIForm_MouseUp(*button* **As Integer**, *shift* **As Integer**, *x* **As Single**, *y* **As Single)**

Private Sub *object* _**MouseUp(**[*index* **As Integer**,]*button* **As Integer**, *shift* **As Integer**, *x* **As Single**, *y* **As Single)**

The MouseDown and MouseUp event syntaxes have these parts:

Part	Description
object	Returns an object expression that evaluates to an object in the Applies To list.
index	Returns an integer that uniquely identifies a control if it's in a control array.
button	Returns an integer that identifies the button that was pressed (MouseDown) or released (MouseUp) to c ause the event. The *button* argument is a bit field with bits corresponding to the left button (bit 0), right button (bit 1), and middle button (bit 2). These bits correspond to the values 1, 2, and 4, respectively. Onl y one of the bits is set, indicating the button that caused the event.
shift	Returns an integer that corresponds to the state of the SHIFT, CTRL, and ALT keys when the button specif ied in the *button* argument is pressed or released. A bit is set if the key is down. The *shift* argument is a b it field with the least-significant bits corresponding to the SHIFT key (bit 0), the CTRL key (bit 1), and the ALT key (bit 2). These bits correspond to the values 1, 2, and 4, respectively. The *shift* argument indicates the state of these keys. Some, all, or none of the bits can be set, indicating that some, all, or none of the k eys are pressed. For example, if both CTRL and ALT were pressed, the value of *shift* would be 6.
x, *y*	Returns a number that specifies the current location of the mouse pointer. The *x* and *y* values are always expressed in terms of the coordinate system set by the **ScaleHeight**, **ScaleWidth**, **ScaleLeft**, and **Scale Top** properties of the object.

Remarks

Use a MouseDown or MouseUp event procedure to specify actions that will occur when a given mouse button is pressed or released. Unlike the Click and DblClick events, MouseDown and MouseUp events enable you to distinguish between the left, right, and middle mouse buttons. You can also write code for mouse-keyboard combinations that use the SHIFT, CTRL, and ALT keyboard modifiers.

The following applies to both Click and DblClick events:

- If a mouse button is pressed while the pointer is over a form or control, that object "captures" the mouse and receives all mouse events up to and including the last MouseUp event. This implies that the *x*, *y* mouse-pointer coordinates returned by a mouse event may not always be in the internal area of the object that receives them.
- If mouse buttons are pressed in succession, the object that captures the mouse after the first press receives all mouse events until all buttons are released.

If you need to test for the *button* or *shift* arguments, you can use constants listed in the Visual Basic (VB) object library in the Object Browser to define the bits within the argument:

Constant (Button)	Value	Description
vbLeftButton	1	Left button is pressed
vbRightButton	2	Right button is pressed
vbMiddleButton	4	Middle button is pressed

Constant (Shift)	Value	Description
vbShiftMask	1	SHIFT key is pressed.
vbCtrlMask	2	CTRL key is pressed.
vbAltMask	4	ALT key is pressed.

The constants then act as bit masks you can use to test for any combination of buttons without having to figure out the unique bit field value for each combination.

> **Note** You can use a MouseMove event procedure to respond to an event caused by moving the mouse. The *button* argument for MouseDown and MouseUp differs from the *button* argument used for MouseMove. For MouseDown and MouseUp, the *button* argument indicates exactly one button per event, whereas for MouseMove, it indicates the current state of all buttons.

See Also

Applies To: MSChart Control

MouseMove Event (ActiveX Controls)

Occurs when the user moves the mouse.

Syntax

Private Sub Form_MouseMove(*button* **As Integer**, *shift* **As Integer**, *x* **As Single**, *y* **As Single)**

Private Sub MDIForm_MouseMove(*button* **As Integer**, *shift* **As Integer**, *x* **As Single**, *y* **As Single)**

Private Sub *object*_**MouseMove(**[*index* **As Integer**,] *button* **As Integer**, *shift* **As Integer**, *x* **As Single**, *y* **As Single)**

The MouseMove event syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
index	An integer that uniquely identifies a control if it's in a control array.
button	An integer that corresponds to the state of the mouse buttons in which a bit is set if the button is down. The *button* argument is a bit field with bits corresponding to the left button (bit 0), right button (bit 1), and middle button (bit 2). These bits correspond to the values 1, 2, and 4, respectively. It indicates the complete state of the mouse buttons; some, all, or none of these three bits can be set, indicating that some, all, or none of the buttons are pressed.
shift	An integer that corresponds to the state of the SHIFT, CTRL, and ALT keys. A bit is set if the key is down. The *shift* argument is a bit field with the least-significant bits corresponding to the SHIFT key (bit 0), the CTRL key (bit 1), and the ALT key (bit 2). These bits correspond to the values 1, 2, and 4, respectively. The *shift* argument indicates the state of these keys. Some, all, or none of the bits can be set, indicating that some, all, or none of the keys are pressed. For example, if both CTRL and ALT were pressed, the value of *shift* would be 6.
x, *y*	A number that specifies the current location of the mouse pointer. The *x* and *y* values are always expressed in terms of the coordinate system set by the **ScaleHeight**, **ScaleWidth**, **ScaleLeft**, and **ScaleTop** properties of the object.

Remarks

The MouseMove event is generated continually as the mouse pointer moves across objects. Unless another object has captured the mouse, an object recognizes a MouseMove event whenever the mouse position is within its borders.

If you need to test for the *button* or *shift* arguments, you can use constants listed in the Visual Basic (VB) object library in the Object Browser to define the bits within the argument:

Constant (Button)	Value	Description
vbLeftButton	1	Left button is pressed.
vbRightButton	2	Right button is pressed.
vbMiddleButton	4	Middle button is pressed.

Constant (Shift)	Value	Description
vbShiftMask	1	SHIFT key is pressed.
vbCtrlMask	2	CTRL key is pressed.
vbAltMask	4	ALT key is pressed.

The constants then act as bit masks you can use to test for any combination of buttons without having to figure out the unique bit field value for each combination.

You test for a condition by first assigning each result to a temporary integer variable and then comparing the *button* or *shift* arguments to a bit mask. Use the **And** operator with each argument to test if the condition is greater than zero, indicating the key or button is pressed, as in this example:

```
LeftDown = (Button And vbLeftButton) > 0
CtrlDown = (Shift And vbCtrlMask) > 0
```

Then, in a procedure, you can test for any combination of conditions, as in this example:

```
If LeftDown And CtrlDown Then
```

> **Note** You can use MouseDown and MouseUp event procedures to respond to events caused by pressing and releasing mouse buttons.

The *button* argument for MouseMove differs from the *button* argument for MouseDown and MouseUp. For MouseMove, the *button* argument indicates the current state of all buttons; a single MouseMove event can indicate that some, all, or no buttons are pressed. For MouseDown and MouseUp, the *button* argument indicates exactly one button per event.

Any time you move a window inside a MouseMove event, it can cause a cascading event. MouseMove events are generated when the window moves underneath the pointer. A MouseMove event can be generated even if the mouse is perfectly stationary.

See Also

Applies To: MSChart Control

OLECompleteDrag Event (ActiveX Controls)

Occurs when a source component is dropped onto a target component, informing the source component that a drag action was either performed or canceled.

Syntax

Private Sub *object*_**OLECompleteDrag**([*effect* **As Long**])

The OLECompleteDrag event syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
effect	A long integer set by the source object identifying the action that has been performed, thus allowing the source to take appropriate action if the component was moved (such as the source deleting data if it is moved from one component to another). The possible values are listed in Settings.

Settings

The settings for *effect* are:

Constant	Value	Description
*xxx***DropEffectNone**	0	Drop target cannot accept the data, or the drop operation was canceled.
*xxx***DropEffectCopy**	1	Drop results in a copy of data from the source to the target. The original data is unaltered by the drag operation.
*xxx***DropEffectMove**	2	Drop results in data being moved from the drag source to the drop source. The drag source should remove the data from itself after the move.

Remarks

The **OLECompleteDrag** event is the final event to be called in an OLE drag/drop operation. This event informs the source component of the action that was performed when the object was dropped onto the target component. The target sets this value through the *effect* parameter of the **OLEDragDrop** event. Based on this, the source can then determine the appropriate action it needs to take. For example, if the object was moved into the target (*xxx***DropEffectMove**), the source needs to delete the object from itself after the move.

If **OLEDragMode** is set to **Automatic**, then Visual Basic handles the default behavior. The event still occurs, however, allowing the user to add to or change the behavior.

Most components support manual OLE drag and drop events, and some support automatic OLE drag and drop events.

See Also

Applies To: Masked Edit Control | MSChart Control

OLEDragDrop Event (ActiveX Controls)

Occurs when a source component is dropped onto a target component when the source component determines that a drop can occur.

Note This event occurs only if **OLEDropMode** is set to **1** (**Manual**).

Syntax

Private Sub *object*_**OLEDragDrop**(*data* **As DataObject**, *effect* **As Long**, *button* **As Integer**, *shift* **As Integer**, *x* **As Single**, *y* **As Single**)

The OLEDragDrop event syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
data	A **DataObject** object containing formats that the source will provide and, in addition, possibly the data for th ose formats. If no data is contained in the **DataObject**, it is provided when the control calls the **GetData** met hod. The **SetData** and **Clear** methods cannot be used here.
effect	A long integer set by the target component identifying the action that has been performed (if any), thus allowi ng the source to take appropriate action if the component was moved (such as the source deleting the data). T he possible values are listed in Settings.
button	An integer which acts as a bit field corresponding to the state of a mouse button when it is depressed. The left button is bit 0, the right button is bit 1, and the middle button is bit 2. These bits correspond to the values 1, 2, and 4, respectively. It indicates the state of the mouse buttons; some, all, or none of these three bits can be set, indicating that some, all, or none of the buttons are depressed.
shift	An integer which acts as a bit field corresponding to the state of the SHIFT, CTRL, and ALT keys when they are de pressed. The SHIFT key is bit 0, the CTRL key is bit 1, and the ALT key is bit 2. These bits correspond to the values 1, 2, and 4, respectively. The *shift* parameter indicates the state of these keys; some, all, or none of the bits can be set, indicating that some, all, or none of the keys are depressed. For example, if both the CTRL and ALT keys were depressed, the value of *shift* would be 6.
x,y	A number which specifies the current location of the mouse pointer. The *x* and *y* values are always expressed i n terms of the coordinate system set by the **ScaleHeight**, **ScaleWidth**, **ScaleLeft**, and **ScaleTop** properties of the object.

Settings

The settings for *effect* are:

Constant	Value	Description
*xxx***DropEffectNone**	0	Drop target cannot accept the data.
*xxx***DropEffectCopy**	1	Drop results in a copy of data from the source to the target. T he original data is unaltered by the drag operation.
*xxx***DropEffectMove**	2	Drop results in data being moved from drag source to drop s ource. The drag source should remove the data from itself aft er the move.
*xxx***DropEffectScroll**	-2147483648	A mask to indicate that the drop target window has scrolled/ would scroll.

Remarks

The source COM component should always mask values from the *effect* parameter to ensure compatibility with future implementations of COM components. Presently, only three of the 32 bits in the *effect* parameter are used. In future versions of Visual Basic, however, these other bits may be used. Therefore, as a precaution against future problems, drag sources and drop targets should mask these values appropriately before performing any comparisons.

For example, a source component should not compare an *effect* against, for example, **vbDropEffectCopy**, such as:

```
If Effect = mskDropEffectCopy...
```

Instead, the source component should mask the value or values being sought, as here:

```
If Effect And mskDropEffectCopy = mskDropEffectCopy...
```

-or-

```
If (Effect And mskDropEffectCopy)...
```

This allows for the definition of new drop effects in future versions of Visual Basic while preserving backwards compatibility with your existing code.

Most components support manual OLE drag and drop events, and some support automatic OLE drag and drop events.

See Also

OLECompleteDrag Event (ActiveX Controls) | OLEDragOver Event (ActiveX Controls) | OLEGiveFeedback Event (ActiveX Controls) | OLESetData Event (ActiveX Controls) | OLEStartDrag Event (ActiveX Controls)

Applies To: Masked Edit Control | MSChart Control

OLEDragDrop Constants

OLE Drag Constants (OLEDragMode Property)

Constant	Value	Description
*xxx***OLEDragManual**	0	OLE drag/drop will be initialized only under programmatic control.
*xxx***OLEDragAutomatic**	1	OLE drag/drop will be initialized when the user drags "out" of the control, or under programmatic control.

OLE Drop Constants (OLEDropMode Property)

*xxx***OLEDropAutomatic**	2	Accepts an OLE drag/drop without programmatic control.
*xxx***OLEDropManual**	1	Accepts an OLE drag/drop under programmatic control only.
*xxx***OLEDropNone**	0	Accepts no OLE drag/drop operations.

OLE DropEffect Constants (OLE DragDrop Event)

*xxx***OLEDropEffectCopy**	1	A mask to indicate that a copy has taken place/would take place.
*xxx***OLEDropEffectMove**	2	A mask to indicate that a move has taken place/would take place.
*xxx***OLEDropEffectNone**	0	No OLE drag/drop operation has taken place/would take place.
*xxx***OLEDropEffectScroll**	-2147483648	A mask to indicate that the drop target window has scrolled/would scroll.

See Also

OLEDragDrop Event (ActiveX Controls) | OLEDragMode Property (ActiveX Controls) | OLEDropMode Property (ActiveX Controls)

OLEDragOver Event (ActiveX Controls)

Occurs when one component is dragged over another.

Syntax

Private Sub *object*_**OLEDragOver**(*data* **As DataObject**, *effect* **As Long**, *button* **As Integer**, *shift* **As Integer**, *x* **As Single**, *y* **As Single**, *state* **As Integer**)

The OLEDragOver event syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
data	A **DataObject** object containing formats that the source will provide and, in addition, possibly the data for those formats. If no data is contained in the **DataObject**, it is provided when the control calls the **GetData** method. The **SetData** and **Clear** methods cannot be used here.
effect	A long integer initially set by the source object identifying all effects it supports. This parameter must be correctly set by the target component during this event. The value of *effect* is determined by logically **Or**'ing together all active effects (as listed in Settings). The target component should check these effects and other parameters to determine which actions are appropriate for it, and then set this parameter to one of the allowable effects (as specified by the source) to specify which actions will be performed if the user drops the selection on the component. The possible values are listed in Settings.
button	An integer which acts as a bit field corresponding to the state of a mouse button when it is depressed. The left button is bit 0, the right button is bit 1, and the middle button is bit 2. These bits correspond to the values 1, 2, and 4, respectively. It indicates the state of the mouse buttons; some, all, or none of these three bits can be set, indicating that some, all, or none of the buttons are depressed.
shift	An integer which acts as a bit field corresponding to the state of the SHIFT, CTRL, and ALT keys when they are depressed. The SHIFT key is bit 0, the CTRL key is bit 1, and the ALT key is bit 2. These bits correspond to the values 1, 2, and 4, respectively. The *shift* parameter indicates the state of these keys; some, all, or none of the bits can be set, indicating that some, all, or none of the keys are depressed. For example, if both the CTRL and ALT keys are depressed, the value of *shift* would be 6.
x,y	A number that specifies the current horizontal (*x*) and vertical (*y*) position of the mouse pointer within the target form or control. The *x* and *y* values are always expressed in terms of the coordinate system set by the **ScaleHeight**, **ScaleWidth**, **ScaleLeft**, and **ScaleTop** properties of the object.
state	An integer that corresponds to the transition state of the control being dragged in relation to a target form or control. The possible values are listed in Settings.

Settings

The settings for *effect* are:

Constant	Value	Description
*xxx***DropEffectNone**	0	Drop target cannot accept the data.
*xxx***DropEffectCopy**	1	Drop results in a copy of data from the source to the target. The original data is unaltered by the drag operation.
*xxx***DropEffectMove**	2	Drop results in data being moved from drag source to drop source. The drag source should remove the data from itself after the move.
*xxx***DropEffectScroll**	-2147483648 (&H80000000)	Scrolling is occurring or about to occur in the target component. This value is used in conjunction with the other values. **Note** Use only if you are performing your own scrolling in the target component.

The settings for *state* are:

Constant	Value	Description
*xxx***Enter**	0	Source component is being dragged within the range of a target.
*xxx***Leave**	1	Source component is being dragged out of the range of a target.

*xxx***Over**	2	Source component has moved from one position in the target to another .

Remarks

> **Note** If the *state* parameter is **vbLeave**, indicating that the mouse pointer has left the target, then the *x* and *y* parameters will contain zeros.

The source component should always mask values from the *effect* parameter to ensure compatibility with future implementations of COM components. Presently, only three of the 32 bits in the *effect* parameter are used. In future versions of Visual Basic, however, these other bits may be used. Therefore, as a precaution against future problems, drag sources and drop targets should mask these values appropriately before performing any comparisons.

For example, a source component should not compare an *effect* against, for example, **mskDropEffectCopy**, such as:

```
If Effect = mskDropEffectCopy...
```

Instead, the source component should mask for the value or values being sought, as here:

```
If Effect And mskDropEffectCopy = mskDropEffectCopy...
```

-or-

```
If (Effect And mskDropEffectCopy)...
```

This allows for the definition of new drop effects in future versions of Visual Basic while preserving backwards compatibility with your existing code.

Most components support manual OLE drag-and-drop events, and some support automatic OLE drag-and-drop events.

See Also

Applies To: Masked Edit Control | MSChart Control

OLEGiveFeedback Event (ActiveX Controls)

Occurs after every OLEDragOver event. OLEGiveFeedback allows the source component to provide visual feedback to the user, such as changing the mouse cursor to indicate what will happen if the user drops the object, or provide visual feedback on the selection (in the source component) to indicate what will happen.

Syntax

Private Sub *object*_**OLEGiveFeedback**(*effect* **As Long**, *defaultcursors* **As Boolean**)

The OLEGiveFeedback event syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
effect	A long integer set by the target component in the OLEDragOver event specifying the action to be perf ormed if the user drops the selection on it. This allows the source to take the appropriate action (such as giving visual feedback). The possible values are listed in Settings.
defaultcursors	A boolean value which determines whether Visual Basic uses the default mouse cursor provided by th e component, or uses a user-defined mouse cursor.
	True (default) = use default mouse cursor.
	False = do not use default cursor. Mouse cursor must be set with the **MousePointer** property of the **Screen** object.

Settings

The settings for *effect* are:

Constant	Value	Description
*xxx***DropEffectNone**	0	Drop target cannot accept the data.
*xxx***DropEffectCopy**	1	Drop results in a copy of data from the source to the target. T he original data is unaltered by the drag operation.
*xxx***DropEffectMove**	2	Drop results in data being moved from drag source to drop s ource. The drag source should remove the data from itself aft er the move.
*xxx***DropEffectScroll**	-2147483648 (&H80000000)	Scrolling is occurring or about to occur in the target compone nt. This value is used in conjunction with the other values. **Note** Use only if you are performing your own s crolling in the target component.

Remarks

If there is no code in the OLEGiveFeedback event, or if the *defaultcursors* parameter is set to **True**, then Visual Basic automatically sets the mouse cursor to the default cursor provided by the component.

The source component should always mask values from the *effect* parameter to ensure compatibility with future implementations of components. Presently, only three of the 32 bits in the *effect* parameter are used. In future versions of Visual Basic, however, these other bits may be used. Therefore, as a precaution against future problems, drag sources and drop targets should mask these values appropriately before performing any comparisons.

For example, a source component should not compare an *effect* against, for example, **mskDropEffectCopy**, such as:

```
If Effect = mskDropEffectCopy...
```

Instead, the source component should mask for the value or values being sought, as here:

```
If Effect And mskDropEffectCopy = mskDropEffectCopy...
```

-or-

```
If (Effect And mskDropEffectCopy)...
```

This allows for the definition of new drop effects in future versions of Visual Basic while preserving backwards compatibility with your existing code.

Most components support manual OLE drag and drop events, and some support automatic OLE drag and drop events.

See Also

Applies To: Masked Edit Control | MSChart Control

OLESetData Event (ActiveX Controls)

Occurs on a source component when a target component performs the **GetData** method on the source's **DataObject** object, but before the data for the specified format has been loaded.

Syntax

Private Sub *object*_**OLESetData**(*data* **As DataObject**, *dataformat* **As Integer**)

The OLESetData event syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
data	A **DataObject** object in which to place the requested data. The component calls the **SetData** method t o load the requested format.
dataformat	An integer specifying the format of the data that the target component is requesting. The source compo nent uses this value to determine what to load into the **DataObject** object.

Remarks

In certain cases, you may want to defer loading data into the **DataObject** object of a source component to save time, especially if the source component supports many formats. This event allows the source to respond to only one request for a given format of data. When this event is called, the source should check the *format* parameter to determine what needs to be loaded and then perform the **SetData** method on the **DataObject** object to load the data which is then passed back to the target component.

See Also

Applies To: Masked Edit Control | MSChart Control

OLEStartDrag Event (ActiveX Controls)

Occurs when a component's **OLEDrag** method is performed, or when a component initiates an OLE drag/drop operation when the **OLEDragMode** property is set to **Automatic**.

This event specifies the data formats and drop effects that the source component supports. It can also be used to insert data into the **DataObject** object.

Syntax

Private Sub *object*_**OLEStartDrag**(*data* **As DataObject**, *allowedeffects* **As Long**)

The OLEStartDrag event syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
data	A **DataObject** object containing formats that the source will provide and, optionally, the data for thos e formats. If no data is contained in the **DataObject**, it is provided when the control calls the **GetData** method, and you should provide the values for the *data* parameter. The **SetData** and **Clear** methods cannot be used here.
allowedeffects	A long integer containing the effects that the source component supports. The possible values are liste d in Settings.

Settings

The settings for *allowedeffects* are:

Constant	Value	Description
*xxx***DropEffectNone**	0	Drop target cannot accept the data.
*xxx***DropEffectCopy**	1	Drop results in a copy of data from the source to the target. The origin al data is unaltered by the drag operation.
*xxx***DropEffectMove**	2	Drop results in data being moved from drag source to drop source. Th e drag source should remove the data from itself after the move.

Remarks

The source component should logically **Or** together the supported values and place the result in the *allowedeffects* parameter. The target component can use this value to determine the appropriate action, and what the appropriate user feedback should be.

The StartDrag event also occurs if the component's **OLEDragMode** property is set to **Automatic**. This allows you to add formats and data to the **DataObject** object after the component has done so. You can also override the default behavior of the component by clearing the **DataObject** object (using the **Clear** method) and then adding your data and formats.

You may want to defer placing data into the **DataObject** object until the target component requests it. This allows the source component to save time by not loading multiple data formats. When the target performs the **GetData** method on the **DataObject**, the source's OLESetData event will occur if the requested data is not contained in the **DataObject**. At this point, the data can be loaded into the **DataObject**, which will in turn provide the data to the target.

If the user does not load any formats into the **DataObject**, then the drag/drop operation is canceled.

See Also

Applies To: Masked Edit Control | MSChart Control

Unformat Event (ActiveX Controls)

Occurs after the **StdFormat** object unformats the value.

Syntax

Sub *object*_**Unformat**(**ByRef** *datavalue* **As StdDataValue**)

The Unformat event syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
datavalue	**StdDataValue** object.

Remarks

This event allows the user to do unformatting that the standard settings of the **StdDataFormat** object cannot accomplish.

See Also

Format Event (ActiveX Controls)

Applies To: Masked Edit Control

Validate Event (ActiveX Controls)

Occurs before the focus shifts to a (second) control that has its **CausesValidation** property set to **True**.

Syntax

Private Sub *object*_**Validate(***KeepFocus* **As Boolean)**

The Validate event syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
KeepFocus	Value that determines if the control loses focus. Setting *KeepFocus* to **True** specifies that the control ret ains the focus.

Remarks

The Validate event works in tandem with the **CausesValidation** property to prevent a control from losing the focus until certain criteria are met.

> **Caution** The Validate event only occurs when the control which is about to receive the focus has its **CausesValidation** property set to **True**.

Example

The example uses three controls to demonstrate the use of the Validate event and **CausesValidation** property. By default, the **CausesValidation** property of the two TextBox controls are set to **True**. Thus when you try to shift the focus from one TextBox to the other, the Validate event occurs. If Text1 doesn't contain a date, or if Text2 doesn't contain a number larger than 10, the shift of focus is prevented. Because the **CausesValidation** property of the Command1 control is set to **False**, however, you can always click the Help button.

To try the example, place one **CommandButton** and two **TextBox** controls on a form. Paste the code into the Declarations section of the form and run the project. Attempt to shift the focus by pressing the Tab key.

```
Private Sub Form_Load()
   ' Set the button's CausesValidation property to False. When the user
   ' clicks the button, the Validate event does not occur.
   ' Set the Caption property of the button to "Help".
   With Command1
      .CausesValidation = False
      .Caption = "Help"
   End With

   Show
   With Text1 ' Select text of Text1 and set focus to it.
      .SelLength = Len(Text1.Text)
      .SetFocus
   End With
End Sub

Private Sub Command1_Click()
   ' Give the user help when the button is clicked.
   MsgBox _
   "Text1 must be set to a date." & VbCrLF & _
   "Text2 must be a number less than 10."
End Sub

Private Sub Text1_Validate(KeepFocus As Boolean)
   ' If the value is not a date, keep the focus, unless the user
   ' clicks Help.
   If Not IsDate(Text1.Text) Then
      KeepFocus = True
      MsgBox "Please insert a date in this field.", ,   "Text1"
   End if
```

```
End Sub

Private Sub Text2_Validate(KeepFocus As Boolean)
   ' If the value is a number larger than 10, keep the focus.
   If Not IsNumeric(Text2.Text) Or Val(Text2.Text) > 10 Then
      KeepFocus = True
MsgBox _
"Please insert a number less than or equal to 10.", , "Text2"
   End If
End Sub
```

See Also

CausesValidation Property (ActiveX Controls)

Applies To: Masked Edit Control | MSChart Control

Validate Event (ActiveX Controls)

Occurs before the focus shifts to a (second) control that has its **CausesValidation** property set to **True**.

Syntax

Private Sub *object*_**Validate(***KeepFocus* **As Boolean)**

The Validate event syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
KeepFocus	Value that determines if the control loses focus. Setting *KeepFocus* to **True** specifies that the control ret ains the focus.

Remarks

The Validate event works in tandem with the **CausesValidation** property to prevent a control from losing the focus until certain criteria are met.

> **Caution** The Validate event only occurs when the control which is about to receive the focus has its **CausesValidation** property set to **True**.

Example

The example uses three controls to demonstrate the use of the Validate event and **CausesValidation** property. By default, the **CausesValidation** property of the two TextBox controls are set to **True**. Thus when you try to shift the focus from one TextBox to the other, the Validate event occurs. If Text1 doesn't contain a date, or if Text2 doesn't contain a number larger than 10, the shift of focus is prevented. Because the **CausesValidation** property of the Command1 control is set to **False**, however, you can always click the Help button.

To try the example, place one **CommandButton** and two **TextBox** controls on a form. Paste the code into the Declarations section of the form and run the project. Attempt to shift the focus by pressing the Tab key.

```
Private Sub Form_Load()
   ' Set the button's CausesValidation property to False. When the user
   ' clicks the button, the Validate event does not occur.
   ' Set the Caption property of the button to "Help".
   With Command1
      .CausesValidation = False
      .Caption = "Help"
   End With

   Show
   With Text1 ' Select text of Text1 and set focus to it.
      .SelLength = Len(Text1.Text)
      .SetFocus
   End With
End Sub

Private Sub Command1_Click()
   ' Give the user help when the button is clicked.
   MsgBox _
   "Text1 must be set to a date." & VbCrLF & _
   "Text2 must be a number less than 10."
End Sub

Private Sub Text1_Validate(KeepFocus As Boolean)
   ' If the value is not a date, keep the focus, unless the user
   ' clicks Help.
   If Not IsDate(Text1.Text) Then
      KeepFocus = True
      MsgBox "Please insert a date in this field.", ,   "Text1"
   End if
```

```
End Sub

Private Sub Text2_Validate(KeepFocus As Boolean)
   ' If the value is a number larger than 10, keep the focus.
   If Not IsNumeric(Text2.Text) Or Val(Text2.Text) > 10 Then
      KeepFocus = True
MsgBox _
"Please insert a number less than or equal to 10.", , "Text2"
   End If
End Sub
```

See Also

CausesValidation Property (ActiveX Controls)

Applies To: Masked Edit Control | MSChart Control

Property Pages Dialog Box (ActiveX Controls)

Allows you to change a control's properties at design time.

Dialog Box Options

Tabs

Visual Basic creates a tabbed dialog box that acts like form by writing code to handle updating property values when a user changes values in the control.

You can add Property Pages to your project using the Add Property Pages command on the Project menu.

OK

Adds the Property Pages and closes the Property Pages dialog box.

Apply

Adds the Property Page without closing the dialog box.

Masked Edit Control

The **Masked Edit** control provides restricted data input as well as formatted data output. This control supplies visual cues about the type of data being entered or displayed. This is what the control looks like as an icon in the Toolbox:

File Name

MSMASK32.OCX

Class Name

MaskEdBox

Remarks

The **Masked Edit** control generally behaves as a standard text box control with enhancements for optional masked input and formatted output. If you do not use an input mask, the **Masked Edit** control behaves much like a standard text box.

If you define an input mask using the **Mask** property, each character position in the **Masked Edit** control maps to either a placeholder of a specified type or a literal character. Literal characters, or literals, can give visual cues about the type of data being used. For example, the parentheses surrounding the area code of a telephone number are literals: `(206)`.

If you attempt to enter a character that conflicts with the input mask, the control generates a ValidationError event. The input mask prevents you from entering invalid characters into the control.

The **Masked Edit** control has three bound properties: **DataChanged**, **DataField**, and **DataSource**. This means that it can be linked to a data control and display field values for the current record in the recordset. The **Masked Edit** control can also write out values to the recordset.

When the value of the field referenced by the **DataField** property is read, it is converted to a **Text** property string, if possible. If the recordset is updatable, the string is converted to the data type of the field.

To clear the **Text** property when you have a mask defined, you first need to set the **Mask** property to an empty string, and then the Ctl**Text** property to an empty string:

```
MaskedEdit1.Mask = ""
MaskedEdit1.Text = ""
```

When you define an input mask, the **Masked Edit** control behaves differently from the standard text box. The insertion point automatically skips over literals as you enter data or move the insertion point.

When you insert or delete a character, all nonliteral characters to the right of the insertion point are shifted, as necessary. If shifting these characters leads to a validation error, the insertion or deletion is prevented, and a ValidationError event is triggered.

Suppose the **Mask** property is defined as "?###", and the current value of the **Text** property is "A12." If you attempt to insert the letter "B" to the left of the letter "A," the "A" would shift to the right. Since the second value of the input mask requires a number, the letter "A" would cause the control to generate a ValidationError event.

The **Masked Edit** control also validates the values of the **Text** property at run time. If you set the **Text** property so that it conflicts with the input mask, the control generates a run-time error.

You may select text in the same way as for a standard text box control. When selected text is deleted, the control attempts to shift the remaining characters to the left of the selection.

Normally, when a selection in the **Masked Edit** control is copied onto the Clipboard, the entire selection, including literals, is transferred onto the Clipboard. You can use the **ClipMode** property to transfer only user-entered data onto the Clipboard — literal characters that are part of the input mask are not copied.

Example

In the example below, `medDate` is the masked edit control. `InitMedDate` is a function that could be called from the form Sub New.

> **Note** If you upgrade code from previous versions of Visual Basic, you will need to update the event declarations, which now include send and event args parameters.

```
Sub InitMedDate()
    With Me.medDate
        'This box will hold date information.
        .Mask = "##/##/##"
        .Text = "12/01/99"
        'The user likes bold Arial font at double the default.
        'Double the size of the control to fit.
        .FontName = "Arial"
        .FontBold = True
        .FontSize *= 2
        .Height *= 2
        .Width *= 2
    End With
End Sub

Public Sub medDate_Enter(ByVal sender As Object, ByVal e As System.EventArgs) Handles medDate
.Enter
    'The user prefers yellow on purple for entering text.
    Me.medDate.BackColor = Color.Purple
    Me.medDate.ForeColor = Color.Yellow
End Sub

Public Sub medDate_Leave(ByVal sender As Object, ByVal e As System.EventArgs) Handles medDate
.Leave
    'Set the colors back to the default after leaving the edit box.
    Me.medDate.BackColor = Me.medDate.DefaultBackColor
    Me.medDate.ForeColor = Me.medDate.DefaultForeColor
End Sub
```

See Also

Properties, Methods, and Events | Masked Edit Control Constants | ValidationError Event

Masked Edit Control Properties, Methods, and Events

Properties

AccessibleDescription Property

AccessibleName Property

AccessibleRole Property

AllowPrompt Property

Anchor Property

Appearance Property (ActiveX Controls)

AutoTab Property

BackColor, ForeColor Properties (ActiveX Controls)

BorderStyle Property (ActiveX Controls)

CausesValidation Property

ClipMode Property

ClipText Property

DataBindings Property

Dock Property

Enabled Property (ActiveX Controls)

Font Property (ActiveX Controls)

Format Property (Masked Edit Control)

FormattedText Property

Height, Width Properties (ActiveX Controls)

HideSelection Property (ActiveX Controls)

hWnd Property (ActiveX Controls)

ImeMode Property

Location Property

Left, Top Properties (ActiveX Controls)

Mask Property

MaxLength Property

MouseIcon Property (ActiveX Controls)

MousePointer Property (ActiveX Controls)

Name Property (ActiveX Controls)

OLEDragMode Property (ActiveX Controls)

OLEDropMode Property (ActiveX Controls)

Parent Property (ActiveX Controls)

PromptChar Property

PromptInclude Property

SelLength, SelStart, SelText Properties (ActiveX Controls)

SelText Property (Masked Edit Control)

Size Property

TabIndex Property

TabStop Property

Text Property (MaskedEdit Control)

Visible Property (ActiveX Controls)

Methods

AboutBox Method

OLEDrag Method (ActiveX Controls)

Refresh Method (ActiveX Controls)

Events

Change Event (ActiveX Controls)

KeyDown, KeyUp Events (ActiveX Controls)

KeyPress Event (ActiveX Controls)

OLECompleteDrag Event (ActiveX Controls)

OLEDragDrop Event (ActiveX Controls)

OLEDragOver Event (ActiveX Controls)

OLEGiveFeedback Event (ActiveX Controls)

OLESetData Event (ActiveX Controls)

OLEStartDrag Event (ActiveX Controls)

ValidationError Event

See Also

Applies To: Masked Edit Control

AllowPrompt Property

Determines whether or not the prompt character is a valid input character.

Syntax

[*form*.]**MaskedEdit**.**AllowPrompt** [= {**True** | **False**}]

Remarks

The **AllowPrompt** property settings are as follows:

Setting	Description
False	(Default) The prompt character is not a valid input character. A ValidationError event is triggered if you enter the prompt character.
True	The prompt character is a valid input character.

For example, suppose you have defined a prompt character of 0, and you want the **Masked Edit** control to accept five digits between 0 and 9. You specify a mask of #####. If the **AllowPrompt** property is **False** and you enter 0, a ValidationError event occurs. If **AllowPrompt** is set to **True**, you can enter 0 as a valid input character.

Data Type

Integer (Boolean)

See Also

Masked Edit Control Constants | PromptChar Property | ValidationError Event

Applies To: Masked Edit Control

AutoTab Property

Determines whether or not the next control in the tab order receives the focus as soon as the **Text** property of the **Masked Edit** control is filled with valid data. The **Mask** property determines whether the values in the **Text** property are valid.

Syntax

[*form*.]**MaskedEdit**.**AutoTab**[= {**True** | **False**}]

Remarks

Automatic tabbing occurs only if all the characters defined by the **Mask** property are entered into the control, the characters are valid, and the **AutoTab** property is set to **True.**

Setting	Description
False	(Default) **AutoTab** is not on. A ValidationError event occurs when you enter more characters than are defined by the input mask.
True	**AutoTab** is on. When you enter all the characters defined by the input mask, focus goes to the next control in the tab sequence, and all subsequent characters entered are handled by the next control.

The **Masked Edit** control is considered filled when you enter the last valid character in the control, regardless of where the character is in the input mask. This property has no effect if the **Mask** property is set to the empty string ("").

Data Type

Integer (Boolean)

See Also

Text Property (MaskedEdit Control) | ValidationError Event

Applies To: Masked Edit Control

ClipMode Property

Determines whether to include or exclude the literal characters in the input mask when doing a cut or copy command.

Syntax

[*form*.]**MaskedEdit**.**ClipMode** [= *setting%*]

Remarks

The following table lists the **ClipMode** property settings for the **Masked Edit** control.

Setting	Description
mskIncludeLiterals (0)	(Default) Include literals on a cut or copy command.
mskExcludeLiterals (1)	Exclude literals on a cut or copy command.

This property has no effect if the **Mask** property is set to the empty string ("").

Data Type

Integer (Enumerated)

See Also

Masked Edit Control Constants

Applies To: Masked Edit Control

ClipText Property

Returns the text in the **Masked Edit** control, excluding literal characters of the input mask. This property is not available at design time and is read-only at run time.

Syntax

[*form*.]**MaskedEdit**.**ClipText**

Remarks

This property acts the same as the **SelText** property when the **Mask** property is set to the empty string ("").

Data Type

String

See Also

SelText Property (Masked Edit Control)

Applies To: Masked Edit Control

Format Property (Masked Edit Control)

Specifies the format for displaying and printing numbers, dates, times, and text.

Syntax

[*form*.]**MaskedEdit**.**Format** [= *posformat$; negformat$; zeroformat$; nullformat$*]

Parameter	Description
posformat$	Expression used to display positive values.
negformat$	Expression used to display negative values.
zeroformat$	Expression used to display zero values.
nullformat$	Expression used to display null or empty values.

Remarks

The **Format** property defines the format expressions used to display the contents of the control. You can use the same format expressions as defined by the Visual Basic **Format** function, with the exception that named formats ("On/Off") cannot be used.

This property can have from one to four parameters separated by semicolons. If one of the parameters is not specified, the format specified by the first parameter is used. If multiple parameters appear, the appropriate number of separators must be used. For example, to specify *posformat$* and *nullformat$*, use the syntax

[*form*.]*MaskedEdit*.**Format** = *posformat$;;; nullformat$*

The following table shows examples of standard formats that you may wish to use.

Data type	Value	Description
Number	(Default) Empty string	General Numeric format. Displays as entered.
Number	$#,##0.00;($#,##0.00)	Currency format. Uses thousands separator; displays negative numbers enclosed in parentheses.
Number	0	Fixed number format. Displays at least one digit.
Number	#,##0	Commas format. Uses commas as thousands separator.
Number	0%	Percent format. Multiplies value by 100 and appends a perce nt sign.
Number	0.00E+00	Scientific format. Uses standard scientific notation.
Date/Time	(Default) c	General Date and Time format. Displays date, time, or both.
Date/Time	dddddd	Long Date format. Same as the Long Date setting in the Inter national section of the Microsoft Windows Control Panel. Exa mple: Tuesday, May 26, 1992.
Date/Time	dd-mmm-yy	Medium Date format. Example: 26-May-92.
Date/Time	ddddd	Short Date format. Same as the Short Date setting in the Inter national section of the Microsoft Windows Control Panel. Exa mple: 5/26/92.
Date/Time	ttttt	Long Time format. Same as the Time setting in the Internatio nal section of the Microsoft Windows Control Panel. Example: 05:36:17 A.M.
Date/Time	hh:mm AM/PM	Medium Time format. Example: 05:36 A.M.
Date/Time	hh:mm	Short Time format. Example: 05:36.

Data Type

String

See Also

Format Event (ActiveX Controls) | Format Function | Unformat Event (ActiveX Controls)

Applies To: Masked Edit Control

FormattedText Property

The **FormattedText** property returns the string displayed in the **Masked Edit** control when the control does not have the focus. This property is not available at design time, and is read-only at run time.

Syntax

[*form*.]**MaskedEdit**.**FormattedText**

Remarks

If the **Format** property is equal to the empty string (""), this property is identical to the **Text** property, except that it is read-only. If the **HideSelection** property is set to **False**, the control does not display the formatted text when it does not have the focus. However, the formatted text is still available through this property.

Data Type

String

See Also

HideSelection Property (ActiveX Controls) | Text Property (MaskedEdit Control)

Applies To: Masked Edit Control

Mask Property

Determines the input mask for the control.

Syntax

[*form*.]**MaskedEdit**.**Mask** [= *string$*]

Remarks

You can define input masks at both design time and run time. However, the following are examples of standard input masks that you may want to use at design time. The control can distinguish between numeric and alphabetic characters for validation, but cannot check for valid content, such as the correct month or time of day.

Mask	Description
Null String	(Default) No mask. Acts like a standard text box.
##-???-##	Medium date (US). Example: 20-May-92
##-##-##	Short date (US). Example: 05-20-92
##:## ??	Medium time. Example: 05:36 AM
##:##	Short time. Example: 17:23

The input mask can consist of the following characters.

Mask character	Description
#	Digit placeholder.
.	Decimal placeholder. The actual character used is the one specified as the decimal placeholder in your international settings. This character is treated as a literal for masking purposes.
,	Thousands separator. The actual character used is the one specified as the thousands separator in your international settings. This character is treated as a literal for masking purposes.
:	Time separator. The actual character used is the one specified as the time separator in your international settings. This character is treated as a literal for masking purposes.
/	Date separator. The actual character used is the one specified as the date separator in your international settings. This character is treated as a literal for masking purposes.
\	Treat the next character in the mask string as a literal. This allows you to include the '#', '&', 'A', and '?' characters in the mask. This character is treated as a literal for masking purposes.
&	Character placeholder. Valid values for this placeholder are ANSI characters in the following ranges: 32-126 and 128-255.
>	Convert all the characters that follow to uppercase.
<	Convert all the characters that follow to lowercase.
A	Alphanumeric character placeholder (entry required). For example: a – z, A – Z, or 0 – 9.
a	Alphanumeric character placeholder (entry optional).
9	Digit placeholder (entry optional). For example: 0 – 9.
C	Character or space placeholder (entry optional). This operates exactly like the & placeholder, and ensures compatibility with Microsoft Access.
?	Letter placeholder. For example: a – z or A – Z.
Literal	All other symbols are displayed as literals; that is, as themselves.

When the value of the **Mask** property is an empty string ("""), the control behaves like a standard text box control. When an input mask is defined, underscores appear beneath every placeholder in the mask. You can only replace a placeholder with a character that is of the same type as the one specified in the input mask. If you enter an invalid character, the masked edit control rejects the character and generates a ValidationError event.

> **Note** When you define an input mask for the **Masked Edit** control and you tab to another control, the ValidationError event is generated if there are any invalid characters in the **Masked Edit** control.

Data Type

String

See Also

Text Property (MaskedEdit Control) | ValidationError Event

Applies To: Masked Edit Control

MaxLength Property

Sets or returns the maximum length of the **Masked Edit** control.

Syntax

[*form*.]**MaskedEdit**.**MaxLength** [= *setting%*]

Remarks

The **Masked Edit** control can have a maximum of 64 characters (the valid range for this property is 1 to 64). The default value is set to 64 characters, including literal characters in the input mask.

If the user enters characters beyond the specified maximum length, the control generates a beep.

Data Type

Integer

See Also

Applies To: Masked Edit Control

PromptChar Property

Sets or returns the character used to prompt a user for input.

Syntax

[*form*.]**MaskedEdit**.**PromptChar** [= *char$*]

Remarks

The underscore character "_" is the default character value for the property. The **PromptChar** property can only be set to exactly one character.

Use the **PromptInclude** property to specify whether prompt characters are contained in the **Text** property.

Use the **AllowPrompt** property to test whether a prompt character is entered by a user.

Data Type

String

See Also

AllowPrompt Property | PromptInclude Property

Applies To: Masked Edit Control

PromptInclude Property

Specifies whether prompt characters are contained in the **Text** property value. Use the **PromptChar** property to change the value of the prompt character.

Syntax

[*form*.]**MaskedEdit**.**PromptInclude** [= { **True** | **False** }]

Remarks

The following table lists the **PromptInclude** property settings for the **Masked Edit** control.

Setting	Description
False	The value of the **Text** property does not contain any prompt character.
True	(Default) The value of the **Text** property contains prompt characters, if any .

If the **Masked Edit** control is bound to a data control, the **PromptInclude** property affects how the data control reads the bound **Text** property. If **PromptInclude** is **False**, the data control ignores any literals or prompt characters in the **Text** property. In this mode, the value that the data control retrieves from the **Masked Edit** control is equivalent to the value of the **ClipText** property.

If **PromptInclude** is **True**, the data control uses the value of the **Text** property as the data value to store.

Data Type

Integer (Boolean)

See Also

DataBinding Object (ActiveX Controls) | DataField Property (ActiveX Controls) | Text Property (MaskedEdit Control)

Applies To: Masked Edit Control

SelText Property (Masked Edit Control)

Sets or returns the text contained in the control.

Syntax

[*form*.]**MaskedEdit**.**SelText**[= *string$*]

Remarks

If an input mask is not defined for the **Masked Edit** control, the **SelText** property behaves like the standard **SelText** property for the **Text Box** control.

If an input mask is defined and there is selected text in the **Masked Edit** control, the **SelText** property returns a text string. Depending on the value of the **ClipMode** property, not all the characters in the selected text are returned. If **ClipMode** is on, literal characters do not appear in the returned string.

When the **SelText** property is set, the **Masked Edit** control behaves as if text was pasted from the Clipboard. This means that each character in *string$* is entered into the control as if the user typed it in.

Data Type

String

See Also

ClipMode Property | SelLength, SelStart, SelText Properties (ActiveX Controls)

Applies To: Masked Edit Control

Text Property (MaskedEdit Control)

Sets or returns the text contained in the control. This property is not available at design time.

Syntax

[*form*.]**MaskedEdit**.**Text**[= *string$*]

Remarks

This property sets and retrieves the text in the **Masked Edit** control, including literal characters and underscores that are part of the input mask. When setting the **Text** property, the *string$* value must match the characters in the input mask exactly, including literal characters and underscores.

> **Note** The **ClipMode** property setting has no effect on the value of the **Text** property.

The **SelText** property provides an easier way of setting the text in the **Masked Edit** control.

Data Type

Variant

See Also

ClipMode Property | FormattedText Property | SelText Property (Masked Edit Control)

Applies To: Masked Edit Control

ValidationError Event

Occurs when the **Masked Edit** control receives invalid input, as determined by the input mask.

Syntax

Private Sub *ctlname*_**ValidationError**(*InvalidText* **As String**; *StartPosition* **As Integer**)

Remarks

InvalidText is the value of the **Text** property, including the invalid character. This means that any placeholders and literal characters used in the input mask are included in *InvalidText*.

StartPosition is the position in *InvalidText* where the error occurred (the first invalid character).

See Also

AllowPrompt Property | AutoTab Property | Mask Property | Text Property (MaskedEdit Control) | ValidateControls Method (ActiveX Controls)

Applies To: Masked Edit Control

Masked Edit Control Constants

Appearance Constants

Constant	Value	Description
mskFlat	0	Flat. Paints controls and forms without visual effects.
mskThreeD	1	(Default) 3D. Paints controls with three-dimensional effec ts.

BorderStyle Constants

Constant	Value	Description
mskBSNone	0	No border
mskFixedSingle	1	Fixed single

ClipBoard Constants

Constant	Value	Description
mskCFRTF	&HFFFFBF01	Rich Text Format (.rtf file)
mskCFText	1	Text (.txt file)
mskCFBitmap	2	Bitmap (.bmp file)
mskCFMetafile	3	Metafile (.wmf file)
mskCFDIB	8	Device-independent bitma p
mskCFPalette	9	Color palette
mskCFEMetafile	14	Extended Metafile (.emf file)
mskCFFiles	15	Filename list

ClipMode Constants

Constant	Value	Description
mskIncludeLiterals	0	Include literals on cut or copy.
mskExcludeLiterals	1	Exclude literals on cut or copy .

DragOver Constants

Constant	Value	Description
mskEnter	0	Source control dragged into target.
mskLeave	1	Source control dragged out of target.
mskOver	2	Source control dragged from one position in target to an other.

Error Constants

Constant	Value	Description
mskDataObjectLocked	672	DataObject formats list may not be cleared or expande d outside of the OLEStartDrag event.
mskExpectedAnArgument	673	Expected at least one argument.
mskFormatNotByteArray	675	Non-intrinsic OLE drag and drop formats used with Set Data require Byte array data. GetData may return more bytes than were given to SetData.
mskGetNotSupported	394	Property is write-only.
mskInvalidObjectUse	425	Invalid object use.
mskInvalidProcedureCall	5	Invalid procedure call.

mskInvalidPropertyValue	380	Invalid property value.
mskRecursiveOleDrag	674	Illegal recursive invocation of OLE drag and drop.
mskSetNotSupported	383	Property is read-only.
mskWrongClipboardFormat	461	Specified format does not match format of data.

MousePointer Constants

Constant	Value	Description
mskDefault	0	Default
mskArrow	1	Arrow
mskCross	2	Crosshair
mskIbeam	3	I beam
mskIcon	4	Icon
mskSize	5	Size
mskSizeNESW	6	Size NE, SW mouse pointer
mskSizeNS	7	Size N, S mouse pointer
mskSizeNWSE	8	Size NW, SE mouse pointer
mskSizeEW	9	Size W, E mouse pointer
mskUpArrow	10	Up arrow
mskHourglass	11	Hourglass
mskNoDrop	12	No drop
mskArrowHourglass	13	Arrow and hourglass
mskArrowQuestion	14	Arrow and question mark
mskSizeAll	15	Size all
mskCustom	99	Custom icon specified by the **MouseIcon** propert y

OLEDrag Constants

Constant	Value	Description
mskOLEDragManual	0	OLE drag/drop is initialized only under programmatic con trol.
mskOLEDragAutomatic	1	OLE drag/drop is initialized when the user drags 'out' of t he control, or under programmatic control.

OLEDrop Constants

Constant	Value	Description
mskOLEDropNone	0	Accepts no OLE drag/drop operations.
mskOLEDropManual	1	Accepts an OLE drag/drop under programmatic control o nly.
mskOLEDropAutomatic	2	Accepts an OLE drag/drop without programmatic control.

OLEDropEffect Constants

Constant	Value	Description
mskOLEDropEffectNone	0	No OLE drag/drop operation has taken place/woul d take place.
mskOLEDropEffectCopy	1	A mask to indicate that a copy has taken place/wou ld take place.
mskOLEDropEffectMove	2	A mask to indicate that a move has take place/woul d take place.
mskOLEDropEffectScroll	&H80000000	A mask to indicate that the drop target window has scrolled/would scroll.

See Also

Appearance Property (ActiveX Controls) | BorderStyle Property (ActiveX Controls) | ClipMode Property | DragOver Event (ActiveX Controls) | Masked Edit Control | MousePointer Property (ActiveX Controls) | OLEDragMode Property (ActiveX Controls) | OLEDropMode Property (ActiveX Controls)

MSChart Control

A chart that graphically displays data.

Syntax

MSChart

Remarks

The **MSChart** control supports the following features:

- True three-dimensional representation.
- Support for all major chart types.
- Data grid population via random data and data arrays.

The **MSChart** control is associated with a data grid (**DataGrid** object). This data grid is a table that holds the data being charted. The data grid can also include labels used to identify series and categories on the chart. The person who designs your chart application fills the data grid with information by inserting data or by importing data from a spreadsheet or array.

Example

```
'In this example, the sales for two companies are shown for four months.
'Create the data array and bind it to the ChartData property.
   Dim Sales(,) As Object = New Object(, ) _
      {{"Company", "Company A", "Company B"}, _
      {"June", 20, 10}, _
      {"July", 10, 5}, _
      {"August", 30, 15}, _
      {"September", 14, 7}}
   chtSales.ChartData = Sales

   'Add a title and legend.
   With Me.chtSales
      .Title.Text = "Sales"
      .Legend.Location.LocationType = _
         MSChart20Lib.VtChLocationType.VtChLocationTypeBottom
      .Legend.Location.Visible = True
   End With

   'Add titles to the axes.
   With Me.chtSales.Plot
      .Axis(MSChart20Lib.VtChAxisId.VtChAxisIdX).AxisTitle.Text = "Year"
      .Axis(MSChart20Lib.VtChAxisId.VtChAxisIdY).AxisTitle.Text = "Millions of $"
   End With

   'Set custom colors for the bars.
   With Me.chtSales.Plot
      'Yellow for Company A
      ' -1 selects all the datapoints.
      .SeriesCollection(1).DataPoints(-1).Brush.FillColor.Set(250, 250, 0)
      'Purple for Company B
      .SeriesCollection(2).DataPoints(-1).Brush.FillColor.Set(200, 50, 200)
   End With
```

See Also

Properties, Methods, and Events | AngleUnits Constants | Axis Object | AxisGrid Object | AxisScale Object | AxisTitle Object | Backdrop Object | Brush Object | CategoryScale Object | Coor Object | DataGrid Object | DataPoint Object | DataPointLabel Object | Error Messages (MSChart Control) | Fill Object | Footnote Object | Frame Object | Intersection Object | Label Object | LCoor Object | Legend Object | Light Object | LightSource Object | Location Object | Marker Object | Pen Object | Plot Object | PlotBase Object | Rect Object | Series Object | SeriesMarker Object | SeriesPosition Object | Shadow Object | StatLine Object | TextLayout Object |

MSChart Control Properties, Methods, and Events

Properties

ActiveSeriesCount Property

AllowDithering Property

AllowDynamicRotation Property

AllowSelections Property

AllowSeriesSelection Property

AutoIncrement Property

Backdrop Property

BorderStyle Property (ActiveX Controls)

CausesValidation Property (ActiveX Controls)

Chart3d Property

ChartData Property

ChartType Property

Column Property (MSChart Control)

ColumnCount Property

ColumnLabel Property (MSChart)

ColumnLabelCount Property

ColumnLabelIndex Property

Container Property (ActiveX Controls)

Data Property (MSChart)

DataBindings Property (ActiveX Controls)

DataGrid Property

DataMember Property (ActiveX Controls)

DataSource Property (ActiveX Controls)

DoSetCursor Property

Drag Method (ActiveX Controls)

DragIcon Property (ActiveX Controls)

DragMode Property (ActiveX Controls)

DrawMode Property (MSChart)

Enabled Property (ActiveX Controls)

Footnote Property

FootnoteText Property

Height, Width Properties (ActiveX Controls)

HelpContextID Property (ActiveX Controls)

hWnd Property (ActiveX Controls)

Index Property (ActiveX Controls)

Left, Top Properties (ActiveX Controls)

Legend Property

MousePointer Property (MSChart)

Name Property (ActiveX Controls)

Object Property (ActiveX Controls)

OLEDragMode Property (ActiveX Controls)

OLEDropMode Property (ActiveX Controls)

Parent Property (ActiveX Controls)

Plot Property

RandomFill Property

Repaint Property

Row Property (MSChart)

RowCount Property (MSChart)

RowLabel Property (MSChart)

RowLabelCount Property

RowLabelIndex Property

SeriesColumn Property

SeriesType Property

ShowLegend Property

Stacking Property

TabIndex Property (ActiveX Controls)

TabStop Property (ActiveX Controls)

Tag Property (ActiveX Controls)

TextLengthType Property

Title Property (MSChart)

TitleText Property

ToolTipText Property (ActiveX Controls)

Visible Property (ActiveX Controls)

Visible Property (MSChart)

WhatsThisHelpID Property (ActiveX Controls)

Methods

Drag Method (ActiveX Controls)

EditCopy Method

EditPaste Method

GetSelectedPart Method

Layout Method

Move Method (ActiveX Controls)

OLEDrag Method (ActiveX Controls)

Refresh Method (ActiveX Controls)

SelectPart Method

SetFocus Method (ActiveX Controls)

ShowWhatsThis Method (ActiveX Controls)

ToDefaults Method

TwipsToChartPart Method

ZOrder Method (ActiveX Controls)

Events

AxisActivated Event

AxisLabelActivated Event

AxisLabelSelected Event

AxisLabelUpdated Event

AxisSelected Event

AxisTitleActivated Event

AxisTitleSelected Event

AxisTitleUpdated Event

AxisUpdated Event

ChartActivated Event

ChartSelected Event

ChartUpdated Event

Click Event (ActiveX Controls)

DataUpdated Event

DblClick Event (ActiveX Controls)

DonePainting Event

DragDrop Event (ActiveX Controls)

DragOver Event (ActiveX Controls)

FootnoteActivated Event

FootnoteSelected Event

FootnoteUpdated Event

GotFocus Event (ActiveX Controls)

KeyDown, KeyUp Events (ActiveX Controls)

KeyPress Event (ActiveX Controls)

LegendActivated Event

LegendSelected Event

LegendUpdated Event

LostFocus Event (ActiveX Controls)

MouseDown, MouseUp Events (ActiveX Controls)

MouseMove Event (ActiveX Controls)

OLECompleteDrag Event (ActiveX Controls)

OLEDragDrop Event (ActiveX Controls)

OLEDragOver Event (ActiveX Controls)

OLEGiveFeedback Event (ActiveX Controls)

OLESetData Event (ActiveX Controls)

OLEStartDrag Event (ActiveX Controls)

PlotActivated Event

PlotSelected Event

PlotUpdated Event

PointActivated Event

PointLabelActivated Event

PointLabelSelected Event

PointLabelUpdated Event

PointSelected Event

PointUpdated Event

SeriesActivated Event

SeriesSelected Event

SeriesUpdated Event

TitleActivated Event

TitleSelected Event

TitleUpdated Event

Validate Event (ActiveX Controls)

See Also

Applies To: MSChart Control

Error Messages (MSChart Control)

The **MSChart** control's errors that occur during program execution are handled like other errors. You must write your own error handling routine to trap and manage errors.

The following table lists the errors that can be trapped by exception handling.

Error Name	Explanation
GetNotSupported	Attempted to read a write-only property.
InvalidPropertyValue	Value is not appropriate for the property used.
SetNotSupported	Attempted to write to a read-only property.
VtChError	Function failed with a non-specific error.
VtChErrorInvalidAxis	One of the arguments in a method is a reference to an axis that does not exist on the chart.
VtChErrorInvalidHandle	Invalid chart handle passed.
VtChErrorInvalidSeriesNumber	One of the arguments in a method is a reference to a series that does not exis t on the chart.
VtChErrorNoData	Data necessary for the completion of the task is unavailable.
VtChErrorRestrictedVersion	The requested feature is not available in this version of the product.
VtErrorActionCanceled	User canceled operation. The user clicked outside the menu instead of selecti ng an item.
VtErrorArchivePointerMismatch	Tried to load a pointer into a reference.
VtErrorArchiveTypeMismatch	Encountered unexpected class type.
VtErrorArchiveVersion	Tried to load incompatible version.
VtErrorBufferTooSmall	String buffer is not large enough to hold the requested string. String has bee n truncated to fit the buffer.
VtErrorCannotOpenFile	Unable to open the requested file.
VtErrorCopyingObject	Failed to copy object.
VtErrorCorruptArchive	Persistent storage is corrupted.
VtErrorCorruptData	An internal error indicating that the **MSChart** control has encountered corru pt data and is unable to continue the requested action.
VtErrorCreateWindow	Could not create window.
VtErrorDefaultFontSubstituted	Default substituted for requested font.
VtErrorDeletingDeletedObject	An internal error indicating that the **MSChart** control attempted to delete an object that has already been deleted.
VtErrorDeletingUsedObject	An internal error indicating that the **MSChart** control attempted to delete an object that is still in use.
VtErrorDisplay	The **MSChart** control is unable to properly communicate with display drivers.
VtErrorDuplicateObject	Encountered duplicate object.
VtErrorDuplicateSymbol	Duplicate symbol.
VtErrorIconNotFound	Could not load icon.
VtErrorIdentifierTooBig	Identifier is too large.
VtErrorInvalidArgument	One of the function arguments was bad.
VtErrorInvalidFont	Requested font description is invalid.
VtErrorInvalidFontName	Requested font name is not available.
VtErrorInvalidIndex	Given index is out of bounds.
VtErrorInvalidObject	Object is uninitialized or invalid.
VtErrorInvalidRequest	The requested action is inappropriate or otherwise invalid.
VtErrorInvalidSyntax	Encountered invalid syntax.
VtErrorInvalidTypeConversion	Requested type conversion is not supported.
VtErrorLoadPicture	Could not load requested picture.
VtErrorNoDisplayResources	The **MSChart** control does not have enough display resources to properly dis play the chart.
VtErrorNoMemory	There is not enough memory to accomplish the requested action.
VtErrorNotFound	Cannot find the item for which the function is looking.
VtErrorNotImplemented	Feature is not yet implemented.

VtErrorOSVersion	Version of Windows is not supported.
VtErrorStreamIo	An error was encountered when trying to read or write a file.
VtErrorTooSmall	The returned result is longer than the return buffer size. A NULL string is placed in the buffer.
VtErrorUnableToLoadString	Unable to load a string; the string may have been too long.
VtErrorUnexpectedEOS	Encountered unexpected end of string.
VtErrorUnrecognizedFunction	Encountered unrecognized function.
VtErrorUserIo	An attempt to write a file to a user-created function failed.
VtFail	A general failure error.
VtOk	No error occurred.

See Also

Add Method (LightSources Collection) | MSChart Control

MSChart Control Objects Listed Alphabetically

Axis Object

AxisGrid Object

AxisScale Object

AxisTitle Object

Backdrop Object

Brush Object

CategoryScale Object

Coor Object

DataGrid Object

DataPoint Object

DataPointLabel Object

DataPoints Collection

Fill Object

Footnote Object

Frame Object

Intersection Object

Label Object

Labels Collection

LCoor Object

Legend Object

Light Object

LightSource Object

LightSources Collection

Location Object

Marker Object

Pen Object

Plot Object

PlotBase Object

Rect Object

Series Collection

Series Object

SeriesCollection Collection

SeriesMarker Object

SeriesPosition Object

Shadow Object

StatLine Object

TextLayout Object

Tick Object

Title Object

ValueScale Object

View3D Object

VtColor Object

VtFont Object

Wall Object

Weighting Object

Axis Object

Represents an axis on a chart.

Syntax

Axis

Remarks

Three axes are available: *x*, *y*, and *z*. The *z* axis is visible only when the chart is a 3D chart.

Example

The following example reads the number of label levels present on the *x* axis using the *x* **Axis** object.

```
Private Sub Command1_Click()
   Dim XAxis As Object
   Dim NumberOfLevels As Integer
   ' Read the number of label level present on the X
   ' Axis.
   Set XAxis = MSChart1.Plot.Axis(VtChAxisIdX, 1)
   NumberOfLevels = XAxis.LabelLevelCount
   MsgBox "Number of Label Levels = " _
   & Str(NumberOfLevels)
End Sub
```

See Also

Properties, Methods, and Events | AxisGrid Object | AxisScale Object | AxisTitle Object | CategoryScale Object | Intersection Object | Labels Collection

Axis Object Properties, Methods, and Events

Properties

AxisGrid Property

AxisScale Property

AxisTitle Property

CategoryScale Property

Intersection Property

LabelLevelCount Property

Labels Property

Pen Property

Tick Property

ValueScale Property

Methods

None

Events

None

See Also

Applies To: Axis Object

AxisGrid Object

Represents the planar area surrounding a chart axis.

Syntax

AxisGrid

Example

The following example changes the appearance of the y **AxisGrid** line style.

```
Private Sub Command1_Click()
   With MSChart1.Plot.Axis(VtChAxisIdY)
      .CategoryScale.Auto = False
      .ValueScale.MajorDivision = 10
      .ValueScale.MinorDivision = 5
      .AxisGrid.MajorPen.Style = VtPenStyleSolid
      .AxisGrid.MajorPen.VtColor.Green = 255
      .AxisGrid.MajorPen.Width = 2
      .AxisGrid.MinorPen.Style = VtPenStyleDashed
      .AxisGrid.MinorPen.VtColor.Red = 255
      .AxisGrid.MinorPen.Width = 1
      .AxisGrid.MinorPen.Cap = VtPenCapRound
   End With
End Sub
```

See Also

Properties, Methods, and Events

AxisGrid Object Properties, Methods, and Events

Properties

MajorPen Property

MinorPen Property

Methods

None

Events

None

See Also

Applies To: AxisGrid Object

AxisScale Object

Controls how chart values are plotted on an axis.

Syntax

AxisScale

Example

The following examples sets the *x* and *y* axes to a percent scale for a two-dimensional line chart. To try the example, draw an **MSChart** and a **CommandButton** control on a form. Paste the code into the **Form** object's code module and run the project.

```
Private Sub Command1_Click()
   ' Change both x and y axes to Percent scale for 2D
   ' Line chart.
   Dim axisID As VtChAxisId
   MSChart1.ChartType = VtChChartType2dLine
   For AxisId = VtChAxisIdX To VtChAxisIdY
      With MSChart1.Plot.Axis(AxisId).AxisScale
         .Type = VtChScaleTypePercent
         .PercentBasis = VtChPercentAxisBasisSumChart
      End With
   Next
End Sub
```

See Also

Properties, Methods, and Events | Axis Object | AxisScale Property

AxisScale Object Properties, Methods, and Events

Properties

Hide Property

LogBase Property

PercentBasis Property

Type Property (MSChart)

Methods

None

Events

None

See Also

Applies To: AxisScale Object

AxisTitle Object

Represents an axis title on a chart.

Syntax

AxisTitle

Example

The following example makes the axis title visible for all axes of a three-dimensional chart.

```
Private Sub Command1_Click()
   ' Makes Axis title visible for all axes of a
   ' 3D chart.
   Dim axisID As VtChAxisId
   MSChart1.chartType = VtChChartType3dBar
   For axisId = VtChAxisIdX To VtChAxisIdZ
      With MSChart1.Plot.axis(axisId, 1)
         .AxisTitle.VtFont.Size = 14
         .AxisTitle.Visible = True
         Select Case axisId
            Case 0
               .AxisTitle.text = "X Axis Title"
            Case 1
               .AxisTitle.text = "Y Axis Title"
            Case 2
               .AxisTitle.text = "2nd Y Axis Title"
            Case 3
               .AxisTitle.text = "Z Axis Title"
         End Select
      End With
   Next
End Sub
```

See Also

Properties, Methods, and Events | Axis Object | Backdrop Object

AxisTitle Object Properties, Methods, and Events

Properties

Backdrop Property

Font Property (MSChart)

Text Property (MSChart)

TextLayout Property

TextLength Property

Visible Property (MSChart)

VtFont Property

Methods

None

Events

None

See Also

Applies To: AxisTitle Object

Backdrop Object

Represents a shadow or pattern behind a chart element.

Syntax

Backdrop

Remarks

For the **Plot** object's **Backdrop** object, set the **Style** property to **VtFillStyleBrush** to see the effects.

See Also

Properties, Methods, and Events | Backdrop Property | Fill Object | Frame Object

Backdrop Object Properties, Methods, and Events

Properties

Fill Property

Frame Property

Shadow Property

Methods

None

Events

None

See Also

Applies To: Backdrop Object

Brush Object

The fill type used to display a chart element.

Syntax

Brush

Example

The following example sets a bold vertical line pattern for the chart backdrop using the **Brush** object.

```
Private Sub Command1_Click()
   ' Sets Backdrop to Fill - Brush Style.
   MSChart1.Backdrop.Fill.Style = VtFillStyleBrush
   ' Sets a pattern for the chart backdrop using the
   ' Brush object.
   With MSChart1.Backdrop.Fill.Brush
      .Style = VtBrushStylePattern
      .Index = VtBrushPatternBoldVertical
   ' Sets Pattern to Bold Vertical lines.
      .FillColor.Set 255, 0, 0   ' Fill Color = Red.
      .PatternColor.Set 0, 0, 255   ' Pattern Color =
                                 ' Blue.
   End With
End Sub
```

See Also

Properties, Methods, and Events | Brush Property | FillStyle Constants (MSChart Control)

Brush Object Properties, Methods, and Events

Properties

FillColor Property

Index Property (Brush)

PatternColor Property

Style Property (MSChart Control)

Methods

None

Events

None

See Also

Applies To: Brush Object

CategoryScale Object

The scale for a category axis.

Syntax

CategoryScale

Example

The following example sets the scaling attributes for a category axis.

```
Private Sub Command1_Click()
   ' Sets scaling attributes for a category axis.
   MSChart1.ChartType = VtChChartType2dLine
   With MSChart1.Plot.Axis(VtChAxisIdX)
      .ValueScale.MajorDivision = 10
      .ValueScale.MinorDivision = 5
      .CategoryScale.Auto = False          ' Sets manual scaling.
      .CategoryScale.DivisionsPerLabel = 2 ' Label appears every two
                                           ' divisions.
      .CategoryScale.DivisionsPerTick = 2  ' Ticks appear every two
                                           ' divisions.
      .CategoryScale.LabelTick = True   ' Labels displayed on top of
                                       ' Tick marks.
   End With
End Sub
```

See Also

Properties, Methods, and Events | CategoryScale Property

CategoryScale Object Properties, Methods, and Events

Properties

Auto Property (CategoryScale)

DivisionsPerLabel Property

DivisionsPerTick Property

LabelTick Property

Methods

None

Events

None

See Also

Applies To: CategoryScale Object

Coor Object

Describes a floating *x* and *y* coordinate pair for a chart.

Syntax

Coor

See Also

Properties, Methods, and Events | LCoor Object

Coor Object Properties, Methods, and Events

Properties

X Property

Y Property

Methods

Set Method (Coor, LCoor)

Events

None

See Also

Applies To: Coor Object

DataGrid Object

Represents a virtual matrix containing labels and data points for the **MSChart** control.

Syntax

DataGrid

Remarks

The **DataGrid** object is configured as rows and columns. You can add and subtract rows, columns, and labels to this matrix to change the appearance of the chart.

Example

The following example sets the chart parameters for a three-dimensional bar chart, fills the chart with random data and labels the data grid columns.

```
Private Sub Command1_Click()
   Dim rowLabelCount, columnLabelCount, rowCount As Integer
   Dim columnCount, labelIndex, Column, Row As Integer
   MSChart1.ChartType = VtChChartType3dBar
   With MSChart1.DataGrid
   ' Set Chart parameters using methods.
      rowLabelCount = 2
      columnLabelCount = 2
      rowCount = 6
      columnCount = 6
      .SetSize RowLabelCount, columnLabelCount, _
      RowCount, ColumnCount
      ' Randomly fill in the data.
      .RandomDataFill
      ' Then assign labels to second Level.
      labelIndex = 2
      column = 1
      .ColumnLabel(column, labelIndex) = "Product 1"
      column = 4
      .ColumnLabel(column, labelIndex) = "Product 2"
      row = 1
      .RowLabel(row, labelIndex) = "1994"
      row = 4
      .RowLabel(row, labelIndex) = "1995"
   End With
End Sub
```

See Also

Properties, Methods, and Events | DataGrid Property

DataGrid Object Properties, Methods, and Events

Properties

ColumnCount Property

ColumnLabel Property (DataGrid Object)

ColumnLabelCount Property

CompositeColumnLabel Property

CompositeRowLabel Property

RowCount Property (MSChart)

RowLabel Property (DataGrid Object)

RowLabelCount Property

Methods

DeleteColumnLabels Method

DeleteColumns Method

DeleteRowLabels Method

DeleteRows Method

GetData Method (MSChart)

InitializeLabels Method

InsertColumnLabels Method

InsertColumns Method

InsertRowLabels Method

InsertRows Method

MoveData Method

RandomDataFill Method

RandomFillColumns Method

RandomFillRows Method

SetData Method (MSChart)

SetSize Method

Events

None

See Also

Applies To: DataGrid Object

DataPoint Object

A member of the **DataPoints** collection that describes the attributes of an individual data point on a chart.

Syntax

DataPoint

Remarks

The **DataPoints** collection is accessed through the **SeriesCollection** object.

> **Caution** The DataPoints collection contains only one member at this time. To access it, you must use the –1, as shown below:

```
MSChart1.Plot.SeriesCollection(1).DataPoints(-1) _
.Brush.FillColor.Set 0, 255, 255
```

Example

The following example assigns a variable to a data point and sets the data point color and marker.

```
Private Sub Command1_Click()
   ' Change the color and marker of First DataPoint in
   ' the First Series.
   With MSChart1.Plot.SeriesCollection(1).DataPoints(-1)
      ' Change Data Point color to blue.
      .Brush.Style = VtBrushStyleSolid
      ' Set Color=Blue.
      .Brush.FillColor.Set 0, 255, 255
      ' Set DataPoint marker visible.
      .Marker.Visible = True
   End With
End Sub
```

See Also

Properties, Methods, and Events | Brush Object | DataPointLabel Object | DataPoints Collection | DataPoints Property

DataPoint Object Properties, Methods, and Events

Properties

Brush Property

DataPointLabel Property

EdgePen Property

Marker Property

Offset Property

Methods

ResetCustom Method

Select Method

Events

None

See Also

Applies To: DataPoint Object

DataPointLabel Object

The label for a data point on a chart.

Syntax

DataPointLabel

See Also

Properties, Methods, and Events | Coor Object | DataPointLabel Property

DataPointLabel Object Properties, Methods, and Events

Properties

Backdrop Property

Component Property

Custom Property

Font Property (MSChart)

LineStyle Property (DataPointLabel Object)

LocationType Property

Offset Property

PercentFormat Property

Text Property (MSChart)

TextLayout Property

TextLength Property

ValueFormat Property

VtFont Property

Methods

ResetCustomLabel Method

Select Method

Events

None

See Also

Applies To: DataPointLabel Object

DataPoints Collection

A group of chart data points.

Syntax

object.**DataPoints.**(*index*)

The **DataPoints** collection syntax has these parts:

Part	Description
object	An object expression that evaluates to an object the Applies To list.
index	Identifies a specific data point within the current series. For this version of the chart, –1 is the only valid valu e for this argument. This allows you two make changes to the default settings for all data points in the series. Settings cannot be changed for individual data points within the series.

Example

The following example sets the markers for a data point in a chart.

```
Private Sub Command1_Click()
   With MSChart1.Plot.SeriesCollection(1).DataPoints(-1)
   ' Set DataPoint marker visible.
      .Marker.Visible = True
      .Marker.Style = VtMarkerStyleDiamond
   End With
End Sub
```

The **Datapoints** collection **Item** method takes a special –1 argument to designate default properties for all data points of the series. The **Datapoint** object returned by **Item**(–1) can be manipulated just like a normal **Datapoint** object. A property set on this default data point affects every data point in the series, except those data points that have had that property set individually. In the example, the data point label for each series is set at three different locations. It makes no difference whether individual data point settings come before or after default settings; the individual settings always override the default settings. The **ResetCustom** method can be used to remove any individual settings for a data point and cause it to use the default settings for the series.

```
Private Sub Command1_Click()
   With MSChart1.Plot
      .SeriesCollection(1).DataPoints.Item(-1) _
      .DataPointLabel.LocationType = VtChLabelLocationTypeAbovePoint

      .SeriesCollection(2).DataPoints.Item(-1) _
      .DataPointLabel.LocationType = VtChLabelLocationTypeCenter

      .SeriesCollection(3).DataPoints.Item(-1) _
      .DataPointLabel.LocationType = VtChLabelLocationTypeBase
   End With
End Sub
```

See Also

Properties, Methods, and Events | DataPoint Object | DataPoints Property

DataPoints Collection Properties, Methods, and Events

Properties

Count Property (ActiveX Controls)

Item Property (MSChart)

Methods

None

Events

None

See Also

Applies To: DataPoints Collection

Fill Object

Describes the type and appearance of an object's backdrop in a chart.

Syntax

Fill

Example

The following example sets a gradient backdrop for a chart using the **Fill** object.

```
Private Sub Command1_Click()
   With MSChart1.backdrop.Fill
      ' Set a brush pattern backdrop.
      .Style = VtFillStyleBrush
      .Brush.Style = VtBrushPattern50Percent
   End With
End Sub
```

See Also

Properties, Methods, and Events | Brush Object | Fill Property | FillStyle Constants (MSChart Control)

Fill Object Properties, Methods, and Events

Properties

Brush Property

Style Property (MSChart Control)

Methods

None

Events

None

See Also

Applies To: Fill Object

Footnote Object

Descriptive text that appears beneath a chart.

Syntax

Footnote

Example

The following example sets the footnote location, text and color for a chart.

```
Private Sub Command1_Click()

   With MSChart1.Footnote
      ' Make Footnote Visible.
      .Location.Visible = True
      .Location.LocationType = _
      VtChLocationTypeBottomLeft

      ' Set Footnote properties.
      .text = "Chart Footnote"
      .VtFont.VtColor.Set 255, 0, 0
   End With

End Sub
```

See Also

Properties, Methods, and Events | Backdrop Object | Footnote Property

Footnote Object Properties, Methods, and Events

Properties

Backdrop Property

Font Property (MSChart)

Location Property

Text Property (MSChart)

TextLayout Property

TextLength Property

VtFont Property

Methods

Select Method

Events

None

See Also

Applies To: Footnote Object

Frame Object

Holds information about the appearance of the frame around a chart element.

Syntax

Frame

Example

The following example sets a blue, double-line frame on a chart backdrop.

```
Private Sub Command1_Click()
   With MSChart1.backdrop.Frame
      .Style = VtFrameStyleDoubleLine
      .Width = 2
      .FrameColor.Set 0, 0, 255   ' Blue frame.
      .SpaceColor.Set 255, 0, 0   ' Red spacing.
   End With
End Sub
```

See Also

Properties, Methods, and Events | Frame Property

Frame Object Properties, Methods, and Events

Properties

FrameColor Property

SpaceColor Property

Style Property (MSChart Control)

Width Property (MSChart)

Methods

None

Events

None

See Also

Applies To: Frame Object

Intersection Object

The point at which an axis intersects an intersecting axis on a chart.

Syntax

Intersection

Example

The following example sets manual intersection position properties and displays labels with the axis.

```
Private Sub Command1_Click()
      With MSChart1.Plot.Axis(VtChAxisIdX).Intersection
      ' Set Intersection Properties.
      .Auto = False   ' Set positioning to manual.
      .Point = 20      ' Set intersection with the Y
                     ' Axis to 20.
      .LabelsInsidePlot = True   ' Display Labels
                                 ' with Axis not at
                                 ' the base.
   End With
End Sub
```

See Also

Properties, Methods, and Events | Intersection Property

Intersection Object Properties, Methods, and Events

Properties

Auto Property (Intersection)

AxisId Property

Index Property (Intersection)

LabelsInsidePlot Property

Point Property

Methods

None

Events

None

See Also

Applies To: Intersection Object

Label Object

An item within a **Labels** collection that describes a specific chart axis label.

Syntax

axis.**Label**

See Also

Properties, Methods, and Events | Labels Collection | Labels Property

Label Object Properties, Methods, and Events

Properties

Auto Property (Label)

Backdrop Property

Font Property (MSChart)

Format Property (Label Object)

FormatLength Property

Standing Property

TextLayout Property

VtFont Property

Methods

None

Events

None

See Also

Applies To: Label Object

Labels Collection

A group of chart axis labels.

Syntax

axis.**Labels**(*index*)

The **Labels** collection syntax has these parts:

Part	Description
axis	An **Axis** object.
index	Identifies a specific axis label (**Label** object)within the current collection .

See Also

Properties, Methods, and Events | Label Object | Labels Property

Labels Collection Properties, Methods, and Events

Properties

Count Property (ActiveX Controls)

Item Property (MSChart)

Methods

None

Events

None

See Also

Applies To: Labels Collection

LCoor Object

Describes a long integer *x* and *y* coordinate pair.

Syntax

Lcoor

See Also

Properties, Methods, and Events | Coor Object

LCoor Object Properties, Methods, and Events

Properties

X Property

Y Property

Methods

Set Method (Coor, LCoor)

Events

None

See Also

Applies To: LCoor Object

Legend Object

Represents the graphical key and accompanying text that describes a chart series.

Syntax

Legend

Example

The following example sets the text and backdrop parameters for a chart legend.

```
Private Sub Command1_Click()
   With MSChart1.Legend
      ' Make Legend Visible.
      .Location.Visible = True
      .Location.LocationType = VtChLocationTypeRight
      ' Set Legend properties.
      .TextLayout.HorzAlignment = _
      VtHorizontalAlignmentRight   ' Right justify.
      ' Use Yellow text.
      .VtFont.VtColor.Set 255, 255, 0
      .Backdrop.Fill.Style = VtFillStyleBrush
      .Backdrop.Fill.Brush.Style = VtBrushStyleSolid
      .Backdrop.Fill.Brush.FillColor.Set 255, 0, 255
   End With
End Sub
```

See Also

Properties, Methods, and Events | Backdrop Object | Legend Property

Legend Object Properties, Methods, and Events

Properties

Backdrop Property

Font Property (MSChart)

Location Property

TextLayout Property

VtFont Property

Methods

Select Method

Events

None

See Also

Applies To: Legend Object

Light Object

Represents the light source illuminating a three-dimensional chart.

Syntax

Light

Example

The following example sets the ambient light and edge lighting intensity for a chart. To try the example, draw an **MSChart** and two **ComboBox** controls on a form. Paste the code into the **Form** object's code module and run the project. Click the ComboBox controls to see the change of ambient light and edge intensity.

```
Private Sub Form_Load()
    ' Configure the chart.
   With MSChart1
      .Title = "Hold down Control and mousedown on chart"
      .chartType = VtChChartType3dBar
      .Plot.Light.AmbientIntensity = 1   ' 100 % Intensity.
      .Plot.Light.EdgeIntensity = 0.5        ' 50 % Intensity.
      .Plot.Light.EdgeVisible = True
   End With
   'Configure ComboBoxe controls.
   ConfigCombo Combo1
   ConfigCombo Combo2
End Sub

Private Sub ConfigCombo(cmb As ComboBox)
   ' Populate a combobox with values.
   Dim i As Single
   For i = 0 To 1 Step 0.1
      cmb.AddItem i
   Next i
   cmb.ListIndex = 0
End Sub

Private Sub Combo1_Click()
   MSChart1.Plot.Light.AmbientIntensity = Combo1.Text
End Sub

Private Sub Combo2_Click()
   MSChart1.Plot.Light.EdgeIntensity = Combo2.Text
End Sub
```

See Also

Properties, Methods, and Events | LightSource Object | LightSources Collection | Z Property

Light Object Properties, Methods, and Events

Properties

AmbientIntensity Property

EdgeIntensity Property

EdgeVisible Property

LightSources Property

Methods

Set Method (Coor, LCoor)

Events

None

See Also

Applies To: Light Object

LightSource Object

Represents the light source used to illuminate elements in a three-dimensional chart.

Syntax

LightSource

See Also

Properties, Methods, and Events

LightSource Object Properties, Methods, and Events

Properties

Intensity Property

X Property

Y Property

Z Property

Methods

Set Method (LightSource)

Events

None

See Also

Applies To: LightSource Object

LightSources Collection

A group of **LightSource** objects in a chart.

Syntax

object.**LightSources**(*index*)

The **LightSources** collection syntax has the following parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
index	Integer. A number that uniquely identifies a member of the collection.

Example

The following example sets the coordinates and intensity for a light source, then adds a light source from a chart. To try the example, draw an **MSChart**, and two **CommandButton** controls on a form. Paste the code into the **Form** object's code module and run the project. Click the buttons to change the light sources.

```
Option Explicit

Private Sub Form_Load()
   ' Configure the chart and its light source.
   With MSChart1
      .chartType = VtChChartType3dBar
      .Plot.Light.AmbientIntensity = 0.01 ' 1 % Intensity.
      .Plot.Light.EdgeIntensity = 0.5      ' 50 % Intensity.
      .Plot.Light.EdgeVisible = True
   End With
End Sub

Private Sub Command1_Click()
   Dim LightSource As Object
   Set LightSource = _
   MSChart1.Plot.Light.LightSources(1)
   ' Set coordinates for Light Source 1 as well as its
   ' intensity.
   LightSource.X = 1
   LightSource.y = 0.5
   LightSource.Z = 1
   LightSource.Intensity = 1
End Sub

Private Sub Command2_Click()
   ' Add a new light source.
   MSChart1.Plot.Light.LightSources.Add 0.5, 0.1, 1, 1
End Sub
```

See Also

Properties, Methods, and Events

LightSources Collection Properties, Methods, and Events

Properties

Count Property (ActiveX Controls)

Item Property (MSChart)

Methods

Add Method (LightSources Collection)

Remove Method (LightSources Collection)

Events

None

See Also

Applies To: LightSources Collection

Location Object

Represents the current position of a textual chart element such as the title, legend, or footnote.

Syntax

Location

The object placeholder represents an object expression that evaluates to an object in the Applies To list.

Example

The following example sets the title location for a chart using the **Location** object. To try the example, draw an **MSChart** and a **ComboBox** control on a form. Paste the code into the **Form** object's code module, and run the project. Click the **ComboBox** to change the title location.

```
Option Explicit

Private Sub Combo1_Click()
   MSChart1.Title.Location.LocationType = Combo1.ListIndex
End Sub

Private Sub Form_Load()
   ' Set Title Text.
   With MSChart1
      .TitleText = "Test Title Location"
      .Title.Location.Visible = True
   End With

   ' Add LocationType constants to ComboBox.
   With Combo1
      .AddItem "VtChLocationTypeTopLeft"     ' 0
      .AddItem "VtChLocationTypeTop"         ' 1
      .AddItem "VtChLocationTypeTopRight"    ' 2
      .AddItem "VtChLocationTypeLeft"        ' 3
      .AddItem "VtChLocationTypeRight"       ' 4
      .AddItem "VtChLocationTypeBottomLeft"  ' 5
      .AddItem "VtChLocationTypeBottom"      ' 6
      .AddItem "VtChLocationTypeBottomRight" ' 7
      .AddItem "VtChLocationTypeCustom"      ' 8
      .ListIndex = 0
   End With
End Sub
```

See Also

Properties, Methods, and Events | Footnote Object | Legend Object | Title Object

Location Object Properties, Methods, and Events

Properties

LocationType Property

Rect Property

Visible Property (MSChart)

Methods

None

Events

None

See Also

Applies To: Location Object

Marker Object

A marker that identifies a data point on a chart.

Syntax

Marker

Example

The following example sets a blue X marker style for a chart series.

```
Option Explicit
Option Base 1
Private Sub Command1_Click()
   ' Display Markers for Series 1.
   Dim i As Integer
   For i = 1 To MSChart1.Plot.SeriesCollection.Count
      With MSChart1.Plot.SeriesCollection _
      .Item(i).DataPoints.Item(-1).Marker
         .Visible = True
         .Size = 20
         .Style = VtMarkerStyleX
         .FillColor.Automatic = False
         .FillColor.Set 0, 0, 255
      End With
   Next i
End Sub
```

See Also

Properties, Methods, and Events | DataPoint Object

Marker Object Properties, Methods, and Events

Properties

FillColor Property

Pen Property

Size Property (MSChart)

Style Property (MSChart Control)

Visible Property (MSChart)

Methods

None

Events

None

See Also

Applies To: Marker Object

Pen Object

Describes the color and pattern of lines or edges on a chart.

Syntax

Pen

Example

The following example sets the pen attributes for a two-dimensional xy chart series. The **GuideLinePen** property returns a reference to a **Pen** object.

```
Private Sub Command1_Click()
   ' Set Guide Lines for 2D XY chart Series 1.
   MSChart1.ChartType = VtChChartType2dXY
   MSChart1.Plot.SeriesCollection.Item(1) _
   .ShowGuideLine(VtChAxisIdX) = True
   With _
   MSChart1.Plot.SeriesCollection.Item(1).GuideLinePen
      ' Set Pen attributes.
      .VtColor.Set 255, 255, 0
      .Width = 10
      .Style = VtPenStyleDashDot
      .Join = VtPenJoinRound
      .Cap = VtPenCapRound
   End With
End Sub
```

See Also

Properties, Methods, and Events | Axis Object | Marker Object | PlotBase Object | Series Object | Wall Object

Pen Object Properties, Methods, and Events

Properties

Cap Property

Join Property

Limit Property

Style Property (MSChart Control)

VtColor Property

Width Property (MSChart)

Methods

None

Events

None

See Also

Applies To: Pen Object

Plot Object

Represents the area upon which a chart is displayed.

Syntax

Plot

Remarks

The **Plot** object allows you to program the following objects:

1. **Axis** object — represents the *x*, *y*, and *z* axis of the chart. The *z* axis is only visible on 3D charts.
2. **BackDrop** object — the area behind the axes.
3. **Light** object — the ambient and edge light of the plot.
4. **LocationRect** object — the location of the plot.
5. **SeriesCollection** object — the collection of series sets.
6. **View3D** object — the elevation and rotation of the 3D image.
7. **Wall** object — the area behind the plot.

Example

The following example sets the chart viewing distance and axis division spacing.

```
Private Sub Command1_Click()
   ' Change the chart type to 3D Bar.
   MSChart1.ChartType = VtChChartType3dBar
   With MSChart1.Plot
      ' Changes 3d bar chart's viewing.
      .DepthToHeightRatio = 2
      .WidthToHeightRatio = 2
      ' Changes the spacing between divisions on the
      ' X-Axis.
      .xGap = 0
      ' Changes the spacing between divisions on the
      ' Z-Axis.
      .zGap = 0.8
   End With
End Sub
```

See Also

Properties, Methods, and Events | Light Object

Plot Object Properties, Methods, and Events

Properties

AngleUnit Property

AutoLayout Property

Axis Property

Backdrop Property

BarGap Property

Clockwise Property

DataSeriesInRow Property

DefaultPercentBasis Property

DepthToHeightRatio Property

Light Property

LocationRect Property

PlotBase Property

Projection Property

SeriesCollection Property

Sort Property (MSChart)

StartingAngle Property

SubPlotLabelPosition Property

UniformAxis Property

View3D Property

Wall Property

Weighting Property

WidthToHeightRatio Property

XGap Property

ZGap Property

Methods

None

Events

None

See Also

Applies To: Plot Object

PlotBase Object

The area beneath a chart.

Syntax

PlotBase

Example

The following example sets the chart base parameters on a three-dimensional bar chart.

```
Private Sub Command1_Click()
   ' Change the chart type to 3D.
   MSChart1.ChartType = VtChChartType3dBar
   With MSChart1.Plot.PlotBase
      ' Change the base height.
      .BaseHeight = 20
      ' Use the pattern style for base.
      .Brush.Style = VtBrushStylePattern
      .Brush.Index = VtBrushPatternHorizontal
      .Brush.FillColor.Set 255, 160, 160
      .Brush.PatternColor.Set 180, 180, 255
      .Pen.Style = VtPenStyleSolid
      .Pen.VtColor.Set 72, 72, 255
   End With
End Sub
```

See Also

Properties, Methods, and Events | Backdrop Object | Brush Object | Plot Object

PlotBase Object Properties, Methods, and Events

Properties

BaseHeight Property

Brush Property

Pen Property

Methods

None

Events

None

See Also

Applies To: PlotBase Object

Rect Object

Defines a coordinate location.

Syntax

Rect

Example

The example increases the size of the chart plot using the **LocationRect** property and the x and y properties of the **Rect** object.

```
Private Sub Command1_Click()
   ' Increase the size of the chart plot.
   MSChart1.Plot.AutoLayout = False
   With MSChart1.Plot.LocationRect
      .Min.x = .Min.x * 1.2
      .Min.y = .Min.y * 1.2
      .Max.x = .Max.x * 1.2
      .Max.y = .Max.y * 1.2
   End With
End Sub
```

See Also

Properties, Methods, and Events | Coor Object

Rect Object Properties, Methods, and Events

Properties

Max Property (MSChart)

Min Property (MSChart)

Methods

None

Events

None

See Also

Applies To: Rect Object

Series Collection

A collection of chart series.

Syntax

Series (*index*)

The **Series** collection syntax has these parts:

Part	Description
index	Integer. Identifies the series of the chart. Series are identified in the order of data grid columns, beginning with 1.

Example

The following example hides all the series in a chart.

```
Private Sub Command1_Click()
   Dim serX As Series
   ' Hides All Series.
   For Each serX In MSChart1.Plot.SeriesCollection
      serX.Position.Hidden = True
   Next
End Sub
```

See Also

Properties, Methods, and Events | Series Object

Series Collection Properties, Methods, and Events

Properties

Item Property (MSChart)

Methods

Count Method

Events

None

See Also

Applies To: Series Collection

Series Object

An item from a **SeriesCollection** collection that represents a group of data points on a chart.

Syntax

Series

Example

The following example changes the appearance of the lines for all series in a three-dimensional line chart.

```
Private Sub Command1_Click()
   Dim serX As Series
   ' Change the chart type to 3D line and smoothing
   ' each line.
   MSChart1.chartType = VtChChartType3dLine
   MSChart1.ColumnCount = 4
   For Each serX In MSChart1.Plot.SeriesCollection
      serX.ShowGuideLine(VtChAxisIdY) = True
      serX.GuideLinePen.Style = VtPenStyleDitted
      serX.Pen.Style = 4
   Next
End Sub
```

See Also

Properties, Methods, and Events | SeriesCollection Collection

Series Object Properties, Methods, and Events

Properties

DataPoints Property

GuidelinePen Property

LegendText Property

Pen Property

Position Property (MSChart)

SecondaryAxis Property

SeriesMarker Property

SeriesType Property

ShowGuideLine Property

ShowLine Property

StatLine Property

TypeByChartType Method

Methods

Select Method

Events

None

See Also

Applies To: Series Object

SeriesCollection Collection

Provides information about the series that make up a chart.

Syntax

SeriesCollection (*index*)

The **SeriesCollection** collection syntax has these parts:

Part	Description
index	Identifies a specific series in the series collection.

Example

See Series Collection.

See Also

Properties, Methods, and Events

SeriesCollection Collection Properties, Methods, and Events

Properties

Item Property (MSChart)

Methods

Count Property (ActiveX Controls)

Events

None

See Also

Applies To: SeriesCollection Collection

SeriesMarker Object

Describes a marker that identifies all data points within one series on a chart.

Syntax

SeriesMarker

Example

The following example sets marker parameters for all series in a chart.

```
Private Sub Command1_Click()
   Dim serX As series
   ' Show markers and unshow the lines for all series.
   MSChart1.chartType = VtChChartType2dLine
   For Each serX In MSChart1.Plot.SeriesCollection
      serX.SeriesMarker.Show = True
      serX.ShowLine = False
   Next
End Sub
```

See Also

Properties, Methods, and Events

SeriesMarker Object Properties, Methods, and Events

Properties

Auto Property (SeriesMarker)

Show Property

Methods

None

Events

None

See Also

Applies To: SeriesMarker Object

SeriesPosition Object

The location where a chart series is drawn in relation to other series. If all series have the same order (position), then they are stacked.

Syntax

SeriesPosition

See Also

Properties, Methods, and Events

SeriesPosition Object Properties, Methods, and Events

Properties

Excluded Property

Hidden Property (MSChart)

Order Property

StackOrder Property

Methods

None

Events

None

See Also

Applies To: SeriesPosition Object

Shadow Object

Holds information about the appearance of a shadow on a chart element.

Syntax

Shadow

Example

The following example sets a shadow on a chart backdrop title.

```
Private Sub Command1_Click()
   ' Show shadow for title.
   With MSChart1.Title
      .Location.Visible = True
      .Text = "Chart Title"
      .Backdrop.Frame.Width = 1
      .Backdrop.Frame.FrameColor.Set 255, 0, 0
      .Backdrop.Frame.Style = VtFrameStyleSingleLine
      .Backdrop.Shadow.Style = VtShadowStyleDrop
      .Backdrop.Shadow.Offset.x = 10
      .Backdrop.Shadow.Offset.y = 10
   End With
End Sub
```

See Also

Properties, Methods, and Events | Brush Object | Coor Object

Shadow Object Properties, Methods, and Events

Properties

Brush Property

Offset Property

Style Property (MSChart Control)

Methods

None

Events

None

See Also

Applies To: Shadow Object

StatLine Object

Describes how statistic lines are displayed on a chart.

Syntax

StatLine

Example

The following example sets the color and pen parameters for a chart statistics line.

```
Private Sub Command1_Click()
   ' Show all statistic lines for series 2.
   MSChart1.chartType = VtChChartType2dLine
   With MSChart1.plot.SeriesCollection(2).StatLine
      .VtColor.Set 128, 128, 255
      .Flag = VtChStatsMinimum Or VtChStatsMaximum _
      Or VtChStatsMean Or VtChStatsStddev Or _
      VtChStatsRegression
      .Style(vtChStatsMinimum) = VtPenStyleDotted
      .width = 2
   End With
End Sub
```

See Also

Properties, Methods, and Events

StatLine Object Properties, Methods, and Events

Properties

Flag Property

Style Property (StatLine)

VtColor Property

Width Property (MSChart)

Methods

None

Events

None

See Also

Applies To: StatLine Object

TextLayout Object

Represents text positioning and orientation.

Syntax

TextLayout

Example

The following example sets the title text position and orientation for a chart.

```
Private Sub Command1_Click()
   ' Sets the title text position and orientation.
   With Form1.MSChart1.Title
      .Location.Visible = True
      .Location.LocationType = VtChLocationTypeLeft
      .Text = "Title TextLayout"
   End With
   With Form1.MSChart1.Title.TextLayout
      .Orientation = VtOrientationUp
      .HorzAlignment = VtHorizontalAlignmentCenter
      .VertAlignment = VtVerticalAlignmentCenter
   End With
End Sub
```

See Also

Properties, Methods, and Events

TextLayout Object Properties, Methods, and Events

Properties

HorzAlignment Property

Orientation Property (MSChart)

VertAlignment Property

WordWrap Property (MSChart)

Methods

None

Events

None

See Also

Applies To: TextLayout Object

Tick Object

A marker indicating a division along a chart axis.

Syntax

Tick

Example

The following example sets the tick length and style for the y axis on a chart.

```
Private Sub Command1_Click()
   ' Set the tick for y axis.
   With Form1.MSChart1.Plot.Axis(VtChAxisIdY, 1).Tick
      .Length = 500
      .Style = VtChAxisTickStyleOutside
   End With
End Sub
```

See Also

Properties, Methods, and Events

Tick Object Properties, Methods, and Events

Properties

Length Property (MSChart)

Style Property (MSChart Control)

Methods

None

Events

None

See Also

Applies To: Tick Object

Title Object

Text identifying the chart.

Syntax

Title

See Also

Properties, Methods, and Events | Backdrop Object

Title Object Properties, Methods, and Events

Properties

Backdrop Property

Font Property (MSChart)

Location Property

Text Property (MSChart)

TextLayout Property

TextLength Property

VtFont Property

Methods

Select Method

Events

None

See Also

Applies To: Title Object

ValueScale Object

Scale used to display a value axis.

Syntax

ValueScale

Example

The following example sets the major and minor grid line color for a two-dimensional bar chart using the **ValueScale** object.

```
Private Sub Command1_Click()
   ' Set chart type to 2d bar.
   MSChart1.ChartType = VtChChartType2dBar
   ' Use manual scale to display y axis (value axis).
   With MSChart1.Plot.Axis(VtChAxisIdY).ValueScale
      .Auto = False
      .MajorDivision = 2
      .MinorDivision = 5
   End With
   ' Show major grid line in red and minor grid line
   ' in blue.
   With MSChart1.Plot.Axis(VtChAxisIdY).AxisGrid
      .MajorPen.VtColor.Set 255, 0, 0
      .MajorPen.Width = 4
      .MinorPen.VtColor.Set 0, 0, 255
      .MinorPen.Width = 2
   End With
End Sub
```

See Also

Properties, Methods, and Events | CategoryScale Object | ValueScale Property

ValueScale Object Properties, Methods, and Events

Properties

Auto Property (ValueScale)

MajorDivision Property

Maximum Property

Minimum Property

MinorDivision Property

Methods

None

Events

None

See Also

Applies To: ValueScale Object

View3D Object

Represents the physical orientation of a three-dimensional chart.

Syntax

View3D

Example

The following example sets the chart elevation and rotation for a three-dimensional bar chart using the view object.

```
Private Sub Command1_Click()
   ' Set the chart type to 3d bar.
   Form1.MSChart1.ChartType = VtChChartType3dBar
   With Form1.MSChart1.Plot.View3d
      .Elevation = 90   ' Look directly down onto the
                        ' top of the chart.
      .Rotation = 90
   End With
End Sub
```

See Also

Properties, Methods, and Events

View3D Object Properties, Methods, and Events

Properties

Elevation Property

Rotation Property

Methods

Set Method (View3D)

Events

None

See Also

Applies To: View3D Object

VtColor Object

Describes a drawing color in a chart.

Syntax

VtColor

Example

The following example sets the fill color for a chart backdrop brush. The **FillColor** property returns a reference to the **VtColor** object.

```
Private Sub Command1_Click()
' Sets Backdrop to Fill - Brush Style.
   MSChart1.Backdrop.Fill.Style = VtFillStyleBrush
   ' Sets chart fill color to red.
   With MSChart1.Backdrop.Fill.Brush.FillColor
      .Red = 255   ' Use properties to set color.
      .Green = 0
      .Blue = 0
   End With
End Sub
```

See Also

Properties, Methods, and Events

VtColor Object Properties, Methods, and Events

Properties

Automatic Property

Blue Property

Green Property

Red Property

Methods

Set Method (VtColor)

Events

None

See Also

Applies To: VtColor Object

VtFont Object

The font used to display chart text.

Syntax

VtFont

Example

The following example sets the font parameters for a chart title.

```
Private Sub Command1_Click()
   ' Ask user to supply a title.
   MSChart1.Title.Text = InputBox("Title?")
   ' Make Chart Title visible.
   MSChart1.Title.Location.Visible = True
   ' Set font for Chart Title.
   With MSChart1.Title.VtFont
      .Name = "Times New Roman"
      .Size = 18
      .Style = VtFontStyleBold Or _
      VtFontStyleItalic
      ' Use both StrikeThrough and Underline in the
      ' text.
      .Effect = VtFontEffectStrikeThrough Or _
      VtFontEffectUnderline
      ' Set text color to Blue.
      .VtColor.Set 0, 0, 255
   End With
End Sub
```

See Also

Properties, Methods, and Events

VtFont Object Properties, Methods, and Events

Properties

Effect Property

Name Property (VtFont)

Size Property (MSChart)

Style Property (MSChart Control)

VtColor Property

Methods

None

Events

None

See Also

Applies To: VtFont Object

Wall Object

A planar area depicting the y axes on a three-dimensional chart.

Syntax

Wall

Example

The following example displays a colored wall for a three-dimensional chart.

```
Private Sub Command1_Click()
   ' Displays a colored wall for a 3D chart.
   Form1.MSChart1.ChartType = VtChChartType3dBar
   With Form1.MSChart1.Plot.Wall
      .Brush.Style = VtBrushStylePattern
      .Brush.Index = VtBrushPatternChecks
      .Brush.FillColor.Set 255, 120, 120
      .Brush.PatternColor.Set 120, 120, 0
      .Width = 20
   End With
End Sub
```

See Also

Properties, Methods, and Events | Brush Object

Wall Object Properties, Methods, and Events

Properties

Brush Property

Pen Property

Width Property (MSChart)

Methods

Set Method (Weighting)

Events

None

See Also

Applies To: Wall Object

Weighting Object

Represents the size of a pie in relation to other pies in the same chart.

Syntax

Weighting

Example

The following example shows the weighting of a pie chart.

```
Private Sub Command1_Click()
   ' Show the weighting of the pie.
   Form1.MSChart1.ChartType = VtChChartType2dPie
   With Form1.MSChart1.Plot.Weighting
      .Basis = VtChPieWeightBasisTotal
      .Style = VtChPieWeightStyleArea
   End With
End Sub
```

See Also

Properties, Methods, and Events

Weighting Object Properties, Methods, and Events

Properties

Basis Property

Style Property (MSChart Control)

Methods

Set Method (Weighting)

Events

None

See Also

Applies To: Weighting Object

MSChart Control Properties Listed Alphabetically

ActiveSeriesCount Property

AllowDithering Property

AllowDynamicRotation Property

AllowSelections Property

AllowSeriesSelection Property

AmbientIntensity Property

AngleUnit Property

Auto Property (CategoryScale)

Auto Property (Intersection)

Auto Property (Label)

Auto Property (SeriesMarker)

Auto Property (ValueScale)

AutoIncrement Property

AutoLayout Property

Automatic Property

Axis Property

AxisGrid Property

AxisId Property

AxisScale Property

AxisTitle Property

Backdrop Property

BarGap Property

BaseHeight Property

Basis Property

Blue Property

Brush Property

Cap Property

CategoryScale Property

Chart3d Property

ChartData Property

ChartType Property

Clockwise Property

Column Property (MSChart Control)

ColumnCount Property

ColumnLabel Property (DataGrid Object)

ColumnLabel Property (MSChart)

ColumnLabelCount Property

ColumnLabelIndex Property

Component Property
CompositeColumnLabel Property
CompositeRowLabel Property
Custom Property
Data Property (MSChart)
DataGrid Property
DataPointLabel Property
DataPoints Property
DataSeriesInRow Property
DefaultPercentBasis Property
DepthToHeightRatio Property
DivisionsPerLabel Property
DivisionsPerTick Property
DoSetCursor Property
DrawMode Property (MSChart)
EdgeIntensity Property
EdgePen Property
EdgeVisible Property
Effect Property
Elevation Property
Excluded Property
Fill Property
FillColor Property
Flag Property
Font Property (MSChart)
Footnote Property
FootnoteText Property
Format Property (Label Object)
FormatLength Property
Frame Property
FrameColor Property
Green Property
GuidelinePen Property
Hidden Property (MSChart)
Hide Property
HorzAlignment Property
Index Property (Brush)
Index Property (Intersection)
Intensity Property
Intersection Property

Item Property (MSChart)
Join Property
LabelLevelCount Property
Labels Property
LabelsInsidePlot Property
LabelTick Property
Legend Property
LegendText Property
Length Property (MSChart)
Light Property
LightSources Property
Limit Property
LineStyle Property (DataPointLabel Object)
Location Property
LocationRect Property
LocationType Property
LogBase Property
MajorDivision Property
MajorPen Property
Marker Property
Max Property (MSChart)
Maximum Property
Min Property (MSChart)
Minimum Property
MinorDivision Property
MinorPen Property
MousePointer Property (MSChart)
Name Property (VtFont)
Offset Property
Order Property
Orientation Property (MSChart)
PatternColor Property
Pen Property
PercentBasis Property
PercentFormat Property
Plot Property
PlotBase Property
Point Property
Position Property (MSChart)
Projection Property

RandomFill Property

Rect Property

Red Property

Repaint Property

Rotation Property

Row Property (MSChart)

RowCount Property (MSChart)

RowLabel Property (DataGrid Object)

RowLabel Property (MSChart)

RowLabelCount Property

RowLabelIndex Property

SecondaryAxis Property

SeriesCollection Property

SeriesColumn Property

SeriesMarker Property

SeriesType Property

Shadow Property

Show Property

ShowGuideLine Property

ShowLegend Property

ShowLine Property

Size Property (MSChart)

Sort Property (MSChart)

SpaceColor Property

Stacking Property

StackOrder Property

Standing Property

StartingAngle Property

StatLine Property

Style Property (MSChart Control)

Style Property (StatLine)

SubPlotLabelPosition Property

Text Property (MSChart)

TextLayout Property

TextLength Property

TextLengthType Property

Tick Property

Title Property (MSChart)

TitleText Property

Type Property (MSChart)

UniformAxis Property

ValueFormat Property

ValueScale Property

VertAlignment Property

View3D Property

Visible Property (MSChart)

VtColor Property

VtFont Property

Wall Property

Weighting Property

Width Property (MSChart)

WidthToHeightRatio Property

WordWrap Property (MSChart)

X Property

XGap Property

Y Property

Z Property

ZGap Property

ActiveSeriesCount Property

Returns the number of series that appear on a chart based on the number of columns in the **DataGrid** object and the type of chart being drawn.

Syntax

*object.***ActiveSeriesCount**

The object placeholder represents an object expression that evaluates to an object in the Applies To list.

See Also

DataGrid Object

Applies To: MSChart Control

AllowDithering Property

Returns or sets a value that determines whether to disable color dithering for charts on 8-bit color monitors in order to enable use of **MSChart** control's own color palette and enhance the chart display.

Syntax

object.**AllowDithering** [=*boolean*]

The **AllowDithering** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
boolean	A Boolean expression that specifies whether a color dithering is allowed, as described in Settings .

Settings

The settings for *boolean* are:

Setting	Description
True	Color dithering is allowed.
False	(Default) MSChart control's color palette is used for enhanced color matching and display .

See Also

Applies To: MSChart Control

AllowDynamicRotation Property

Returns or sets a value that indicates whether users can interactively rotate three-dimensional charts by holding down the control key to display the rotation cursor.

Syntax

object.**AllowDynamicRotation** [= *boolean*]

The **AllowDynamicRotation** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
boolean	A Boolean expression that specifies whether a dynamic rotation is allowed, as described in Settings .

Settings

The settings for *boolean* are:

Setting	Description
True	(Default) The user can interactively rotate the chart with the cursor .
False	The user cannot interactively rotate the chart with the cursor.

See Also

Applies To: MSChart Control

AllowSelections Property

Returns or sets a value that indicates whether or not users can select chart objects.

Syntax

object.**AllowSelections** [= *boolean*]

The **AllowSelections** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
boolean	A Boolean expression that specifies whether selections can be made, as described in Settings .

Settings

The settings for *boolean* are:

Setting	Description
True	(Default) The user can interactively select chart objects .
False	The user cannot select chart objects.

See Also

Applies To: MSChart Control

AllowSeriesSelection Property

Returns or sets a value that indicates whether a series is selected when a user clicks on an individual chart data point.

Syntax

object.**AllowSeriesSelection** [= *boolean*]

The **AllowSeriesSelection** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
boolean	A Boolean expression that specifies whether series are selected, as described in Settings .

Settings

The settings for *boolean* are:

Setting	Description
True	(Default) Users can select a series by clicking a data point.
False	Clicking a data point selects only that data point, not the entire series .

See Also

Applies To: MSChart Control

AmbientIntensity Property

Returns or sets the percentage of ambient light illuminating a three-dimensional chart.

Syntax

object.**AmbientIntensity** [= *intensity*]

The **AmbientIntensity** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
intensity	Single. The chart light intensity. Valid values are 0 to 1. If set to 1, all sides of the chart elements are fully illuminated no matter what light sources are turned on. If set at 0, there is no contribution fro m ambient light; only the sides of the chart elements facing active light sources are illuminated.

See Also

Applies To: Light Object

AngleUnit Property

Returns or sets the unit of measure used for all chart angles.

Syntax

object.**AngleUnit** [= *unit*]

The **AngleUnit** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
unit	Integer. A VtAngleUnits constant describing the unit of measure. The angles can be measured in degrees, radians, or grads.

See Also

AngleUnits Constants

Applies To: Plot Object

AngleUnits Constants

VtAngleUnits provides the valid units for measuring chart angles.

Constants	Description
VtAngleUnitsDegrees	Chart angles are measured in degrees.
VtAngleUnitsRadians	Chart angles are measured in radians.
VtAngleUnitsGrads	Chart angles are measured in grads.

See Also

AngleUnit Property

Auto Property (CategoryScale)

Returns or sets a value that indicates whether the axis is automatically scaled.

Syntax

object.**Auto** [= *boolean*]

The **Auto** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
boolean	A Boolean expression that specifies whether the item is displayed, as described in Settings .

Settings

The settings for *boolean* are:

Setting	Description
True	(Default) The axis is automatically scaled based on the data being charted.
False	The axis is not automatically scaled. Values in **DivisionsPerLabel** and **DivisionsPerTick** are used to determine the scale.

See Also

Auto Property (Intersection) | Auto Property (Label) | Auto Property (SeriesMarker) | Auto Property (ValueScale) | DivisionsPerLabel Property | DivisionsPerTick Property | LabelTick Property

Applies To: CategoryScale Object

Auto Property (Intersection)

Returns or sets a value that determines whether or not the **Intersection** object uses the value of the **Point** property to position the axis.

Syntax

object.**Auto** [= *boolean*]

The **Auto** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
boolean	A Boolean expression that specifies whether the item is displayed, as described in Settings.

Settings

The settings for *boolean* are:

Setting	Description
True	(Default) The axis is positioned in its standard location.
False	The intersecting axis is positioned at the value indicated by **Point**.

See Also

Auto Property (CategoryScale) | Auto Property (Label) | Auto Property (SeriesMarker) | Auto Property (ValueScale)

Applies To: Intersection Object

Auto Property (Label)

Returns or sets a value that determines whether axis labels are automatically rotated to improve the chart layout.

Syntax

object.**Auto** [= *boolean*]

The **Auto** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
boolean	A Boolean expression that specifies how axis labels will be positioned, as described in Settings .

Settings

The settings for *boolean* are:

Setting	Description
True	(Default) The labels may be rotated.
False	The labels are not rotated. Long labels may not display properly .

See Also

Auto Property (CategoryScale) | Auto Property (Intersection) | Auto Property (SeriesMarker) | Auto Property (ValueScale)

Applies To: Label Object

Auto Property (SeriesMarker)

Returns or sets a value that determines if the **SeriesMarker** object assigns the next available marker to all data points in the series.

Syntax

object.**Auto** [= *boolean*]

The **Auto** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
boolean	A Boolean expression that controls how markers are assigned, as described in Settings .

Settings

The settings for *boolean* are:

Setting	Description
True	(Default) The **SeriesMarker** object assigns the marker .
False	You can assign a custom marker.

Remarks

Set this property to **False** if you wish to change the series marker type.

This property is automatically set to **False** if the **Marker** property of the **DataPoint** object is set.

See Also

Auto Property (CategoryScale) | Auto Property (Intersection) | Auto Property (Label) | Auto Property (ValueScale)

Applies To: SeriesMarker Object

Auto Property (ValueScale)

Returns or sets a value that determines whether automatic scaling is used to draw the value axis.

Syntax

object.**Auto** [= *boolean*]

The **Auto** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
boolean	A Boolean expression that determines whether automatic scaling is used, as described in Settings .

Settings

The settings for *boolean* are:

Setting	Description
True	The scale is automatically set based on the data being charted.
False	The values in the **Minimum**, **Maximum**, **MajorDivisions** and **MinorDivisions** properties are used to scale the axis.

See Also

Auto Property (CategoryScale) | Auto Property (Intersection) | Auto Property (Label) | Auto Property (SeriesMarker)

Applies To: ValueScale Object

AutoIncrement Property

Returns or sets a value that determines if the properties that set the current data point are incremented during data entry without manually setting the **Column** and **Row** properties.

Syntax

object.**AutoIncrement** [= *boolean*]

The **AutoIncrement** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
boolean	A Boolean expression that specifies whether the current data point is incremented, as described in Settings.

Settings

The settings for *boolean* are:

Setting	Description
True	When the **Data** property is changed, the **Row** property updates to the next row in the column. If you are at the end of a column, the **Column** property increments to the next column.
False	(Default) The current data point is not incremented.

See Also

Applies To: MSChart Control

AutoLayout Property

Returns or sets a value that determines whether or not a **Plot** object is in manual or automatic layout mode.

Syntax

object.**AutoLayout** [= *boolean*]

The **AutoLayout** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
boolean	A Boolean expression that specifies the layout mode, as described in Settings .

Settings

The settings for *boolean* are:

Setting	Description
True	(Default) The **Plot** object automatically determines the proper size and position of the plot based on the size and position of other elements.
False	The coordinates specified by **Plot** object's **LocationRect** property are used to position the plot.

See Also

Applies To: Plot Object

Automatic Property

Returns or sets a value that determines whether the color is calculated automatically. This is only used for edge pens on chart elements.

Syntax

object.**Automatic** [= *boolean*]

The **Automatic** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
boolean	A Boolean expression that determines whether the color is calculated automatically, as described in Setting s.

Settings

The settings for *boolean* are:

Setting	Description
True	Color automatically picks up the brush color used on the chart series .
False	The color is determined based on the settings of **Value**.

See Also

Applies To: VtColor Object

Axis Property

Returns a reference to an **Axis** object that describes an axis on a chart.

Syntax

object.**Axis**(*axisID*)

The **Axis** property syntax has these parts:

Part	Description
object	Required. An object expression that evaluates to an object in the Applies To list.
axisID	Required. An integer or VtChAxisId constant that identifies a specific axis.
index	Optional. Reserved for future use. Identifies the specific axis when there is more than one axis with the same *axisID*.

Remarks

Three axes are available: *x*, *y*, and *z*. The *z* axis is visible only when the chart is a 3D chart.

See Also

AxisId Constants

Applies To: Plot Object

AxisGrid Property

Returns a reference to an **AxisGrid** object that represents the planar area surrounding a chart axis.

Syntax

object.**AxisGrid**

The object placeholder represents an object expression that evaluates to an object in the Applies To list.

See Also

Applies To: Axis Object

AxisId Property

Returns a specific axis that intersects with the current axis.

Syntax

object.**AxisId**

The object placeholder represents an object expression that evaluates to an object in the Applies To list.

Return Value

The return value is an integer that identifies the intersecting axis.

See Also

AxisId Constants

Applies To: Intersection Object

AxisId Constants

The **VtChAxisId** constants provide options for identifying a chart axis.

Constants	Description
VtChAxisIdX	Identifies the *x* axis.
VtChAxisIdY	Identifies the *y* axis.
VtChAxisIdY2	Identifies the secondary *y* axis.
VtChAxisIdZ	Identifies the *z* axis.

See Also

Axis Property | AxisId Property | GetSelectedPart Method | SelectPart Method | ShowGuideLine Property | TwipsToChartPart Method

AxisScale Property

Returns a reference to an **AxisScale** object that describes how chart values are plotted on an axis.

Syntax

object.**AxisScale**

The object placeholder represents an object expression that evaluates to an object in the Applies To list.

See Also

AxisScale Object

Applies To: Axis Object

AxisTitle Property

Returns a reference to an **AxisTitle** object associated with the axis of a chart.

Syntax

*object.***AxisTitle**

The object placeholder represents an object expression that evaluates to an object in the Applies To list.

See Also

Applies To: Axis Object

Backdrop Property

Returns a reference to a **Backdrop** object that describes the shadow, pattern, or picture behind a chart or chart element.

Syntax

object.**Backdrop**

The object placeholder represents an object expression that evaluates to an object in the Applies To list.

See Also

Applies To: AxisTitle Object | DataPointLabel Object | Footnote Object | Label Object | Legend Object | MSChart Control | Plot Object | Title Object

BarGap Property

Returns or sets the spacing of two-dimensional bars or clustered three-dimensional bars within a category.

Syntax

object.**BarGap** [= *value*]

The **BarGap** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
value	Single. The bar spacing value. This is measured as a percentage of the bar width. A value of 0 results in the bars touching. A value of 100 means the gap between the bars is as wide as the bars.

See Also

Applies To: Plot Object

BaseHeight Property

Returns or sets the height of the three-dimensional chart base in points.

Syntax

object.**BaseHeight** [= *height*]

The **BaseHeight** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list .
height	Single. The base height.

See Also

Applies To: PlotBase Object

Basis Property

Returns or sets the type of weighting used to determine pie size on a chart.

Syntax

object.**Basis** [= *type*]

The **Basis** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list .
type	A **VtChPieWeightBasis** constant that identifies the weighting type.

See Also

PieWeightBasis Constants

Applies To: Weighting Object

PieWeightBasis Constants

The **VtChPieWeightBasis** constants provide options for displaying pie chart slices.

Constant	Description
VtChPieWeightBasisNone	All pies are drawn the same size.
VtChPieWeightBasisTotal	The slice values in each pie are totaled and the pie with the highest total identified. The size of each pie in the chart is determined by the ratio of its total value compared to the largest pie.
VtChPieWeightBasisSeries	The first column of data in the data grid holds the relative size index. In other words, if you have 5 categories, you can control the size of the pies representing each category by using the first column of the data grid to number the rows 1 through 5. The size of the pie is determined by the ratio of its first column value and the largest value in the first column. The pie containing the 1 is the largest pie; the one containing the 5 the smallest. It is most common to exclude this first column of data so that the values are not drawn as a pie slice.

See Also

Basis Property | Set Method (Weighting)

Blue Property

Returns or sets the blue component of the RGB value in a chart.

Syntax

object.**Blue [=***b***]**

The **Blue** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list .
b	Integer. The blue value.

Remarks

RGB specifies the relative intensity of red, green, and blue to cause a specific color to be displayed. The valid range for a normal RGB color is 0 to 16,777,215. The value for any argument to RGB that exceeds 255 is assumed to be 255.

See Also

Applies To: VtColor Object

Brush Property

Returns a reference to a **Brush** object that describes the fill type used to display a chart element.

Syntax

object.**Brush**

The *object* placeholder represents an object expression that evaluates to an object in the Applies To list.

Example

See Brush Object.

See Also

Brush Object

Applies To: DataPoint Object | Fill Object | PlotBase Object | Shadow Object | Wall Object

Cap Property

Returns or sets a value that determines how line ends are capped.

Syntax

object.**Cap** [= *type*]

The **Cap** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list .
type	Integer. A VtPenCap constant that describes the line pen cap style.

See Also

PenCap Constants

Applies To: Pen Object

PenCap Constants

VtPenCap provides methods for displaying line endings.

Constant	Description
VtPenCapButt	The line is squared off at the endpoint.
VtPenCapRound	A semicircle with the diameter of the line thickness is drawn at the end of the line.
VtPenCapSquare	The line continues beyond the endpoint for a distance equal to half the line thickness a nd is squared off.

See Also

Cap Property

CategoryScale Property

Returns a reference to a **CategoryScale** object that describes the scale information for a category axis.

Syntax

object.**CategoryScale**

The object placeholder represents an object expression that evaluates to an object in the Applies To list.

See Also

ValueScale Property

Applies To: Axis Object

Chart3d Property

Returns a value that specifies whether or not a chart is three dimensional.

Syntax

object.**Chart3D**

The object placeholder represents an object expression that evaluates to an object in the Applies To list.

Return Values

Setting	Description
True	The chart is a three-dimensional chart.
False	The chart is not a three-dimensional chart.

Remarks

Whether or not a chart is three-dimensional depends on the **ChartType** property's setting.

See Also

Applies To: MSChart Control

ChartData Property

Returns or sets an array that contains the values that will be displayed by the chart.

Syntax

object.**ChartData** [= *data*]

The **ChartData** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
data	Variant. A two-dimensional array that holds the data used to draw the chart .

Remarks

If the first series of a multi-dimensional array contains strings, those strings will become the labels of the chart.

ChartData is the default property for the MSChart control.

Example

The following example uses a Visual Basic array to load the chart data grid directly. To try the example, draw a **MSChart** control on a form, paste the code into the **Form** object's code module, and run the project.

```
Private Sub Form_Load()
    Dim arrData(3, 1 To 3)
    arrData(1, 1) = "Jan"   ' Set the labels in the first series.
    arrData(2, 1) = "Feb"
    arrData(3, 1) = "Mar"

    arrData(1, 2) = 8
    arrData(2, 2) = 4
    arrData(3, 2) = 0.3

    arrData(1, 3) = 0.2
    arrData(2, 3) = 3
    arrData(3, 3) = 6.3
    MSChart1.ChartData = arrData
End Sub
```

In this example, the lower subscript bound was declared as 1 using the **Option Base 1** statement, rather than the default of 0. We use a **Variant** array where the first series values are set to string variables and the second and third series' values are set to numeric values. This allows both the chart's labels and data to be set simultaneously. Note that declaring the array as type **String** works too, as long as the second and third series contain numeric values. If you wish only to set the chart's data, the array can be of the numeric types **Integer**, **Long**, **Single** or **Double**. Note that doing this will replace the existing chart labels with default row/column labels. Also, a one-dimensional array will work as well as a two-dimensional one as long as the last values are either numeric or text representations of numeric values.

The following example returns data from the chart. The example contains a loop to print out the array returned from the chart. Note the use of the **Lbound** and **Ubound** functions to determine the array bounds from the chart. To try the example, draw a **CommandButton** control onto a form with a **MSChart** control. Paste the code into the code module, run the project, and click the button.

```
Private Sub Command1_Click()
   Dim y() As Variant
   Dim i, j As Integer
   y = MSChart1.ChartData
   For i = LBound(y, 2) To UBound(y, 2)
      Debug.Print
      For j = LBound(y, 1) To UBound(y, 1)
         Debug.Print y(j, i)
```

```
        Next j
    Next i
End Sub
```

The returned array lower bound values are equal to 0. The returned array will always be a two-dimensional array of type **Variant**. Since **ChartData** is the default property for the chart, the object name alone, such as MSChart1, can be substituted for MSChart1.ChartData. So you could use MSChart1 = data or data = MSChart1.

See Also

Applies To: MSChart Control

ChartType Property

Returns or sets the chart type being used to display a chart.

Syntax

object.**ChartType** [= *type*]

The **ChartType** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list .
type	Integer. A VtChChartType constant that describes the chart type.

See Also

ChartType Constants

Applies To: MSChart Control

ChartType Constants

VtChChartType provides chart type options.

Constant	Description
VtChChartType3dBar	3D Bar
VtChChartType2dBar	2D Bar
VtChChartType3dLine	3D Line
VtChChartType2dLine	2D Line
VtChChartType3dArea	3D Area
VtChChartType2dArea	2D Area
VtChChartType3dStep	3D Step
VtChChartType2dStep	2D Step
VtChChartType3dCombination	3D Combination
VtChChartType2dCombination	2D Combination
VtChChartType2dPie	2D Pie
VtChChartType2dXY	2D XY

See Also

ChartType Property | TypeByChartType Method

Clockwise Property

Returns or sets a value that specifies whether pie charts are drawn in a clockwise direction.

Syntax

object.**Clockwise** [= *boolean*]

The **Clockwise** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
boolean	A Boolean expression that controls the direction used to draw pie charts, as described in Settings .

Settings

The settings for *boolean* are:

Setting	Description
True	(Default) Pie charts are drawn in a clockwise direction .
False	The charts are drawn in a counterclockwise direction.

See Also

Applies To: Plot Object

Column Property (MSChart Control)

Returns or sets the current data column in the data grid.

Syntax

object.**Column** [= *col*]

The **Column** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list .
col	Integer. The current data column.

Remarks

You must select a column before you can use other properties to change the column's corresponding chart series or any data point within the series.

See Also

Applies To: MSChart Control

ColumnCount Property

Returns or sets the number of columns in the current data grid associated with a chart.

Syntax

object.**ColumnCount** [= *count*]

The **ColumnCount** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list .
count	The number of data columns.

See Also

SetSize Method

Applies To: DataGrid Object | MSChart Control

ColumnLabel Property (DataGrid Object)

Returns or sets the label on a data column in the grid associated with a chart.

Syntax

object.**ColumnLabel**(*column*, *labelIndex*) [= *text*]

The **ColumnLabel** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
column	Integer. Identifies a specific data column. Columns are numbered from left to right beginning with 1. Any columns containing labels are not counted as data columns.
labelIndex	Integer. Identifies a specific label. If more than one level of column labels exist for the column, you must identify one of them. Column labels are numbered from bottom to top beginning at 1.
text	String. The column label text.

Example

See ColumnLabel Property (MSChart).

See Also

ColumnLabel Property (MSChart) | RowLabel Property (DataGrid Object)

Applies To: DataGrid Object

ColumnLabel Property (MSChart)

Returns or sets the label text associated with a column in the data grid of a chart.

Syntax

object.**ColumnLabel** [= *text*]

The **ColumnLabel** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list .
text	String. Label text associated with a column in the data grid.

Remarks

This property sets the label for the column currently identified by the **Column** property.

Example

The example adds three column labels to the z axis of a chart. To try the example, draw an **MSChart** control on a form. Paste the code into the **Form** object's code module, and run the project. Click the form to see the labels.

```
Private Sub Form_Click()
   With MSChart1
      .chartType = VtChChartType3dLine

      .Plot.Axis(VtChAxisIdZ).Labels(1).VtFont.Size = 24
      .DataGrid.ColumnLabel(1, 1) = "Label Z1"
      .DataGrid.ColumnLabel(2, 1) = "Label Z2"
      .DataGrid.ColumnLabel(3, 1) = "Label Z3"
   End With
End Sub

Private Sub Form_Load()
   ' The first series contains labels for the X axis.
   Dim arrData(4, 1 To 4)
   arrData(1, 1) = "Jan"
   arrData(2, 1) = "Feb"
   arrData(3, 1) = "Mar"
   arrData(4, 1) = "Apr"

   arrData(1, 2) = 8
   arrData(2, 2) = 4
   arrData(3, 2) = 0.3
   arrData(4, 2) = 3

   arrData(1, 3) = 0.2
   arrData(2, 3) = 3
   arrData(3, 3) = 6.3
   arrData(4, 3) = 2

   arrData(1, 4) = 5
   arrData(2, 4) = 7
   arrData(3, 4) = 2
   arrData(4, 4) = 9
   MSChart1.ChartData = arrData
End Sub
```

See Also

ColumnLabel Property (DataGrid Object) | RowLabel Property (MSChart)

Applies To: MSChart Control

ColumnLabelCount Property

Returns or sets the number of levels of labels on the columns in the data grid associated with a chart.

Syntax

object.**ColumnLabelCount** [= *count*]

The **ColumnLabelCount** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
count	Integer. The number of column label levels. Set this property to add or delete levels of labels on data grid columns.

Remarks

Column label levels are numbered from bottom to top, beginning at 1. Levels are added or subtracted from the top.

See Also

SetSize Method

Applies To: DataGrid Object | MSChart Control

ColumnLabelIndex Property

Returns or sets a specific level of column labels associated with a chart.

Syntax

object.**ColumnLabelIndex** [= *index*]

The **ColumnLabelIndex** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list .
index	Integer. Identifies a column label level.

Remarks

To set a label on a column with more than one level of labels, or to return the current value for a label, you must first identify which level you want to affect. Column label levels are numbered from bottom to top, beginning at 1.

See Also

Applies To: MSChart Control

Component Property

Returns or sets the type of label to be used to identify the data point.

Syntax

object.**Component** [= *type*]

The **Component** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
type	Integer. A VtChLabelComponent constant that identifies the label type .

See Also

LabelComponent Constants

Applies To: DataPointLabel Object

LabelComponent Constants

The **VtChLabelComponent** constants provide options for displaying chart labels.

Constant	Description
VtChLabelComponentValue	The value of the data point appears in the label. Data points in XY, Pola r, and Bubble charts actually have two or three values. The default label for these chart types display all values in a standard format. You can cu stomize this format to highlight an individual data value.
VtChLabelComponentPercent	The value of the data point is displayed in the label as a percentage of t he total value of the series.
VtChLabelComponentSeriesName	The series name is used to label the data point. This name is taken fro m the label associated with the column in the data grid.
VtChLabelComponentPointName	The data point name is used to label the data point.

See Also

Component Property

CompositeColumnLabel Property

Returns the multilevel label string that identifies a column in the data grid associated with a chart.

Syntax

object.**CompositeColumnLabel**(*column*)

The **CompositeColumnLabel** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
column	Integer. Identifies a specific data column. Columns are numbered from left to right beginning with 1. Any columns containing labels are not counted as data columns.

See Also

CompositeRowLabel Property

Applies To: DataGrid Object

CompositeRowLabel Property

Returns the multilevel label string that identifies a row in the data grid associated with a chart.

Syntax

object.**CompositeRowLabel** (*row*)

The **CompositeRowLabel** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
row	Integer. Identifies a specific data row. Rows are numbered from top to bottom beginning with 1. Any rows containing labels are not counted as data rows.

See Also

CompositeColumnLabel Property

Applies To: DataGrid Object

Custom Property

Returns or sets a value that determines if custom text is used to label a data point on a chart.

Syntax

object.**Custom** [= *boolean*]

The **Custom** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
boolean	A Boolean expression that specifies whether custom text is used, as described in Settings .

Settings

The settings for *boolean* are:

Setting	Description
True	The label contains custom text.
False	(Default) Information specified by the **DataPointLabel** object's **Components** property is used to label t he data point.

See Also

Applies To: DataPointLabel Object

Data Property (MSChart)

Returns or sets a value that is inserted into the current data point in the data grid of a chart.

Syntax

object.**Data** [= *value*]

The **Data** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list .
value	Integer. The data point value.

Remarks

If the current data point already contains a value, it is replaced by the new value. The chart is redrawn to reflect the new value for the current data point.

See Also

Applies To: MSChart Control

DataGrid Property

Returns a reference to a **DataGrid** object that describes the data grid associated with a chart.

Syntax

object.**DataGrid**

The object placeholder represents an object expression that evaluates to an object in the Applies To list.

See Also

Applies To: MSChart Control

DataPointLabel Property

Returns a reference to a **DataPointLabel** object that describes a label on an individual chart data point.

Syntax

object.**DataPointLabel**

The object placeholder represents an object expression that evaluates to an object in the Applies To list.

See Also

Applies To: DataPoint Object

DataPoints Property

Returns a reference to a **DataPoint** Collection that describes the data points within a chart series.

Syntax

object.**DataPoints**

The object placeholder represents an object expression that evaluates to an object in the Applies To list.

See Also

Applies To: Series Object

DataSeriesInRow Property

Returns or sets a value that indicates whether series data is being read from a row or a column in a data grid associated with a chart.

Syntax

object.**DataSeriesInRow** [= *boolean*]

The **DataSeriesInRow** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
boolean	A Boolean expression that controls how series data is read, as described in Settings .

Settings

The settings for *boolean* are:

Setting	Description
True	Series data is being read from a row in a data grid .
False	(Default) Series data is being read from a column.

See Also

Applies To: Plot Object

DefaultPercentBasis Property

Returns the default axis percentage basis for the chart.

Syntax

object.**DefaultPercentBasis**

The object placeholder represents an object expression that evaluates to an object in the Applies To list.

Return Value

An integer that specifies the default axis percentage basis. Possible return values are VtChPercentAxisBasis constants.

See Also

Applies To: Plot Object

DepthToHeightRatio Property

Returns or sets the percentage of the chart height to be used as the chart depth.

Syntax

object.**DepthToHeightRatio** [= *pctg*]

The **DepthToHeightRatio** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list .
pctg	Single. The chart height percentage.

Remarks

This property has an effect only on 3 dimensional chart types.

See Also

Applies To: Plot Object

DivisionsPerLabel Property

Returns or sets the number of divisions to skip between labels.

Syntax

object.**DivisionsPerLabel** [= *num*]

The **DivisionsPerLabel** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list .
num	Integer. An integer representing the number of divisions.

Remarks

If this property is set, the object's **Auto** property is automatically set to **False**.

See Also

Auto Property (CategoryScale) | DivisionsPerTick Property

Applies To: CategoryScale Object

DivisionsPerTick Property

Returns or sets the number of divisions to skip between major tick marks.

Syntax

object.**DivisionsPerTick** [= *num*]

The **DivisionsPerTick** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list .
num	Integer. An integer representing the number of divisions.

Remarks

If this property is set, the object's **Auto** property is automatically set to **False**.

See Also

Auto Property (CategoryScale) | DivisionsPerLabel Property

Applies To: CategoryScale Object

DoSetCursor Property

Returns or sets a value that indicates whether or not the cursor can be set by a chart. The **DoSetCursor** property determines whether or not the application can control what the mouse pointer looks like.

Syntax

object.**DoSetCursor** [= *boolean*]

The **DoSetCursor** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
boolean	A Boolean expression that specifies whether custom text is used, as described in Settings.

Settings

The settings for *boolean* are:

Setting	Description
True	(Default) The application can control the mouse pointer appearance .
False	The application cannot control the mouse pointer appearance.

See Also

Applies To: MSChart Control

DrawMode Property (MSChart)

Returns or sets a value that determines when and how a chart is repainted.

Syntax

object.**DrawMode** [= *mode*]

The **DrawMode** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
mode	Integer. A value that determines how and when the chart will be redrawn, as shown in Settings .

Settings

The settings for *mode* are:

Constant	Value	Description
VtChDrawModeDraw	0	Draws directly to the display device.
VtChDrawModeBlit	1	Blits an offscreen drawing to the display device.

See Also

DrawMode Constants

Applies To: MSChart Control

DrawMode Constants

The **VtChDrawMode** constants provide options for redisplaying a chart after it has been altered.

Constants	Description
VtChDrawModeDraw	Draw mode. The chart is redrawn on the screen every time you change a setting.
VtChDrawModeBlit	Blit mode. The chart is redrawn off the screen and displayed after the redraw is complete.

Blit mode stores a bitmap copy of the chart in memory when the chart is laid out. Repainting the chart uses the bitmap and draws very quickly. It requires more memory than Draw mode, but can save time waiting for the chart to redraw on screen. Blit mode is particularly useful when working with charts that contain many elements.

EdgeIntensity Property

Returns or sets the intensity of light used to draw the edges of objects in a three-dimensional chart.

Syntax

object.**EdgeIntensity** [= *edgeint*]

The **EdgeIntensity** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
edgeint	Single. The edge light intensity. Valid values are 0 to 1.0. An intensity of 0 turns edges off, drawing the edges as black lines; and an intensity of 1 fully illuminates the edges using the element's pen color.

Remarks

If this property is set, then the **Light** object's **EdgeVisible** property is automatically set to **True**.

See Also

Applies To: Light Object

EdgePen Property

Returns the **Pen** object used to draw the edge of the data point on a chart.

Syntax

object.**EdgePen**

The object placeholder represents an object expression that evaluates to an object in the Applies To list.

See Also

Applies To: DataPoint Object

EdgeVisible Property

Returns or sets a value that determines whether edges are displayed on the elements in a three-dimensional chart.

Syntax

object.**EdgeVisible** [= *boolean*]

The **EdgeVisible** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
boolean	A Boolean expression that specifies edges are displayed, as described in Settings.

Settings

The settings for *boolean* are:

Setting	Description
True	Edges are visible.
False	Edges are not displayed on elements in the three-dimensional chart.

See Also

Applies To: Light Object

Effect Property

Returns or sets the font effects in a chart.

Syntax

object.**Effect** [= *effects*]

The **Effect** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list .
effects	Integer. A **VtFontEffect** constant describing the font effect.

See Also

FontEffect Constants

Applies To: VtFont Object

FontEffect Constants

VtFontEffect provides methods of altering fonts for the desired effect.

Constant	Description
VtFontEffectStrikeThrough	Applies the strike-through attribute to the font.
VtFontEffectUnderline	Applies the underscore attribute to the font.

See Also

Effect Property

Elevation Property

Returns or sets a value that describes the degree of elevation from which a three-dimensional chart is viewed.

Syntax

object.**Elevation** [= *degree*]

The **Elevation** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
degree	Single. The degree of elevation. Elevation can be any number from 0 to 90 degrees. If you set the elevation to 90 degrees, you look directly down onto the top of the chart. If you set the elevation to 0, you look directly at the side of the chart. The de fault elevation is 30 degrees. By default, degrees are used to measure elevation. However, these settings us e the current settings for the AngleUnit property. The other options are: Grads and Radians.

See Also

Applies To: View3D Object

Excluded Property

Returns or sets a value that determines whether a series is included on the chart.

Syntax

object.**Excluded** [= *boolean*]

The **Excluded** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
boolean	A Boolean expression that controls whether a series is included on the chart, as described in Settings .

Settings

The settings for *boolean* are:

Setting	Description
True	The chart is drawn without including the series.
False	(Default) The series is included when the chart is drawn. A series may be included in a chart, but still not display because it is **Hidden**.

See Also

Hidden Property (MSChart)

Applies To: SeriesPosition Object

Fill Property

Returns a reference to a **Fill** object that describes the type and appearance of a chart object's backdrop.

Syntax

object.**Fill**

The object placeholder represents an object expression that evaluates to an object in the Applies To list.

Example

See Fill Object.

See Also

Fill Object

Applies To: Backdrop Object

FillColor Property

Returns a reference to a **VtColor** object that describes the color used to fill a chart element.

Syntax

object.**FillColor**

The object placeholder represents an object expression that evaluates to an object in the Applies To list.

Remarks

If the **FillColor** property of the **Marker** object is set, then the **Marker** object's **Visible** property is automatically set to **True**.

Example

See VtColor Object.

See Also

FillStyle Constants (MSChart Control)

Applies To: Brush Object | Marker Object

Flag Property

Returns or sets which statistic lines are being displayed for a series.

Syntax

object.**Flag** [= *lines*]

The **Flag** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
lines	Integer. A VtChStats constant used to describe the stat line. If more than one statistics line is displayed, the constants are combined with an OR operator.

See Also

StatsType Constants

Applies To: StatLine Object

Font Property (MSChart)

Returns a reference to a standard **Font** object that describes the font used to display text on the chart.

Syntax

object.**Font**

The object placeholder represents an object expression that evaluates to an object in the Applies To list.

See Also

Applies To: AxisTitle Object | DataPointLabel Object | Footnote Object | Label Object | Legend Object | Title Object

Footnote Property

Returns a reference to a **Footnote** object that provides information about the descriptive text used to annotate a chart.

Syntax

object.**Footnote**

The object placeholder represents an object expression that evaluates to an object in the Applies To list.

See Also

Footnote Object

Applies To: MSChart Control

FootnoteText Property

Returns or sets the text used as the footnote.

Syntax

object.**FootnoteText** [= *text*]

The **FootnoteText** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list .
text	String. The footnote text.

Remarks

The same results can be achieved by using the **Text** property of the **Footnote** object.

See Also

Applies To: MSChart Control

Format Property (Label Object)

Returns or sets the characters that define the format used to display the axis label.

Syntax

object.**Format** [= *format*]

The **Format** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list .
format	String. Defines the format used to display the axis label.

See Also

Applies To: Label Object

FormatLength Property

Returns the length of the format string.

Syntax

object.**FormatLength**

The object placeholder represents an object expression that evaluates to an object in the Applies To list.

Return Value

Integer

See Also

Applies To: Label Object

Frame Property

Returns a reference to a **Frame** object that describes the appearance of the frame around a chart element.

Syntax

object.**Frame**

The object placeholder represents an object expression that evaluates to an object in the Applies To list.

See Also

Frame Object

Applies To: Backdrop Object

FrameColor Property

Returns a reference to a **VtColor** object that specifies the color used to frame a chart element.

Syntax

*object.***FrameColor**

The object placeholder represents an object expression that evaluates to an object in the Applies To list.

Example

See Frame Object.

See Also

SpaceColor Property

Applies To: Frame Object

Green Property

Returns or sets the green component of the RGB value in a chart.

Syntax

object.**Green [**=*g***]**

The **Green p**roperty syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list .
g	Integer. The green value.

Remarks

RGB specifies the relative intensity of red, green, and blue to cause a specific color to be displayed. The valid range for a normal RGB color is 0 to 16,777,215. The value for any argument to RGB that exceeds 255 is assumed to be 255.

See Also

Applies To: VtColor Object

GuidelinePen Property

Returns a reference to a **Pen** object that describes the pattern of line and color used to display guidelines.

Syntax

*object.***GuidelinePen**

The object placeholder represents an object expression that evaluates to an object in the Applies To list.

Remarks

Setting this property automatically sets the **ShowGuideLines** property to **True**.

Example

See Pen Object.

See Also

Applies To: Series Object

Hidden Property (MSChart)

Returns or sets a value that determines whether a series is displayed on the chart.

Syntax

object.**Hidden** [= *boolean*]

The **Hidden** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
boolean	A Boolean expression that controls whether a series is displayed on the chart, as described in Settings .

Settings

The settings for *boolean* are:

Setting	Description
True	The chart is drawn without displaying the series. However, any space allocated for the serie s still exists.
False	(Default) The series is displayed.

See Also

Applies To: SeriesPosition Object

Hide Property

Returns or sets a value that determines whether the axis on a chart is hidden.

Syntax

object.**Hide** [= *boolean*]

The **Hide** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
boolean	A Boolean expression that specifies whether the axis is hidden, as described in Settings .

Settings

The settings for *boolean* are:

Setting	Description
True	The axis scale, line, ticks and title are hidden .
False	(Default) The axis appears on the chart.

See Also

Applies To: AxisScale Object

HorzAlignment Property

Returns or sets the method of horizontal alignment of text.

Syntax

object.**HorzAlignment** [= *type*]

The **HorzAlignment** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
type	Integer. A VtHorizontalAlignment constant used to describe the horizontal alignment method of text .

See Also

HorizontalAlignment Constants

Applies To: TextLayout Object

HorizontalAlignment Constants

The **VtHorizontalAlignment** constants provide options for text alignment.

Constant	Description
VtHorizontalAlignmentLeft	All lines of text are aligned on the left margin.
VtHorizontalAlignmentRight	All lines of text are aligned on the right margin .
VtHorizontalAlignmentCenter	All lines of text are centered horizontally.

See Also

HorzAlignment Property

Index Property (Brush)

Returns or sets the pattern or hatch used in the brush if its **Style** property is set to **VtBrushStylePattern** or **VtBrushStyleHatch**.

Syntax

object.**Index** [= *num*]

The **Index property syntax has** these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
num	A VtBrushPattern constant or VtBrushHatch constant describing the brush pattern .

See Also

BrushHatch Constants | BrushPattern Constants | Index Property (Intersection)

Applies To: Brush Object

Index Property (Intersection)

Returns which axis intersects another axis when there is more than one axis with the same index.

Syntax

object.**Index**

The object placeholder represents an object expression that evaluates to an object in the Applies To list.

Return Value

The return value is an integer that specifies the index of the intersecting axis. Currently, 1 is the only valid value for this argument.

See Also

Index Property (Brush)

Applies To: Intersection Object

Intensity Property

Returns or sets the strength of the light coming from the light source.

Syntax

object.**Intensity** [= *strength*]

The **Intensity** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
strength	Single. The light strength. If the intensity is set to 100 percent (1), chart surfaces facing the light source are fully illuminated. If the light is set at 50 percent (.5), these surfaces receive 50 percent illumination from th is light. Valid range is 0 to 1.

Remarks

Intensity is the default property of the **LightSource** object.

See Also

Applies To: LightSource Object

Intersection Property

Returns a reference to an **Intersection** object that describes the point at which an axis intersects another axis on a chart.

Syntax

object.**Intersection**

The object placeholder represents an object expression that evaluates to an object in the Applies To list.

See Also

Intersection Object

Applies To: Axis Object

Item Property (MSChart)

Returns a reference to an object within a collection that describes a chart element.

Syntax

object.**Item** (*index*)

The **Item** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list .
index	An unique integer that identifies the member of the collection.

See Also

Applies To: DataPoints Collection | Labels Collection | LightSources Collection | SeriesCollection Collection

Join Property

Returns or sets a value that determines how line segments are formed.

Syntax

object.**Join** [= *type*]

The **Join** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list .
type	Integer. A VtPenJoin constant that describes the style of pen join.

See Also

PenJoin Constants

Applies To: Pen Object

PenJoin Constants

The **VtPenJoin** constants provide options for joining line segments in a series.

Constant	Description
VtPenJoinMiter	The outer edges of the two lines are extended until they meet.
VtPenJoinRound	A circular arc is drawn around the point where the two lines meet .
VtPenJoinBevel	The notch between the ends of two joining lines is filled.

See Also

Join Property

LabelLevelCount Property

Returns the number of levels of labels for a given axis. Not available at design time.

Syntax

object.**LabelLevelCount** [= *count*]

The **LabelLevelCount** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list .
count	Integer. An integer that describes the number of labels.

See Also

Applies To: Axis Object

Labels Property

Returns a reference to a **Labels** collection that describes the labels on a chart axis.

Syntax

object.**Labels**

The object placeholder represents an object expression that evaluates to an object in the Applies To list.

See Also

Label Object | Labels Collection

Applies To: Axis Object

LabelsInsidePlot Property

Returns or sets a value that determines whether to leave the axis labels at the normal location or move them with the axis to the new intersection point.

Syntax

object.**LabelsInsidePlot** [= *boolean*]

The **LabelsInsidePlot** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
boolean	A Boolean expression that specifies where to display the axis labels, as described in Settings.

Settings

The settings for *boolean* are:

Setting	Description
True	(Default) The axis labels remain at the normal location.
False	The labels move inside the plot to the new intersection point.

Remarks

If this property is set, then the Intersection object's **Auto** property is automatically set to **False**.

See Also

Applies To: Intersection Object

LabelTick Property

Returns or sets a value that indicates whether category axis labels are centered on an axis tick mark.

Syntax

object.**LabelTick** [= *boolean*]

The **LabelTick** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
boolean	A Boolean expression that specifies whether the item is displayed, as described in Settings.

Settings

The settings for *boolean* are:

Setting	Description
True	The labels are centered on a tick mark.
False	(Default) The labels are centered between two tick marks.

Remarks

If this property is set, the object's **Auto** property is automatically set to **False**.

See Also

Auto Property (CategoryScale)

Applies To: CategoryScale Object

Legend Property

Returns a reference to a **Legend** object that contains information about the appearance and behavior of the graphical key and accompanying text that describes the chart series.

Syntax

object.**Legend**

The object placeholder represents an object expression that evaluates to an object in the Applies To list.

See Also

Applies To: MSChart Control

LegendText Property

Returns or sets the text that identifies the series in the legend of a chart.

Syntax

object.**LegendText** [= *text*]

The **LegendText** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list .
text	String. The text that identifies the current series in the legend.

Remarks

By default, this text is the same as the **Text** property of the **ColumnLabel** object.

See Also

Applies To: Series Object

Length Property (MSChart)

Returns or sets the length of axis tick marks, measured in points.

Syntax

object.**Length** [**=** *length*]

The **Length** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list .
length	Integer. The axis tick mark length.

See Also

Applies To: Tick Object

Light Property

Returns a reference to a **Light** object that provides information about the light illuminating a three-dimensional chart.

Syntax

object.**Light**

The object placeholder represents an object expression that evaluates to an object in the Applies To list.

See Also

Applies To: Plot Object

LightSources Property

Returns a reference to a **LightSources** collection that describes all light sources used to illuminate a three-dimensional chart.

Syntax

object.**LightSources**

The object placeholder represents an object expression that evaluates to an object in the Applies To list.

See Also

LightSources Collection

Applies To: Light Object

Limit Property

Returns or sets the joint limit, in points, of the line.

Syntax

object.**Limit** [= *joint*]

The **Limit** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
joint	Single. A joint limit as a multiple of the line width. If two lines meet at a sharp angle, a mitered join results in a corner point that extends beyond the actual corner. If the distance from the inner join point to the outer join point exceeds the value in this variable, the join automatically changes to a bevel.

See Also

Applies To: Pen Object

LineStyle Property (DataPointLabel Object)

Returns or sets the type of line used to connect a data point to a label on a chart.

Syntax

object.**LineStyle** [= *type*]

The **LineStyle** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
type	Integer. A VtChLabelLineStyle constant identifying the connecting line.

See Also

LabelLineStyle Constants

Applies To: DataPointLabel Object

LabelLineStyle Constants

The **VtChLabelLineStyle** constants provide options for displaying lines connecting a label and series.

The following table lists the valid constants for **VtChLabelLineStyle**:

Constants	Description
VtChLabelLineStyleNone	No line connects the label and series.
VtChLabelLineStyleStraight	A straight line connects the label and series.
VtChLabelLineStyleBent	A bent line connects the label and series.

See Also

LineStyle Property (DataPointLabel Object)

Location Property

Returns a reference to a **Location** object that describes the position of textual chart elements.

Syntax

object.**Location**

The object placeholder represents an object expression that evaluates to an object in the Applies To list.

See Also

Location Object

Applies To: Footnote Object | Legend Object | Title Object

LocationRect Property

Returns a reference to a **Rect** object that specifies the location of the chart plot using *x* and *y* coordinates.

Syntax

*object.***LocationRect**

The object placeholder represents an object expression that evaluates to an object in the Applies To list.

Remarks

The values of this property are used to position the plot if **AutoLayout** is **False.**

If this property is set, then the **AutoLayout** property is automatically set to **False**.

Example

See Rect Object.

See Also

Applies To: Plot Object

LocationType Property

Returns or sets the standard position used to display a chart element.

Syntax

object.**LocationType** [=*type*]

The **LocationType** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
type	Integer. For the **DataPointLabel** object, a VtChLabelLocationType constant identifying label position . For a **Location** object, a VtChLocationType constant describing the location of text.

See Also

LocationType Constants

Applies To: DataPointLabel Object | Location Object

LocationType Constants

VtChLocationType provides location options for chart elements.

Constant	Example
VtChLocationTypeTop	Top
VtChLocationTypeTopLeft	Top Left
VtChLocationTypeTopRight	Top Right
VtChLocationTypeLeft	Left
VtChLocationTypeRight	Right
VtChLocationTypeBottom	Bottom
VtChLocationTypeBottomLeft	Bottom Left
VtChLocationTypeBottomRight	Bottom Right
VtChLocationTypeCustom	Custom

See Also

LocationType Property

LabelLocationType Constants

The **VtChLabelLocationType** constants provide options for determining series label location.

The following table lists the valid constants for **VtChLabelLocationType**:

Constants	Description
VtChLabelLocationTypeNone	No label displayed.
VtChLabelLocationTypeAbovePoint	The label is displayed above the data point.
VtChLabelLocationTypeBelowPoint	The label is displayed below the data point.
VtChLabelLocationTypeCenter	The label is displayed centered on the data point.
VtChLabelLocationTypeBase	The label is displayed at the base along the category axis, dir ectly beneath the data point.
VtChLabelLocationTypeInside	The label is displayed inside a pie slice.
VtChLabelLocationTypeOutside	The label is displayed outside a pie slice.
VtChLabelLocationTypeLeft	The label is displayed to the left of the data point.
VtChLabelLocationTypeRight	The label is displayed to the right of the data point.

LogBase Property

Returns or sets the logarithm base used to plot chart values on a logarithmic axis.

Syntax

object.**LogBase** [= *base*]

The **LogBase** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
base	Integer. An integer that identifies the logarithm base. The default base is 10. The valid range is 2 to 100 .

Remarks

This property has an effect only when the **Type** property is set to **VtChScaleTypeLogarithmic**.

See Also

Applies To: AxisScale Object

MajorDivision Property

Returns or sets the number of major divisions displayed on the axis.

Syntax

object.**MajorDivision** [= *num*]

The **MajorDivision** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list .
num	Integer. Number of divisions.

Remarks

If this property is set, then the **ValueScale** object's **Auto** property is automatically set to **False**.

See Also

MinorDivision Property

Applies To: ValueScale Object

MajorPen Property

Returns a reference to a **Pen** object that describes the appearance of the major axis grid lines.

Syntax

*object.***MajorPen**

The object placeholder represents an object expression that evaluates to an object in the Applies To list.

See Also

MinorPen Property

Applies To: AxisGrid Object

Marker Property

Returns a reference to a **Marker** object that describes the icon used to identify a data point on a chart.

Syntax

object.**Marker**

The object placeholder represents an object expression that evaluates to an object in the Applies To list.

See Also

Applies To: DataPoint Object

Max Property (MSChart)

Returns a reference to a **Coor** object that specifies the ending corner of a rectangle.

Syntax

object.**Max**

The object placeholder represents an object expression that evaluates to an object in the Applies To list.

See Also

Min Property (MSChart)

Applies To: Rect Object

Maximum Property

Returns or sets the highest or ending value on the chart value axis.

Syntax

object.**Maximum** [= *value*]

The **Maximum** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list .
value	Double. The highest axis value.

Remarks

If this property is set, then the **ValueScale** object's **Auto** property is automatically set to **False**.

The **Maximum** property should be set before the **Minimum** property to avoid a chart display error.

See Also

Minimum Property

Applies To: ValueScale Object

Min Property (MSChart)

Returns a reference to a **Coor** object that specifies the starting corner of a rectangle.

Syntax

object.**Min**

The object placeholder represents an object expression that evaluates to an object in the Applies To list.

See Also

Max Property (MSChart)

Applies To: Rect Object

Minimum Property

Returns or sets the lowest or beginning value on the chart value axis.

Syntax

object.**Minimum** [= *value*]

The **Minimum** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list .
value	Double. The lowest axis value.

Remarks

If this property is set, then the **ValueScale** object's **Auto** property is automatically set to **False**.

The **Maximum** property should be set before the **Minimum** property to avoid a chart display error.

See Also

MajorDivision Property

Applies To: ValueScale Object

MinorDivision Property

Returns or sets the number of minor divisions displayed on the axis.

Syntax

object.**MinorDivision** [= *num*]

The **MinorDivision** property syntax has these parts:

Part	Description
Object	An object expression that evaluates to an object in the Applies To list .
num	Integer. Number of minor divisions.

Remarks

If this property is set, then the **ValueScale** object's **Auto** property is automatically set to **False**.

See Also

Applies To: ValueScale Object

MinorPen Property

Returns a reference to a **Pen** object that describes the appearance of the minor axis grid lines.

Syntax

*object.***MinorPen**

The object placeholder represents an object expression that evaluates to an object in the Applies To list.

See Also

MajorPen Property

Applies To: AxisGrid Object

MousePointer Property (MSChart)

Returns or sets a value indicating the type of mouse pointer displayed when the mouse is over a particular part of an object at run time.

Syntax

object.**MousePointer** [= *value*]

The **MousePointer** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
value	Integer. A VtMousePointer constant that specifies the type of mouse pointer displayed.

See Also

MousePointer Constants

Applies To: MSChart Control

MousePointer Constants

The **VtMousePointer** constants provide the following types of pointers.

Constant	Description
VtMousePointerArrow	Arrow pointer
VtMousePointerArrowHourGlass	Arrow and hourglass
VtMousePointerArrowQuestion	Arrow and question mark
VtMousePointerCross	Crosshair pointer
VtMousePointerDefault	Default chart pointer
VtMousePointerHourGlass	Hourglass pointer
VtMousePointerIbeam	Ibeam pointer
VtMousePointerIcon	Small square within a square pointer
VtMousePointerNoDrop	No drop pointer
VtMousePointerSize	Sizing arrows
VtMousePointerSizeAll	Sizing in all directions arrows
VtMousePointerSizeNESW	Double arrow pointing northeast and southwest
VtMousePointerNS	Double arrow pointing north and south
VtMousePointerNWSE	Double arrow pointing northwest and southeast
VtMousePointerWE	Double arrow pointing east and west
VtMousePointerUpArrow	Arrow pointing up

See Also

MousePointer Property (MSChart)

Name Property (VtFont)

Returns or sets the name of the font. This is the default property of the **VtFont** object.

Syntax

object.**Name** [= *text*]

The **Name** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list .
text	Integer. The text containing the font name.

See Also

Applies To: MSChart Control | VtFont Object

Offset Property

Returns or sets the distance that a chart element is offset or pulled away from its default location.

Syntax

object.**Offset** [= *offset*]

The **Offset** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
offset	For the **DataPoint** object, this is an integer describing the offset distance. Offset is measured in inches or centimeters depending upon your default Windows settings. For the **DataPointLabel** and **Shadow** objects, this is a reference to a **Coor** object that describe the *x* and *y* values of the offset.

Remarks

For the **DataPointLabel** object, this property indicates the distance that a data point label is offset or pulled away from one of the predefined (standard) label positions. The offset is added to the position calculated for the point based on the **DataPointLabel** object's **LocationType** setting.

See Also

Applies To: DataPoint Object | DataPointLabel Object | Shadow Object

Order Property

Returns or sets the position of the series in the chart. If the position in order matches another series, the series are stacked.

Syntax

object.**Order** [= *order*]

The **Order** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list .
order	Integer. The position order.

See Also

Applies To: SeriesPosition Object

Orientation Property (MSChart)

Returns or sets the method of orientation for text.

Syntax

object.**Orientation** [= *type*]

The **Orientation** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
type	Integer. A VtOrientation constant used to describe the orientation method .

See Also

Orientation Constants

Applies To: TextLayout Object

Orientation Constants

The **VtOrientation** constants provide options for positioning text.

Constant	Description
VtOrientationHorizontal	The text is displayed horizontally.
VtOrientationVertical	The letters of the text are drawn one on top of each other from the top down .
VtOrientationUp	The text is rotated to read from bottom to top.
VtOrientationDown	The text is rotated to read from top to bottom.

See Also

Orientation Property (MSChart)

PatternColor Property

Returns a reference to a **VtColor** object that describes the pattern color used to fill a chart element.

Syntax

object.**PatternColor**

The object placeholder represents an object expression that evaluates to an object in the Applies To list.

See Also

Applies To: Brush Object

Pen Property

Returns or sets a reference to a **Pen** object that describes the color and pattern of lines or edges on chart elements.

Syntax

object.**Pen**

The object placeholder represents an object expression that evaluates to an object in the Applies To list.

See Also

Pen Object

Applies To: Axis Object | Marker Object | PlotBase Object | Series Object | Wall Object

PercentBasis Property

Returns or sets the type of percentage used to plot chart values on a percent axis.

Syntax

object.**PercentBasis** [= *type*]

The **PercentBasis** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
type	A VtChPercentAxisBasis constant used to describe the percentage used to plot percent axis values .

Remarks

This property has an effect only when the **Type** property is set to **VtChScaleTypePercent**.

See Also

PercentAxisBasis Constants | Type Property (MSChart)

Applies To: AxisScale Object

PercentAxisBasis Constants

VtChPercentAxisBasis provides methods of displaying percentage axes.

Constant	Description
VtChPercentAxisBasisMaxChart	The largest value in the chart is considered 100 percent and all other values on the chart are displayed as percentages of that value.
VtChPercentAxisBasisMaxRow	The largest value in each row is considered 100 percent and all other values in that row are displayed as percentages of that value.
VtChPercentAxisBasisMaxColumn	The largest value in each series is considered 100 percent and all other values in that series are displayed as percentages of that value.
VtChPercentAxisBasisSumChart	All values in the chart are added together, and that value is considered 100 percent. All other values are displayed as percentages of that value.
VtChPercentAxisBasisSumRow	All values in each row are added together and the total value for each row is considered 100 percent. All other values in that same row are displayed as percentages of that value. This is the basis for 100 percent stacked charts.
VtChPercentAxisBasisSumColumn	All values in each series are added together to give a total value for each series. All values are displayed as a percentage of their series total value.

See Also

PercentBasis Property

PercentFormat Property

Returns or sets a string that describes the format used to display the label as a percent.

Syntax

object.**PercentFormat** [= *format*]

The **PercentFormat** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list .
format	String. Describes the format used to display a label as a percent.

Remarks

Use the **DataPointLabel** object's **Component** property to change the label type.

The following table lists several examples of percentage format strings. The values listed at left are the valid formats.

	3	-3	.3
0%	300%	-300%	30%
0.0%	300.0%	-300.0%	30.0%
0.00%	300.00%	-300.00%	30.00%

See Also

Applies To: DataPointLabel Object

Plot Property

Returns a reference to a **Plot** object that describes the area upon which a chart is displayed.

Syntax

object.**Plot**

The object placeholder represents an object expression that evaluates to an object in the Applies To list.

See Also

Applies To: MSChart Control

PlotBase Property

Returns a reference to a **PlotBase** object that describes the appearance of the area beneath a chart.

Syntax

object.**PlotBase**

The object placeholder represents an object expression that evaluates to an object in the Applies To list.

See Also

Applies To: Plot Object

Point Property

Returns or sets the point where the current axis intersects with another axis.

Syntax

object.**Point** [= *point*]

The **Point** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list .
point	Double. The point on an axis where the current axis intersects.

Remarks

If this property is set, then the Intersection object's **Auto** property is automatically set to **False**.

See Also

Applies To: Intersection Object

Position Property (MSChart)

Returns a reference to a **SeriesPosition** object that describes the location of one series in relation to other chart series.

Syntax

object.**Position**

The object placeholder represents an object expression that evaluates to an object in the Applies To list.

See Also

SeriesPosition Object

Applies To: Series Object

Projection Property

Returns or sets the type of projection used to display the chart.

Syntax

object.**Projection** [= *type*]

The **Projection** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
type	Integer. A VtProjectionType constant used to describe the type of chart projection .

Remarks

The **Projection** property determines the appearance of 3 dimensional charts.

Example

The example allows you to see the different effects obtainable by setting the **Projection** property. To try the example, draw an **MSChart** and a **ComboBox** control on a form. Paste the code into the Declarations section, and press F5. Click the **ComboBox** to see the different settings.

```
Option Explicit

Private Sub Combo1_Click()
   ' Change the projection when clicked.
   MSChart1.Plot.Projection = Combo1.ListIndex
End Sub

Private Sub Form_Load()
   ' Set the chart to a 3D type.
   MSChart1.chartType = VtChChartType3dBar
   ' Configure the ComboBox by adding the valid settings.
   With Combo1
      .AddItem "VtProjectionTypePerspective" ' 0
      .AddItem "VtProjectionTypeOblique"      ' 1
      .AddItem "VtProjectionTypeOrthogonal"  ' 2
      .AddItem "VtProjectionTypeFrontal"      ' 3
      .AddItem "VtProjectionTypeOverhead"    ' 4
      .ListIndex = 0
   End With
End Sub
```

See Also

ProjectionType Constants

Applies To: Plot Object

ProjectionType Constants

VtProjectionType provides viewpoint and perspective options for displaying and viewing a chart.

Constants	Description
VtProjectionTypePerspective	This provides the most realistic three-dimensional appearance. Objects farther away from you converge toward a vanishing point. This is the default projection.
VtProjectionTypeOblique	This is sometimes referred to as 2.5 dimensional. The chart does have depth, but the xy plane does not change when the chart is rotated or elevated.
VtProjectionTypeOrthogonal	Perspective is not applied in this three-dimensional view. The major advantage of using this type of projection is that vertical lines remain vertical, making some charts easier to read.

See Also

Projection Property

RandomFill Property

Indicates whether the data for a chart data grid was randomly generated.

Syntax

object.**RandomFill** [= *boolean*]

The **RandomFill** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
boolean	A Boolean expression that controls how data is generated, as described in Settings .

Settings

The settings for *boolean* are:

Setting	Description
True	(Default) Random data is used to draw the chart.
False	No random data is generated. The user provides the data for the chart .

See Also

RandomDataFill Method | RandomFillColumns Method | RandomFillRows Method

Applies To: MSChart Control

Rect Property

Returns a reference to a **Rect** object that defines a coordinate location.

Syntax

object.**Rect**

The object placeholder represents an object expression that evaluates to an object in the Applies To list.

See Also

Rect Object

Applies To: Location Object

Red Property

Returns or sets the red component of the RGB value in a chart.

Syntax

object.**Red [=***r***]**

The **Red** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list .
r	Integer. The red value.

Remarks

RGB specifies the relative intensity of red, green, and blue to cause a specific color to be displayed. The valid range for a normal RGB color is 0 to 16,777,215. The value for any argument to RGB that exceeds 255 is assumed to be 255.

See Also

Applies To: VtColor Object

Repaint Property

Returns or sets a value that determines if the **MSChart** control is repainted after a change is made to the chart.

Syntax

object.**Repaint** [= *boolean*]

The **Repaint** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
boolean	A Boolean expression that controls whether the chart is repainted, as described in Settings .

Settings

The settings for *boolean* are:

Setting	Description
True	(Default) Refreshes the control.
False	Does not allow the control to repaint when a change is made to the chart. This is useful when several operations are performed on the chart and you do not want the chart to continually repaint during the process.

See Also

Applies To: MSChart Control

Rotation Property

Returns or sets a value that describes the degree of rotation from which a three-dimensional chart is viewed.

Syntax

object.**Rotation** [= *degree*]

The **Rotation** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
degree	Single. The degree of rotation. Rotation can range from 0 to 360 degrees. By default, degrees are used to measure rotation. However, thes e settings use the current settings for the AngleUnit property. The other options are: Grads and Radians.

See Also

Applies To: View3D Object

Row Property (MSChart)

Returns or sets a specific row in the current column of a data grid associated with a chart.

Syntax

object.**Row** [= *num*]

The **Row** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
num	Integer. A row number in the current column. Rows are numbered from top to bottom beginning with 1 .

See Also

Applies To: MSChart Control

RowCount Property (MSChart)

Returns or sets how many rows there are in each column of a data grid associated with a chart.

Syntax

object.**RowCount** [= *count*]

The **RowCount** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list .
count	Integer. The number of rows in a column.

Remarks

The virtual data grid is represented by the **DataGrid** object.

See Also

Applies To: DataGrid Object | MSChart Control

RowLabel Property (DataGrid Object)

Returns or sets a specific row label in the current data grid associated with a chart.

Syntax

object.**RowLabel** (*row*, *labelIndex*) [= *text*]

The **RowLabel** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
row	Integer. Specifies a row. Rows are numbered from top to bottom beginning at 1.
labelIndex	Integer. Specifies a specific level of row labels. Row labels are numbered from left to right beginning at 1.
text	String. Text of the row label.

See Also

RowLabel Property (MSChart)

Applies To: DataGrid Object

RowLabel Property (MSChart)

Returns or sets a data label that can be used to identify the current data point in a chart.

Syntax

object.**RowLabel** [= *text*]

The **RowLabel** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
text	String. The text for a row label. The label you specify sets the label for the data points identified by the **Row** property. This label appears along the category axis for most chart types and is used as the label for each in dividual pie in a pie chart. Label text may not be displayed if it is too long to fit on a chart.

See Also

RowLabel Property (DataGrid Object)

Applies To: MSChart Control

RowLabelCount Property

Returns or sets the number of levels of labels on the rows in a data grid associated with a chart.

Syntax

object.**RowLabelCount** [= *count*]

The **RowLabelCount** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
count	Integer. The number of label levels. Set this property to add or delete levels of labels from data grid rows. Row label levels are numbered from right to left, beginning at 1. Levels are added or subtracted from the left.

See Also

SetSize Method

Applies To: DataGrid Object | MSChart Control

RowLabelIndex Property

Returns or sets a value that specifies a level of row labels.

Syntax

object.**RowLabelIndex** [= *index*]

The **RowLabelIndex** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
index	Integer. A row label level. To set a label on a row with more than one level of labels, or to return the current value for a label, you must first identify which level you want to affect. Row label levels are numbered from right to left, beginning at 1.

See Also

Applies To: MSChart Control

SecondaryAxis Property

Returns or sets a value that determines whether the series is charted on the secondary axis.

Syntax

object.**SecondaryAxis** [= *boolean*]

The **SecondaryAxis** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
boolean	A Boolean expression that controls whether the series is charted on the secondary axis, as described in Settings.

Settings

The settings for *boolean* are:

Setting	Description
True	The series is charted on the secondary axis.
False	(Default) The series is not charted on the secondary axis.

See Also

Applies To: Series Object

SeriesCollection Property

Returns a reference to a **SeriesCollection** collection that provides information about the series that make up a chart.

Syntax

object.**SeriesCollection**

The object placeholder represents an object expression that evaluates to an object in the Applies To list.

See Also

Applies To: Plot Object

SeriesColumn Property

Returns or sets the column position for the current series data.

Syntax

object.**SeriesColumn** [= *pos*]

The **SeriesColumn** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
pos	Integer. The position of the column containing the current series data. You can use this property to reorder series. If two series are assigned the same position, they are stacked.

See Also

Applies To: MSChart Control

SeriesMarker Property

Returns a reference to a **SeriesMarker** object that describes a marker that identifies all data points within one series on a chart.

Syntax

object.**SeriesMarker**

The object placeholder represents an object expression that evaluates to an object in the Applies To list.

See Also

Applies To: Series Object

SeriesType Property

Returns or sets the type used to display the current series.

Syntax

object.**SeriesType** [= *type*]

The **SeriesType** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
type	Integer. A VtChSeriesType constant describing the method used to display the series. You must select the series to change using the **Column** property before using the **SeriesType** property.

See Also

SeriesType Constants

Applies To: MSChart Control | Series Object

SeriesType Constants

The **VtChSeriesType** constants provide options for types of series.

Constant	Series Type
VtChSeriesType3dBar	3D Bar
VtChSeriesType2dBar	2D Bar
VtChSeriesType3dLine	3D Line
VtChSeriesType2dLine	2D Line
VtChSeriesType3dArea	3D Area
VtChSeriesType2dArea	2D Area
VtChSeriesType3dStep	3D Step
VtChSeriesType2dStep	2D Step
VtChSeriesType2dXY	XY
VtChSeriesType2dPie	2D Pie

See Also

SeriesType Property | TypeByChartType Method

Shadow Property

Returns a reference to a **Shadow** object that describes the appearance of a shadow on chart elements.

Syntax

object.**Shadow**

The object placeholder represents an object expression that evaluates to an object in the Applies To list.

See Also

Applies To: Backdrop Object

Show Property

Returns or sets a value that determines whether series markers are displayed on a chart.

Syntax

object.**Show** [= *boolean*]

The **Show** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
boolean	A Boolean expression that controls whether series markers are displayed, as described in Settings .

Settings

The settings for *boolean* are:

Setting	Description
True	Series markers are displayed
False	(Default) Series markers are not displayed .

See Also

Applies To: SeriesMarker Object

ShowGuideLine Property

Returns or sets a value that determines whether or not the connecting data point lines on a chart are displayed for a series.

Syntax

object.**ShowGuideLines** (*axisId*, *index*) [= *boolean*]

The **ShowGuideLine** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
axisId	Integer. A VtChAxisId constant describing the series axis you want to set this property for.
index	Integer. An integer reserved for future use. For this version of MSChart control, 1 is the only valid value for this argument.
boolean	A Boolean expression that controls whether the series is charted on the secondary axis, as described in Settings.

Settings

The settings for *boolean* are:

Setting	Description
True	The series guidelines are displayed.
False	(Default) The series guidelines are not displayed.

See Also

AxisId Constants

Applies To: Series Object

ShowLegend Property

Returns or sets a value that indicates whether a legend is visible for a chart.

Syntax

object.**ShowLegend** [= *boolean*]

The **ShowLegend** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
boolean	A Boolean expression that controls whether a legend is displayed on the chart, as described in Settings .

Settings

The settings for *boolean* are:

Setting	Description
True	The legend appears on the chart in the position indicated by the **Location** object.
False	(Default) The legend is not displayed on the chart. The default legend location is to the right side of the chart.

Remarks

The default legend location is to the right side of the chart.

See Also

Applies To: MSChart Control

ShowLine Property

Returns or sets a value that determines whether the lines connecting data points on a chart are visible.

Syntax

object.**ShowLine** [= *boolean*]

The **ShowLine** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
boolean	A Boolean expression that controls whether lines are displayed, as described in Settings .

Settings

The settings for *boolean* are:

Setting	Description
True	(Default) The lines connecting data points appear on the chart .
False	The data point lines do not appear.

See Also

Applies To: Series Object

Size Property (MSChart)

Returns or sets the size of a chart element in points.

Syntax

object.**Size** [= *size*]

The **Size** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list .
size	Single. Size of the chart element in points.

Applies To: Marker Object | VtFont Object

Sort Property (MSChart)

Returns or sets the type of sort order used in a pie chart.

Syntax

object.**Sort** [= *type*]

The **Sort** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list .
type	Integer. A VtSortType constant used to describe the plot sort order.

See Also

SortType Constants

Applies To: Plot Object

SortType Constants

The **VtSortType** constants provide options for sorting pie charts.

Constant	Description
VtSortTypeNone	Pie slices are drawn in the order the data appears in the data grid.
VtSortTypeAscending	Pie slices are drawn, in order, from the smallest to the largest slice, starting at the defined starting angle and in the defined plot direction.
VtSortTypeDescending	Pie slices are drawn, in order, from the largest to the smallest slice, starting at the defined starting angle and in the defined plot direction.

See Also

Sort Property (MSChart)

SpaceColor Property

Returns a reference to a **VtColor** object that specifies the color used fill the space between double frames around a chart element.

Syntax

object.**SpaceColor**

The object placeholder represents an object expression that evaluates to an object in the Applies To list.

Example

See Frame Object.

See Also

FrameColor Property

Applies To: Frame Object

Stacking Property

Sets a value that determines whether all the series in the chart are stacked.

Syntax

object.**Stacking** [= *boolean*]

The **Stacking** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
boolean	A Boolean expression that controls whether all chart series are stacked, as described in Settings .

Settings

The settings for *boolean* are:

Setting	Description
True	All chart series are stacked.
False	(Default) Chart series are not stacked .

See Also

Applies To: MSChart Control

StackOrder Property

Returns or sets in what position the current series is drawn if it is stacked with other series.

Syntax

object.**StackOrder** [= *position*]

The **StackOrder** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
position	Integer. The order of the series if stacked with other series. Lower stack orders are on the bottom of the stack.

See Also

Applies To: SeriesPosition Object

Standing Property

Returns or sets a value that specifies whether axis labels are displayed horizontally in the *x* or *z* plane or vertically on the text baseline in the *y* plane.

Syntax

object.**Standing** [= *boolean*]

The **Standing** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
boolean	A Boolean expression that specifies how to display the axis labels, as described in Settings .

Settings

The settings for *boolean* are:

Setting	Description
True	The axis labels are displayed vertically on the text baseline in the *y* plane .
False	(Default) The axis labels are displayed horizontally in the *x* or *z* plane.

See Also

Applies To: Label Object

StartingAngle Property

Returns or sets the position where you want to start drawing pie charts.

Syntax

object.**StartingAngle** [= *angle*]

The **StartingAngle** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
angle	Single. This angle can be measured in degrees, radians, or grads, depending on the current AngleUnits setting. A value of 0 degrees indicates the 3 o'clock position. Setting the starting angle to 90 degrees moves the starting position to 12 o'clock if the Clockwise property is set to counterclockwise, or to 6 o'clock if it is set to clockwise. Valid values range from -360 to 360 degrees.

Applies To: Plot Object

StatLine Property

Returns a reference to a **StatLine** object that describes how statistic lines are displayed on a chart.

Syntax

object.**StatLine**

The object placeholder represents an object expression that evaluates to an object in the Applies To list.

See Also

Applies To: Series Object

Style Property (MSChart Control)

Returns or sets the style used to draw certain chart elements

Syntax

object.**Style** [=*style*]

The **Style** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
style	For the **Brush** object, a VtBrushStyle constant describing the brush pattern. For the **Fill** object, a VtFillStyle constant that describes the style of fill. A fill can have a brush, which is a solid color or patterned fill. For the **Frame** object, a VtFrameStyle constant that describes the type of frame. For the **Marker** object, a VtMarkerStyle constant that lists the marker type. For the **Pen** object, a VtPenStyle constant that describes the style of pen. For the **Shadow** object, a VtShadowStyle constant used to describe the shadow type. For the **Tick** object, a VtChAxisTickStyle constant used to describe the axis tick position. For the **VtFont** object, a VtFontStyle constant describing the style of font. For the **Weighting** object, a VtChPieWeightStyle constant that identifies the weighting factor meth od.

See Also

AxisTickStyle Constants | BorderStyle Constants (MSChart) | FillStyle Constants (MSChart Control) | FontStyle Constants | FrameStyle Constants | MarkerStyle Constants | PenStyle Constants | PieWeightStyle Constants | ShadowStyle Constants | StatsType Constants | Style Property (StatLine)

Applies To: Brush Object | Fill Object | Frame Object | Marker Object | Pen Object | Shadow Object | Tick Object | VtFont Object | Weighting Object

AxisTickStyle Constants

The **VtChAxisTickStyle** constants provide options for indicating axis tick mark location.

Constants	Description
VtChAxisTickStyleNone	No tick marks are displayed on the axis.
VtChAxisTickStyleCenter	Tick marks are centered across the axis.
VtChAxisTickStyleInside	Tick marks are displayed inside the axis.
VtChAxisTickStyleOutside	Tick marks are displayed outside the axis.

See Also

Style Property (MSChart Control)

BorderStyle Constants (MSChart)

The **VtBorderStyle** constants provide options for the type of border to be placed around the chart control.

Constant	Description
VtBorderStyleFixedSingle	A single border is placed around the chart control .
VtBorderStyleNone	No border is placed around the chart control.

BrushHatch Constants

VtBrushHatch provides valid brush types if **VtBrushStyle** is set to **VtBrushStyleHatch**.

Constant	Description
VtBrushHatchHorizontal	Horizontal hatch lines
VtBrushHatchVertical	Vertical hatch lines
VtBrushHatchDownDiagonal	Down diagonal hatch lines
VtBrushHatchUpDiagonal	Up diagonal hatch lines
VtBrushHatchCross	Cross hatch lines
VtBrushHatchDiagonalCross	Diagonal cross hatch lines

See Also

Index Property (Brush)

BrushStyle Constants

VtBrushStyle provides valid brush types.

Constant	Description
VtBrushStyleNull	No brush (background shows through)
VtBrushStyleSolid	Solid color brush
VtBrushStylePattern	Bitmap patterned brush
VtBrushStyleHatched	Hatched brush

See Also

Style Property (MSChart Control)

BrushPattern Constants

VtBrushPattern provides valid brush types if **VtBrushStyle** is set to **VtBrushStylePattern**.

Constant	Description
VtBrushPattern94percent	94 percent pattern color
VtBrushPattern88percent	88 percent pattern color
VtBrushPattern75percent	75 percent pattern color
VtBrushPattern50percent	50 percent pattern color
VtBrushPattern25percent	25 percent pattern color
VtBrushPatternBoldHorizontal	Bold horizontal lines
VtBrushPatternBoldVertical	Bold vertical lines
VtBrushPatternBoldDownDiagonal	Bold down diagonal lines
VtBrushPatternBoldUpDiagonal	Bold up diagonal lines
VtBrushPatternChecks	Checks pattern
VtBrushPatternWeave	Weave pattern
VtBrushPatternHorizontal	Horizontal lines
VtBrushPatternVertical	Vertical lines
VtBrushPatternDownDiagonal	Down diagonal lines
VtBrushPatternUpDiagonal	Up diagonal lines
VtBrushPatternGrid	Grid pattern
VtBrushPatternTrellis	Trellis pattern
VtBrushPatternInvertedTrellis	Inverted trellis pattern

See Also

Index Property (Brush)

FillStyle Constants (MSChart Control)

The **VtFillStyle** constants provide options for indicating the type of fill used to paint a backdrop.

Constant	Description
VtFillStyleNull	No fill (background shows through)
VtFillStyleBrush	A solid color or pattern fill

Remarks

The **FillStyle** constants are used with the **Style** property of the Fill object.

An example of the constant usage is shown below:

```
With MSChart1.Backdrop.Fill
   .Style = VtFillStyleBrush
   .Brush.FillColor.Set 0, 200, 250 ' Set the color of the fill.
End With
```

See Also

Brush Object | Fill Object | FillColor Property | Style Property (MSChart Control)

FontStyle Constants

VtFontStyle provides valid font attribute options.

Constant	Description
VtFontStyleBold	Applies the bold attribute to the font.
VtFontStyleItalic	Applies the italic attribute to the font.
VtFontStyleOutline	Applies the outline attribute to the font .

See Also

Style Property (MSChart Control)

FrameStyle Constants

The **VtFrameStyle** constants provide options for displaying backdrop frames.

Constant	Description
VtFrameStyleNull	No frame.
VtFrameStyleSingleLine	A single line encloses the backdrop.
VtFrameStyleDoubleLine	Two equal width lines enclose the backdrop.
VtFrameStyleThickInner	A thick inner line and a thin outer line enclose the backdrop.
VtFrameStyleThickOuter	A thin inner line and a thick outer line enclose the backdrop.

See Also

Frame Object | Frame Property | FrameColor Property | Style Property (MSChart Control)

MarkerStyle Constants

The **VtMarkerStyle** constants provide options for displaying data point markers.

Constant	Description
VtMarkerStyleNull	Suppressed
VtMarkerStyleDash	Dash marker
VtMarkerStylePlus	Plus marker
VtMarkerStyleX	X marker
VtMarkerStyleStar	Star marker
VtMarkerStyleCircle	Circle marker
VtMarkerStyleSquare	Square marker
VtMarkerStyleDiamond	Diamond marker
VtMarkerStyleUpTriangle	Triangle marker
VtMarkerStyleDownTriangle	Down triangle marker
VtMarkerStyleFilledCircle	Filled circle marker
VtMarkerStyleFilledSquare	Filled square marker
VtMarkerStyleFilledDiamond	Filled diamond marker
VtMarkerStyleFilledUpTriangle	Filled triangle marker
VtMarkerStyleFilledDownTriangle	Filled down triangle marker
VtMarkerStyle3dBall	Three-dimensional ball marker

See Also

Style Property (MSChart Control)

PenStyle Constants

The **VtPenStyle** constants provide options for the pen used to draw chart lines.

Constant	Description
VtPenStyleNull	No pen is applied
VtPenStyleSolid	Solid line pen
VtPenStyleDashed	Dashed line pen
VtPenStyleDotted	Dotted line pen
VtPenStyleDashDot	Dash-dot line pen
VtPenStyleDashDotDot	Dash-dot-dot line pen
VtPenStyleDitted	Ditted line pen
VtPenStyleDashDit	Dash-ditted line pen
VtPenStyleDashDitDit	Dash-dit-dit line pen

See Also

Style Property (MSChart Control) | Style Property (StatLine)

PieWeightStyle Constants

The **VtChPieWeightStyle** constants provide options for displaying individual pies within a single chart.

Constant	Description
VtChPieWeightStyleArea	The area of the individual pies changes, based on their weighting.
VtChPieWeightStyleDiameter	The diameter of the individual pies changes, based on their weighting.

See Also

Set Method (Weighting) | Style Property (MSChart Control)

ShadowStyle Constants

The **VtShadowStyle** constants provides shadow options.

Constant	Description
VtShadowStyleNull	No shadow.
VtShadowStyleDrop	Drop shadow.

See Also

Style Property (MSChart Control)

Style Property (StatLine)

Returns or sets the line type used to display the statistic line.

Syntax

object.**Style** (*type*)[= *style*]

The **Style** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list .
type	Integer. A VtChStats constant used to describe the line type.
style	Integer. A VtPenStyle constant used to describe the stat line style.

See Also

PenStyle Constants | Style Property (MSChart Control)

Applies To: StatLine Object

StatsType Constants

The **VtChStats** constants provide methods of displaying statistic lines on a chart.

Constant	Description
VtChStatsMinimum	Shows the minimum value in the series.
VtChStatsMaximum	Shows the maximum value in the series.
VtChStatsMean	Shows the mathematical mean of the values in the series .
VtChStatsStddev	Shows the standard deviation of the values in the series.
VtChStatsRegression	Shows a trend line indicated by the values in a series.

See Also

Flag Property | Style Property (StatLine)

SubPlotLabelPosition Property

Returns or sets the position used to display a label on each pie in a chart.

Syntax

object.**SubPlotLabelPosition** [= *pos*]

The **SubPlotLabelPosition** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
pos	Integer. A VtChSubPlotLabelLocationType constant used to describe the position of the chart label .

See Also

SubPlotLabelLocationType Constants

Applies To: Plot Object

SubPlotLabelLocationType Constants

The **VtSubPlotLabelLocationType** constants provide methods for displaying the subplot label.

Constant	Description
VtChSubPlotLabelLocationTypeNone	No subplot label is displayed.
VtChSubPlotLabelLocationTypeAbove	The subplot label is displayed above the pie .
VtChSubPlotLabelLocationTypeBelow	The subplot label is displayed below the pie .
VtChSubPlotLabelLocationTypeCenter	The subplot label is centered on the pie.

See Also

SubPlotLabelPosition Property

Text Property (MSChart)

Returns or sets the text used to display a chart element such as an axis title, data point label, footnote, or chart title.

Syntax

object.**Text** [= *text*]

The **Text** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list .
text	String. A string that contains the text used for the chart element.

Remarks

The **Text** property is the default property for each of the objects to which it applies.

See Also

Applies To: AxisTitle Object | DataPointLabel Object | Footnote Object | Title Object

TextOutputType Constants

The **VtTextOutputType** constants provide methods of outputting text.

Constant	Description
VtTextOutputTypeHardware	Device context type Null.
VtTextOutputTypePolygon	Device context type Metafile.

TextLayout Property

Returns a reference to a **TextLayout** object that describes text positioning and orientation.

Syntax

object.**TextLayout**

The object placeholder represents an object expression that evaluates to an object in the Applies To list.

See Also

Applies To: AxisTitle Object | DataPointLabel Object | Footnote Object | Label Object | Legend Object | Title Object

TextLength Property

Returns or sets the number of characters in the text of a chart axis title, data point label, footnote, or chart title.

Syntax

object.**TextLength** [= *size*]

The **TextLength** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list .
size	Integer. The number of characters in the text.

See Also

Applies To: AxisTitle Object | DataPointLabel Object | Footnote Object | Title Object

TextLengthType Property

Returns or sets a value that specifies how text is drawn to optimize the appearance either on the screen or printed page.

Syntax

object.**TextLengthType** [= *type*]

The **TextLengthType** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
type	Integer. A VtTextLengthType constant indicating the method used to draw text .

See Also

TextLengthType Constants

Applies To: MSChart Control

TextLengthType Constants

The **VtTextLengthType** constants provide options for optimizing text layout for the screen or for the printer.

Constant	Description
VtTextLengthTypeVirtual	Choose this constant to use TrueType virtual font metrics to optimize text layout for printing. TrueType virtual font metrics may not be very accurate for text displayed on the screen. Text displayed on the screen may be a larger or smaller than the virtual metrics requested. Larger text may not fit where it is supposed to and part of a character, a whole character, or even in some cases words may be clipped.
VtTextLengthTypeDevice	Choose this constant to optimize text layout for the screen. Text in charts laid out for screen display always fits correctly within its chart area. The printed text is generally a bit smaller and so the text may appear in slightly different places.

See Also

TextLengthType Property

Tick Property

Returns a reference to a **Tick** object that describes a marker indicating a division along a chart axis.

Syntax

object.**Tick**

The object placeholder represents an object expression that evaluates to an object in the Applies To list.

See Also

Applies To: Axis Object

Title Property (MSChart)

Reference to a **Title** object that describes the text used to title a chart.

Syntax

object.**Title**

The object placeholder represents an object expression that evaluates to an object in the Applies To list.

See Also

Applies To: MSChart Control

TitleText Property

Returns or sets the text displayed as the chart title.

Syntax

object.**TitleText** [= *text*]

The **TitleText** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list .
text	The text used to display a chart title.

Remarks

This property provides a simple means to set or return the chart title. This property is functionally identical to using `MSChart.Title.Text`.

See Also

Applies To: MSChart Control

Type Property (MSChart)

Returns or sets the scale type of an axis.

Syntax

object.**Type** [= *type*]

The **Type** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list .
type	A VtChScaleType constant describing the axis scale type.

See Also

ScaleType Constants

Applies To: AxisScale Object

ScaleType Constants

VtChScaleType provides methods for plotting chart values and displaying the chart scale.

Constant	Description
VtChScaleTypeLinear	Chart values are plotted in a linear scale with values ranging from the minimum to the maximum chart range value.
VtChScaleTypeLogarithmic	Chart values are plotted in a logarithmic scale with values based on a specific log scale set with the *logBase* argument of this function.
VtChScaleTypePercent	Chart values are plotted in a linear scale with values based on the percentages of the chart range values.

See Also

Type Property (MSChart)

UniformAxis Property

Returns or sets a value that specifies whether the unit scale for all value axes in a chart is uniform.

Syntax

object.**UniformAxis** [= *boolean*]

The **UniformAxis** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
boolean	A Boolean expression that controls the unit scale, as described in Settings .

Settings

The settings for *boolean* are:

Setting	Description
True	(Default) The unit scale for all value axes is uniform.
False	The unit scale is not uniform. The unit scale is determined by the plot size and positioning s et according to the AutoLayout or LocationRect property. If **AutoLayout** is **True**, the plot si ze and position are based on the size and position of other automatically laid out elements. If **False**, the coordinates specified by **LocationRect** are used to position the plot and deter mine the axes unit scale.

See Also

Applies To: Plot Object

ValueFormat Property

Returns or sets the format used to display the label as a value.

Syntax

object.**ValueFormat** [= *format*]

The **ValueFormat** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list .
format	String. Describes the format used to display a label as a value.

Remarks

Use the **DataPointLabel** object's **Component** property to change the label type.

See Also

Applies To: DataPointLabel Object

ValueScale Property

Returns a reference to a **ValueScale** object that describes the scale used to display a value axis.

Syntax

object.**ValueScale**

The object placeholder represents an object expression that evaluates to an object in the Applies To list.

See Also

CategoryScale Property

Applies To: Axis Object

VertAlignment Property

Returns or sets the method used to vertically align text.

Syntax

object.**VertAlignment** [= *type*]

The **VertAlignment** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
type	Integer. A VtVerticalAlignment constant used to describe the vertical alignment method of text .

See Also

VerticalAlignment Constants

Applies To: TextLayout Object

VerticalAlignment Constants

The **VtVerticalAlignment** constants provide methods of vertically aligning text.

Constant	Description
VtVerticalAlignmentTop	All lines of text are aligned at the top margin.
VtVerticalAlignmentBottom	All lines of text are aligned at the bottom margin .
VtVerticalAlignmentCenter	All lines of text are centered vertically.

See Also

VertAlignment Property

View3D Property

Returns a reference to a **View3D** object that describes the physical orientation of a three-dimensional chart.

Syntax

object.**View3D**

The object placeholder represents an object expression that evaluates to an object in the Applies To list.

See Also

Applies To: Plot Object

Visible Property (MSChart)

Returns or sets a value that determines whether a chart element is displayed.

Syntax

object.**Visible** [= *boolean*]

The **Visible** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
boolean	A Boolean expression that specifies whether the item is displayed, as described in Settings .

Settings

The settings for *boolean* are:

Setting	Description
True	The chart, axis title, label, or marker are displayed .
False	The elements are hidden.

See Also

Applies To: AxisTitle Object | Location Object | Marker Object | MSChart Control

VtColor Property

Returns a reference to a **VtColor** object that describes a drawing color in a chart.

Syntax

object.**VtColor**

The object placeholder represents an object expression that evaluates to an object in the Applies To list.

See Also

Applies To: Pen Object | StatLine Object | VtFont Object

VtFont Property

Returns a reference to a **VtFont** object that describes the font used to display chart text.

Syntax

object.**VtFont**

The object placeholder represents an object expression that evaluates to an object in the Applies To list.

See Also

Applies To: AxisTitle Object | DataPointLabel Object | Footnote Object | Label Object | Legend Object | Title Object

Wall Property

Returns a reference to a **Wall** object that describes the planar area depicting the y axes on a three-dimensional chart.

Syntax

object.**Wall**

The object placeholder represents an object expression that evaluates to an object in the Applies To list.

See Also

Applies To: Plot Object

Weighting Property

Returns a reference to a **Weighting** object that describes the size of a pie in relation to other pies in the same chart.

Syntax

object.**Weighting**

The object placeholder represents an object expression that evaluates to an object in the Applies To list.

See Also

Applies To: Plot Object

Width Property (MSChart)

Returns or sets the width of a chart element, in points.

Syntax

object.**Width** [= *width*]

The **Width** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list .
width	Single. The width of the chart element.

See Also

Applies To: Frame Object | MSChart Control | Pen Object | StatLine Object | Wall Object

WidthToHeightRatio Property

Returns or sets the percentage of the chart height to be used as the chart width.

Syntax

object.**WidthToHeightRatio** [= *pctg*]

The **WidthToHeightRatio** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list .
pctg	Single. The chart height percentage.

See Also

Applies To: Plot Object

WordWrap Property (MSChart)

Returns or sets a value that determines whether text wraps.

Syntax

object.**WordWrap** [= *boolean*]

The **WordWrap** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
boolean	A Boolean expression that determines whether text wraps, as described in Settings .

Settings

The settings for *boolean* are:

Setting	Description
True	Text wraps.
False	(Default) Text does not wrap .

See Also

Applies To: TextLayout Object

X Property

Returns or sets the *x* value in a floating coordinate pair for a chart.

Syntax

object.**X** [= *x*]

The **X** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
x	Single. (Long for **LCoor** object.) Identifies the *x* value of the coordinate .

See Also

Set Method (Coor, LCoor) | Y Property

Applies To: Coor Object | LCoor Object | LightSource Object

XGap Property

Returns or sets the spacing of bars between divisions on the *x* axis. This space is measured as a percentage of the bar width.

Syntax

object.**xGap** [= *spacing*]

The **xGap** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
spacing	Single. The bar width percentage. A value of 0 results in the series of bars touching .

See Also

Applies To: Plot Object

Y Property

Returns or sets the *y* value in a floating coordinate pair for a chart.

Syntax

object.**Y** [= *y*]

The **Y** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
y	Single. (Long for **LCoor** object.) Identifies the *y* value of the coordinate .

See Also

Set Method (Coor, LCoor) | X Property

Applies To: Coor Object | LCoor Object | LightSource Object

Z Property

Returns or sets the *z* value in a coordinate location.

Syntax

object.**Z** [= *z*]

The **Z** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
z	Single. (Long for **Lcoor** object.) Identifies the *z* value of the coordinate .

See Also

X Property | Y Property

Applies To: LightSource Object

ZGap Property

Returns or sets the spacing of three-dimensional bars between divisions on the *z* axis. This space is measured as a percentage of the bar depth.

Syntax

object.**zGap** [= *spacing*]

The **zGap** property syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
spacing	Single. The bar depth percentage. A value of 0 results in the series of bars touching along the *z* axis .

See Also

Applies To: Plot Object

MSChart Control Methods Listed Alphabetically

Add Method (LightSources Collection)

Count Method

DeleteColumnLabels Method

DeleteColumns Method

DeleteRowLabels Method

DeleteRows Method

EditCopy Method

EditPaste Method

GetData Method (MSChart)

GetSelectedPart Method

InitializeLabels Method

InsertColumnLabels Method

InsertColumns Method

InsertRowLabels Method

InsertRows Method

Layout Method

MoveData Method

RandomDataFill Method

RandomFillColumns Method

RandomFillRows Method

Remove Method (LightSources Collection)

ResetCustom Method

ResetCustomLabel Method

Select Method

SelectPart Method

Set Method (Coor, LCoor)

Set Method (LightSource)

Set Method (View3D)

Set Method (VtColor)

Set Method (Weighting)

SetData Method (MSChart)

SetSize Method

ToDefaults Method

TwipsToChartPart Method

TypeByChartType Method

Add Method (LightSources Collection)

Adds a **LightSource** object to the **LightSources** collection.

Syntax

*object.***Add** (*x,y,z,intensity)*

The **Add** method syntax has these parts:

Part	Description
collection	A object expression that evaluates to an object in the Applies To list.
x, y, z	Integers. Indicate the light source location.
intensity	Single. Indicates the light source intensity.

Remarks

Setting *x*, *y*, and *z* to zero generates a **VtChInvalidArgument** error.

See Also

Applies To: LightSources Collection

Count Method

Returns the number of objects in a collection.

Syntax

*object.***Count**

The object placeholder represents an object expression that evaluates to an object in the Applies To list.

See Also

Applies To: Series Collection | SeriesCollection Collection

DeleteColumnLabels Method

Deletes levels of labels from the data columns in a data grid associated with a chart.

Syntax

object.**DeleteColumnLabels** (*labelIndex*, *count*)

The **DeleteColumnLabels** method syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
labelIndex	Integer. Identifies the number of the first level of labels you want to delete. Column label levels are n umbered bottom to top, beginning with 1.
count	Integer. Specifies the number of label levels you want to delete. The number of columns being delete d is calculated from the column identified in *labelIndex* up.

See Also

DeleteColumns Method | DeleteRowLabels Method | DeleteRows Method

Applies To: DataGrid Object

DeleteColumns Method

Deletes columns of data and their associated labels from the data grid associated with a chart.

Syntax

*object.***DeleteColumns** (*column, count)*

The **DeleteColumns** method syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
column	Integer. Identifies a specific data column. Columns are numbered from left to right beginning with 1 .
count	Integer. Specifies the number of columns you want to delete.

See Also

DeleteColumnLabels Method | DeleteRowLabels Method | DeleteRows Method

Applies To: DataGrid Object

DeleteRowLabels Method

Deletes levels of labels from the data rows in a data grid associated with a chart.

Syntax

object.**DeleteRowLabels** (*labelIndex*, *count*)

The **DeleteRowLabels** method syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
labelIndex	Integer. Identifies the number of the first level of labels you want to delete. Row labels are numbered right to left, beginning with 1.
count	Integer. Specifies the number of label levels you want to delete. Row labels are deleted from the row identified by *labelIndex* to the left.

See Also

DeleteColumnLabels Method | DeleteColumns Method | DeleteRows Method

Applies To: DataGrid Object

DeleteRows Method

Deletes rows of data and their associated labels from the data grid associated with a chart.

Syntax

object.**DeleteRows** (*row*, *count*)

The **DeleteRows** method syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
row	Integer. Identifies a specific data row. Rows are numbered from top to bottom beginning with 1 .
count	Integer. Specifies the number of rows you want to delete.

See Also

DeleteColumnLabels Method | DeleteColumns Method | DeleteRowLabels Method

Applies To: DataGrid Object

EditCopy Method

Copies a picture of the current chart to the clipboard in Windows metafile format. It also copies the data being used to create the chart to the clipboard.

Syntax

object.**EditCopy**

The object placeholder represents an object expression that evaluates to an object in the Applies To list.

Remarks

This method allows you to paste the chart's data or a picture of the chart itself into another application. Since both the data and the picture of the chart are stored on the clipboard, what gets pasted into the new application varies depending on the type of application. For example, if you execute the chart's **EditCopy** method in your code and then go to an Excel spreadsheet and select **Edit Paste**, the chart data set is placed in the spreadsheet. To insert the picture of the chart into the spreadsheet, select **Edit Paste Special** and select the **Picture** type.

See Also

EditPaste Method

Applies To: MSChart Control

EditPaste Method

Pastes a Windows metafile graphic or tab-delimited text from the clipboard into the current selection on a chart.

Syntax

object.**EditPaste**

The object placeholder represents an object expression that evaluates to an object in the Applies To list.

Remarks

The chart can accept several types of information from the clipboard, depending on the currently selected chart element when **EditPaste** is called. If the entire chart is selected, the chart looks for data on the clipboard and attempts to use this new data to redraw the chart. If an item that can accept a picture, such as a bar or chart backdrop is selected, the chart looks for a metafile on the clipboard. If it finds a metafile, it uses that metafile to fill the selected object.

See Also

EditCopy Method

Applies To: MSChart Control

GetData Method (MSChart)

Returns the value currently stored in a specific data point in the data grid associated with a chart.

Syntax

object.**GetData** (*row*, *column*, *dataPoint*, *nullFlag*)

The **GetData** method syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list .
row	Integer. Identifies the row containing the data point value.
column	Integer. Identifies the column containing the data point value.
dataPoint	Double. The data point value.
nullFlag	Integer. Indicates whether or not the data point value is a null.

See Also

Applies To: DataGrid Object

GetSelectedPart Method

Identifies the currently selected chart element.

Syntax

object.**GetSelectedPart** (*part*, *index1*, *index2*, *index3*, *index4*)

The **GetSelectedPart** method syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
part	Integer. Specifies the chart element. Valid constants are VtChPartType.
index1	Integer. If element refers to a series or a data point, this argument specifies which series. Series are numbered in the order their corresponding columns appear in the data grid from left to right, beginning with 1. If element refers to an axis or axis label, this argument identifies the axis type with a VtChAxisId constant.
Index2	Integer. If element refers to a data point, this argument specifies which data point in the series identified by index1.
index3	Integer. If element refers to an axis label, this argument refers to the level of the label. Axis label levels are numbered from the axis out, beginning with 1. If element is not an axis label, the argument is unused.
index4	Integer. This argument is unused at this time.

See Also

AxisId Constants | PartType Constants

Applies To: MSChart Control

InitializeLabels Method

Assigns each label in the first level of data grid labels a unique identifier.

Syntax

*object.***InitializeLabels**

The object placeholder represents an object expression that evaluates to an object in the Applies To list.

See Also

Applies To: DataGrid Object

InsertColumnLabels Method

Inserts levels of labels for the data columns in a data grid associated with a chart.

Syntax

object.**InsertColumnLabels** (*labelIndex*, *count*)

The **InsertColumnLabels** method syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
labelIndex	Integer. Identifies the number of the first level of labels you want to insert. Column label levels are numbered bottom to top, beginning with 1.
count	Integer. Specifies the number of label levels you want to insert. The number of columns being inserted is calculated from the column identified in *labelIndex* up.

See Also

InsertColumns Method | InsertRowLabels Method | InsertRows Method

Applies To: DataGrid Object

InsertColumns Method

Adds one or more data columns to the data grid associated with a chart.

Syntax

object.**InsertColumns** (*column, count*)

The **InsertColumns** method syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
column	Integer. Identifies a specific data column. Columns are numbered from left to right beginning with 1 .
count	Integer. Specifies the number of columns you want to insert.

See Also

InsertColumnLabels Method | InsertRowLabels Method | InsertRows Method

Applies To: DataGrid Object

InsertRowLabels Method

Inserts levels of labels from the data rows in a data grid associated with a chart.

Syntax

object.**InsertRowLabels** (*labelIndex*, *count*)

The **InsertRowLabels** method syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
labelIndex	Integer. Identifies the number of the first level of labels you want to insert. Row labels are numbered right to left, beginning with 1.
count	Integer. Specifies the number of label levels you want to insert. Row labels are inserted from the row identified by *labelIndex* to the left.

See Also

InsertColumnLabels Method | InsertColumns Method | InsertRows Method

Applies To: DataGrid Object

InsertRows Method

Adds one or more data rows to the data grid associated with a chart.

Syntax

object.**InsertRows** (*row, count*)

The **InsertRows** method syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
row	Integer. Identifies a specific data row. Rows are numbered from top to bottom beginning with 1.
count	Integer. Specifies the number of rows you want to insert. Rows contain null data until you fill them with d ata.

See Also

InsertColumnLabels Method | InsertColumns Method | InsertRowLabels Method

Applies To: DataGrid Object

Layout Method

Lays out a chart, forcing recalculation of automatic values.

Syntax

object.**Layout**

The object placeholder represents an object expression that evaluates to an object in the Applies To list.

Remarks

A chart is laid out the first time it is drawn. When any chart settings change, the chart is again laid out at the next draw. There are a number of settings the chart calculates, such as the axis minimum and maximum values, based on the chart type or some other setting. These values are not determined until the chart is laid out. If you attempt to "get" these automatic values before the chart is properly laid out, they will not reflect the new values.

See Also

Applies To: MSChart Control

MoveData Method

Moves a range of data within a data grid associated with a chart.

Syntax

object.**MoveData** (*top*, *left*, *bottom*, *right*, *overOffset*, *downOffset*)

The **MoveData** method syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
top	Integer. Identifies the first row in the range to move.
left	Integer. Identifies the first column in the range to move.
bottom	Integer. Identifies the last row in the range to move.
right	Integer. Identifies the last column in the range to move.
overOffset	Integer. Identifies the horizontal direction data should be moved. A positive value moves data to the right; a negative value moves data to the left.
downOffset	Integer. Identifies the vertical direction data should be moved. A positive value moves data down, a negative value moves data up.

See Also

Applies To: DataGrid Object

RandomDataFill Method

Fills the data grid associated with a specific chart with randomly generated data.

Syntax

*object.***RandomDataFill**

The object placeholder represents an object expression that evaluates to an object in the Applies To list.

See Also

MoveData Method | RandomFill Property | RandomFillColumns Method | RandomFillRows Method

Applies To: DataGrid Object

RandomFillColumns Method

Fills a number of data grid columns associated with a chart with random values.

Syntax

object.**RandomFillColumns** (*column, count*)

The **RandomFillColumns** method syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
column	Integer. Identifies the first column you wish to fill. Columns are numbered from left to right beginning with 1.
count	Integer. Specifies the number of columns you want to fill with random data.

See Also

RandomDataFill Method | RandomFill Property | RandomFillRows Method

Applies To: DataGrid Object

RandomFillRows Method

Fills a number of data grid rows associated with a chart with random values.

Syntax

object.**RandomFillRows** (*row*, *count*)

The **RandomFillRows** method syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
row	Integer. Identifies the first row you wish to fill. Rows are numbered from top to bottom beginning with 1 .
count	Integer. Specifies the number of rows you want to fill with random data.

See Also

MoveData Method | RandomDataFill Method | RandomFill Property | RandomFillColumns Method

Applies To: DataGrid Object

Remove Method (LightSources Collection)

Removes a **LightSource** from the **LightSources** collection.

Syntax

object.**Remove** (*index*)

The **Remove** method syntax has these parts:

Part	Description
collection	A object expression that evaluates to an object in the Applies To list.
index	Integer. A specific light source by position in the list of light sources .

Applies To: LightSources Collection

ResetCustom Method

Resets any custom attributes placed on a data point to the series default.

Syntax

object.**ResetCustom**

The object placeholder represents an object expression that evaluates to an object in the Applies To list.

See Also

Applies To: DataPoint Object

ResetCustomLabel Method

Resets any custom attributes placed on a data point label in a chart to the series default.

Syntax

*object.***ResetCustomLabel**

The object placeholder represents an object expression that evaluates to an object in the Applies To list.

See Also

Applies To: DataPointLabel Object

Select Method

Selects the specified chart element.

Syntax

object.**Select**

The object placeholder represents an object expression that evaluates to an object in the Applies To list.

Applies To: DataPoint Object | DataPointLabel Object | Footnote Object | Legend Object | Series Object | Title Object

SelectPart Method

Selects the specified chart part.

Syntax

object.**SelectPart** (*part*, *index1*, *index2*, *index3*, *index4*)

The **SelectPart** method syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
part	Integer. Specifies the chart part. Valid constants are VtChPartType.
index1	Integer. If *part* refers to a series or a data point, this argument specifies which series. Series are numbered in the order their corresponding columns appear in the data grid from left to right, beginning with 1. If *part* refers to an axis or axis label, this argument identifies the axis type with a VtChAxisId constant.
index2	Integer. If *part* refers to a data point, this argument specifies which data point in the series is identified by index1. Data points are numbered in the order their corresponding rows appear in the data grid from top to bottom, beginning with 1. If *part* refers to an axis, axis title, or axis label, this argument refers to the axis index which is currently not used. In this case, the only valid value for this argument is 1.
index3	Integer. If *part* refers to an axis label, this argument refers to the level of the label. Axis label levels are numbered from the axis out, beginning with 1. If *part* is not an axis label, the argument is unused.
index4	Integer. This argument is unused at this time.

See Also

AxisId Constants | PartType Constants

Applies To: MSChart Control

PartType Constants

The **VtChPartType** constants provide options for chart elements.

Constant	Description
VtChPartTypeChart	Identifies the chart control.
VtChPartTypeTitle	Identifies the chart title.
VtChPartTypeFootnote	Identifies the chart footnote.
VtChPartTypeLegend	Identifies the chart legend.
VtChPartTypePlot	Identifies the chart plot.
VtChPartTypeSeries	Identifies a chart series.
VtChPartTypePoint	Identifies an individual data point .
VtChPartTypePointLabel	Identifies a data point label.
VtChPartTypeAxis	Identifies an axis.
VtChPartTypeAxisLabel	Identifies an axis label.
VtChPartTypeAxisTitle	Identifies an axis title.

See Also

GetSelectedPart Method | SelectPart Method | TwipsToChartPart Method

Set Method (Coor, LCoor)

Sets the *x* and *y* coordinate values for a chart.

Syntax

object.**Set** (*x*,*y*)

The **Set** method syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
x	Single. (Long for **LCoor** object.) Identifies the *x* value of the coordinate.
y	Single. (Long for **LCoor** object.) Identifies the *y* value of the coordinate .

See Also

X Property | Y Property

Applies To: Coor Object | LCoor Object

Set Method (LightSource)

Sets the *x*, *y*, and *z* coordinates and the intensity for the **LightSource** object location.

Syntax

object.**Set** (*x*,*y*,*z*, *intensity*)

The **Set** method syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list .
x, *y*, *z*	Integers. Indicate the light source location.
intensity	Single. Indicate the light source intensity.

See Also

Applies To: LightSource Object

Set Method (View3D)

Sets the rotation and degree of elevation for a three-dimensional chart.

Syntax

object.**Set** (*rotation*, *elevation*)

The **Set** method syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
rotation	Single. The degree of rotation. Rotation can range from 0 to 360 degrees. By default, degrees are used to measure rotation. However, thes e settings use the current settings for the **AngleUnit** property. The other options are: Grads and Radians.
elevation	Single. The degree of elevation. Elevation can be any number from 0 to 90 degrees. If you set the elevation to 90 degrees, you look directly down onto the top of the chart. If you set the elevation to 0, you look directly at the side of the chart. The de fault elevation is 30 degrees. By default, degrees are used to measure elevation. However, these settings us e the current settings for the **AngleUnits** property. The other options are: Grads and Radians.

See Also

Applies To: View3D Object

Set Method (VtColor)

Sets the red, green and blue values of the **VtColor** object.

Syntax

object.**Set** (*red*,*green*,*blue*)

The **Set** method syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
red, *green*, *blue*	Integer. The values for the red, green and blue components of color.

Remarks

RGB specifies the relative intensity of red, green, and blue to cause a specific color to be displayed. The valid range for a normal RGB color is 0 to 16,777,215. The value for any argument to RGB that exceeds 255 is assumed to be 255.

See Also

Applies To: VtColor Object

Set Method (Weighting)

Sets the basis and style of the **Weighting** object.

Syntax

object.**Set** (*basis, style*)

The **Set** method syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
basis	A VtChPieWeightBasis constant that identifies the weighting type.
style	A VtChPieWeightStyle constant that identifies the weighting factor method .

See Also

PieWeightBasis Constants | PieWeightStyle Constants

SetData Method (MSChart)

Sets the value for a specific data point in the data grid associated with a chart.

Syntax

object.**SetData** *(row, column, dataPoint, nullFlag)*

The **SetData** method syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list .
row	Integer. Identifies the row containing the data point value.
column	Integer. Identifies the column containing the data point value.
dataPoint	Double. The data point value.
nullFlag	Integer. Indicates whether or not the data point value is a null.

See Also

Applies To: MSChart Control

SetSize Method

Resizes the number of data columns and rows, as well as the number of levels of column labels and row labels of a data grid associated with a chart at one time.

Syntax

object.**SetSize** (*rowLabelCount*, *columnLabelCount*, *dataRowCount*, *columnLabelCount*)

The **SetSize** method syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
rowLabelCount	Integer. Returns or sets the number of levels of row labels you want on the data grid.
columnLabelCount	Integer. Returns or sets the number of levels of column labels you want on the data grid.
dataRowCount	Integer. Returns or sets the number of data rows you want on the data grid.
columnLabelCount	Integer. Returns or sets the number of data columns you want on the data grid.

Remarks

This method can be used in place of **RowCount**, **ColumnCount**, **RowLabelCount** and **ColumnLabelCount**.

If you reduce the size of the data grid, data in deleted rows or columns is destroyed.

See Also

Applies To: DataGrid Object

ToDefaults Method

Returns the chart to its initial settings.

Syntax

*object.***ToDefaults**

The object placeholder represents an object expression that evaluates to an object in the Applies To list.

See Also

Applies To: MSChart Control

TwipsToChartPart Method

Identifies a chart part by using the x and y set of coordinates on to identify that part.

Syntax

object.**TwipsToChartPart** (*xVal*, *yVal*, *part*, *index1*, *index2*, *index3*, *index4*)

The **TwipsToChartPart** method syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
xVal,*yVal*	Long. The horizontal and vertical coordinates of the point.
part	Integer. A VtChPartType constant that identifies the chart part that is located at the *xVal* and *yVal* coordinates.
index1	Integer. If *part* refers to a series or a data point, this argument specifies which series. Series are numbered in the order their corresponding columns appear in the data grid from left to right, beginning with 1. If part refers to an axis or axis label, this argument identifies the axis type using the VtChAxisId constant.
index2	Integer. If *part* refers to a data point, this argument specifies which data point in the series identified by index1. Data points are numbered in the order their corresponding rows appear in the data grid from top to bottom, beginning with 1. If part refers to an axis, axis title, or axis label, this argument refers to the axis index which is currently not used. In this case, the only valid value for this argument is 1.
index3	Integer. If *part* refers to an axis label, this argument refers to the level of the label. Axis label levels are numbered from the axis out, beginning with 1. If *part* is not an axis label, the argument is unused.
index4	Integer. This argument is unused at this time.

See Also

AxisId Constants | PartType Constants

Applies To: MSChart Control

TypeByChartType Method

Returns the series type used to draw a series if the chart type is set to *chType*. This method allows you to get the series type information based on a specified chart type without actually setting the chart type.

Syntax

object.**TypeByChartType** (*chtype*)

The **TypeByChartType** method syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list .
chtype	Integer. A VtChChartType constant describing the chart type.

Return Values

Depending on the chart type specified by the *chtype* argument, the **TypeByChartType** method returns one of the **VtChSeriesType** values, as shown below:

Constant	Value	Description
VtChSeriesTypeDefault	-1	Default
VtChSeriesType3dBar	0	3D Bar
VtChSeriesType2dBar	1	2D Bar
VtChSeriesType3dLine	5	3D Line
VtChSeriesType2dLine	6	2D Line
VtChSeriesType3dArea	7	3D Area
VtChSeriesType3dStep	9	3D Step
VtChSeriesType2dStep	10	2D Step
VtChSeriesType2dXY	11	XY
VtChSeriesType2dPie	24	2D Pie

See Also

ChartType Constants | SeriesType Constants

Applies To: Series Object

MSChart Control Events Listed Alphabetically

AxisActivated Event

AxisLabelActivated Event

AxisLabelSelected Event

AxisLabelUpdated Event

AxisSelected Event

AxisTitleActivated Event

AxisTitleSelected Event

AxisTitleUpdated Event

AxisUpdated Event

ChartActivated Event

ChartSelected Event

ChartUpdated Event

DataUpdated Event

DonePainting Event

FootnoteActivated Event

FootnoteSelected Event

FootnoteUpdated Event

LegendActivated Event

LegendSelected Event

LegendUpdated Event

PlotActivated Event

PlotSelected Event

PlotUpdated Event

PointActivated Event

PointLabelActivated Event

PointLabelSelected Event

PointLabelUpdated Event

PointSelected Event

PointUpdated Event

SeriesActivated Event

SeriesSelected Event

SeriesUpdated Event

TitleActivated Event

TitleSelected Event

TitleUpdated Event

AxisActivated Event

Occurs when the user double clicks on a chart axis.

Syntax

Private Sub *object*_**AxisActivated** (*axisId* **As Integer**, *axisIndex* **As Integer**, *mouseFlags* **As Integer**, *cancel* **As Integer**)

The AxisActivated event syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
axisId	Integer. An integer that identifies a specific axis, as described in Settings.
axisIndex	Integer. An integer reserved for future use. For this version of MSChart control, 1 is the only valid val ue for this argument.
mouseFlags	Integer. An integer that indicates whether a key is held down when the mouse button is clicked, as de scribed in Settings.
cancel	Integer. An integer that is not used at this time.

Settings

The event handler determines which axis is activated and sets *axisId* to:

Constants	Description
VtChAxisIdX	If the *x* axis is affected.
VtChAxisIdY	If the *y* axis is affected.
VtChAxisIdY2	If the secondary *y* axis is affected .
VtChAxisIdZ	If the *z* axis is affected.

The event handler determines if a key is held down when the mouse button is clicked and sets *mouseFlags* to:

Constants	Description
VtChMouseFlagShiftKeyDown	If the SHIFT key is held down.
VtChMouseFlagControlKeyDown	If the CONTROL key is held down.

See Also

Applies To: MSChart Control

AxisLabelActivated Event

Occurs when the user double clicks on an axis label.

Syntax

Private Sub *object*_**AxisLabelActivated** (*axisId* **As Integer**, *axisIndex* **As Integer**, *labelSetIndex* **As Integer**, *labelIndex* **As Integer**, *mouseFlags* **As Integer**, *cancel* **As Integer**)

The AxisLabelActivated event syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
axisId	Integer. An integer that identifies a specific axis, as described in Settings.
axisIndex	Integer. An integer reserved for future use. For this version of MSChart control, 1 is the only valid value for this argument.
labelSetIndex	Integer. An integer that identifies the level of labels you are double clicking on. Levels of labels ar e numbered from the axis out, beginning with 1.
labelIndex	Integer. An integer that is currently unused.
mouseFlags	Integer. An integer that indicates if a key is held down when the mouse button is clicked.
cancel	Integer. An integer that is not used at this time.

Settings

The event handler determines which axis label is activated and sets *axisId* to:

Constants	Description
VtChAxisIdX	If the *x* axis is affected.
VtChAxisIdY	If the *y* axis is affected.
VtChAxisIdY2	If the secondary *y* axis is affected.
VtChAxisIdZ	If the *z* axis is affected.

The event handler determines if a key is held down when the mouse button is clicked and sets *mouseFlags* to:

Constants	Description
VtChMouseFlagShiftKeyDown	If the SHIFT key is held down.
VtChMouseFlagControlKeyDown	If the CONTROL key is held down.

See Also

AxisLabelSelected Event | AxisLabelUpdated Event | UpdateFlags Constants

Applies To: MSChart Control

AxisLabelSelected Event

Occurs when the user clicks an axis label.

Syntax

Private Sub *object*_**AxisLabelSelected** (*axisId* **As Integer**, *axisIndex* **As Integer**, *labelSetIndex* **As Integer**, *labelIndex* **As Integer**, *mouseFlags* **As Integer**, *cancel* **As Integer**)

The AxisLabelSelected event syntax has these parts.

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
axisId	Integer. An integer that identifies a specific axis, as described in Settings.
axisIndex	Integer. An integer reserved for future use. For this version of MSChart control, 1 is the only valid v alue for this argument.
labelSetIndex	Integer. An integer that identifies the level of labels you are double clicking on. Levels of labels are numbered from the axis out, beginning with 1.
labelIndex	Integer. An integer that is currently unused.
mouseFlags	Integer. An integer that indicates if a key is held down when the mouse button is clicked, as describ ed in Settings.
cancel	Integer. An integer that is not used at this time.

Settings

The event handler determines which axis label is selected and sets *axisId* to:

Constants	Description
VtChAxisIdX	If the *x* axis is affected.
VtChAxisIdY	If the *y* axis is affected.
VtChAxisIdY2	If the secondary *y* axis is affected.
VtChAxisIdZ	If the *z* axis is affected.

The event handler determines if a key is held down when the mouse button is clicked and sets *mouseFlags* to:

Constants	Description
VtChMouseFlagShiftKeyDown	If the SHIFT key is held down.
VtChMouseFlagControlKeyDown	If the CONTROL key is held down.

See Also

AxisLabelActivated Event | AxisLabelUpdated Event

Applies To: MSChart Control

AxisLabelUpdated Event

Occurs when an axis label has changed.

Syntax

Private Sub *object*_**AxisLabelUpdated** (*axisId* **As Integer**, *axisIndex* **As Integer**, *labelSetIndex* **As Integer**, *labelIndex* **As Integer**, *updateFlags* **As Integer**)

The AxisLabelUpdated event syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
axisId	Integer. An integer that identifies a specific axis, as described in Settings.
axisIndex	Integer. An integer reserved for future use. For this version of MSChart control, 1 is the only valid value for this argument.
labelSetIndex	Integer. An integer that identifies the level of labels you are double clicking on. Levels of labels ar e numbered from the axis out, beginning with 1.
labelIndex	Integer. An integer that is currently unused.
updateFlags	Integer. An integer provides information about the update of the label, as described in Settings.

Settings

The event handler determines which axis label is updated and sets *axisId* to:

Constants	Description
VtChAxisIdX	If the *x* axis is affected.
VtChAxisIdY	If the *y* axis is affected.
VtChAxisIdY2	If the secondary *y* axis is affected .
VtChAxisIdZ	If the *z* axis is affected.

The event handler determines the affect of the update, and sets *updateFlags* to:

Constants	Description
VtChNoDisplay	Absence of update flags; the chart display is not affected. (Defined as 0.)
VtChDisplayPlot	Update will cause the plot to repaint.
VtChLayoutPlot	Update will cause the plot to lay out.
VtChDisplayLegend	Update will cause the legend to repaint.
VtChLayoutLegend	Update will cause the legend to lay out.
VtChLayoutSeries	Update will cause the series to lay out.
VtChPositionSection	A chart section has been moved or resized.

See Also

AxisLabelActivated Event | AxisLabelSelected Event

Applies To: MSChart Control

AxisSelected Event

Occurs when the user clicks on a chart axis.

Syntax

Private Sub *object*_**AxisSelected** (*axisId* **As Integer**, *axisIndex* **As Integer**, *mouseFlags* **As Integer**, *cancel* **As Integer**)

The AxisSelected event syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
axisId	Integer. An integer that identifies a specific axis, as described in Settings.
axisIndex	Integer. An integer reserved for future use. For this version of MSChart control, 1 is the only valid value for this argument.
mouseFlags	Integer. An integer that indicates whether key is held down when the mouse button is clicke d, as described in Settings.
cancel	Integer. An integer that is not used at this time.

Settings

The event handler determines which axis is selected and sets *axisId* to:

Constants	Value	Description
VtChAxisIdX	0	If the *x* axis is affected.
VtChAxisIdY	1	If the *y* axis is affected.
VtChAxisIdY2	2	If the secondary *y* axis is affected.
VtChAxisIdZ	3	If the *z* axis is affected.
VtChAxisIDNone	4	No axis.

The event handler determines if a key is held down when the mouse button is clicked and sets *mouseFlags* to:

Constants	Description
VtChMouseFlagShiftKeyDown	If the SHIFT key is held down.
VtChMouseFlagControlKeyDown	If the CONTROL key is held down.

See Also

Applies To: MSChart Control

AxisTitleActivated Event

Occurs when the user double clicks on an axis title.

Syntax

Private Sub *object*_**AxisTitleActivated** (*axisId* **As Integer**, *axisIndex* **As Integer**, *mouseFlags* **As Integer**, *cancel* **As Integer**)

The AxisTitleActivated event syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
axisId	Integer. An integer that identifies a specific axis.
AxisIndex	Integer. An integer reserved for future use. For this version of MSChart control, 1 is the only valid v alue for this argument.
mouseFlags	Integer. An integer that indicates whether a key is held down when the mouse button is clicked.
cancel	Integer. This argument is not used at this time.

Settings

The event handler determines which axis title is activated and sets *axisId* to:

Constants	Value	Description
VtChAxisIdX	0	If the *x* axis is affected.
VtChAxisIdY	1	If the *y* axis is affected.
VtChAxisIdY2	2	If the secondary *y* axis is affected.
VtChAxisIdZ	3	If the *z* axis is affected.
VtChAxisIDNone	4	No axis.

The event handler determines if a key is held down when the mouse button is clicked and sets *mouseFlags* to:

Constants	Description
VtChMouseFlagShiftKeyDown	If the SHIFT key is held down.
VtChMouseFlagControlKeyDown	If the CONTROL key is held down.

See Also

AxisTitleSelected Event | AxisTitleUpdated Event

Applies To: MSChart Control

AxisTitleSelected Event

Occurs when the user clicks on an axis title.

Syntax

Private Sub *object*_**AxisTitleSelected** (*axisId* **As Integer**, *axisIndex* **As Integer**, *mouseFlags* **As Integer**, *cancel* **As Integer**)

The AxisTitleSelected event syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
axisId	Integer. An integer that identifies a specific axis.
AxisIndex	Integer. An integer reserved for future use. For this version of MSChart control, 1 is the only valid v alue for this argument.
mouseFlags	Integer. An integer that indicates whether a key is held down when the mouse button is clicked.
cancel	Integer. This argument is not used at this time.

Settings

The event handler determines which axis title is selected and sets *axisId* to:

Constants	Value	Description
VtChAxisIdX	0	If the *x* axis is affected.
VtChAxisIdY	1	If the *y* axis is affected.
VtChAxisIdY2	2	If the secondary *y* axis is affected.
VtChAxisIdZ	3	If the *z* axis is affected.
VtChAxisIDNone	4	No axis.

The event handler determines if a key is held down when the mouse button is clicked and sets *mouseFlags* to:

Constants	Description
VtChMouseFlagShiftKeyDown	If the SHIFT key is held down.
VtChMouseFlagControlKeyDown	If the CONTROL key is held down.

See Also

AxisTitleActivated Event | AxisTitleUpdated Event

Applies To: MSChart Control

AxisTitleUpdated Event

Occurs when an axis title has changed.

Syntax

Private Sub *object*_**AxisTitleUpdated** (*axisId* **As Integer**, *axisIndex* **As Integer**, *updateFlags* **As Integer**)

The AxisTitleUpdated event syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
axisId	Integer. An integer that identifies a specific axis.
axisIndex	Integer. An integer reserved for future use. For this version of MSChart control, 1 is the only valid value for this argument.
updateFlags	Integer. An integer that provides information about the update of the title.

Settings

The event handler determines which axis title is updated and sets *axisId* to:

Constants	Value	Description
VtChAxisIdX	0	If the *x* axis is affected.
VtChAxisIdY	1	If the *y* axis is affected.
VtChAxisIdY2	2	If the secondary *y* axis is affected.
VtChAxisIdZ	3	If the *z* axis is affected.
VtChAxisIDNone	4	No axis.

The following table lists the constants for *updateFlags*.

Constants	Description
VtChNoDisplay	Absence of update flags; the chart display is not affected. (Defined as 0.)
VtChDisplayPlot	Update will cause the plot to repaint.
VtChLayoutPlot	Update will cause the plot to lay out.
VtChDisplayLegend	Update will cause the legend to repaint.
VtChLayoutLegend	Update will cause the legend to lay out.
VtChLayoutSeries	Update will cause the series to lay out.
VtChPositionSection	A chart section has been moved or resized.

See Also

AxisTitleActivated Event | AxisTitleSelected Event | UpdateFlags Constants

Applies To: MSChart Control

AxisUpdated Event

Occurs when an axis has changed.

Syntax

Private Sub *object*_**AxisUpdated** (*axisId* **As Integer**, *axisIndex* **As Integer**, *updateFlags* **As Integer**)

The AxisUpdated event syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
axisId	Integer. An integer that identifies a specific axis.
axisIndex	Integer. An integer reserved for future use. For this version of MSChart control, 1 is the only valid value for this argument.
updateFlags	Integer. An integer that provides information about the update of the axis.

Settings

The event handler determines which axis is updated and sets *axisId* to:

Constants	Value	Description
VtChAxisIdX	0	If the *x* axis is affected.
VtChAxisIdY	1	If the *y* axis is affected.
VtChAxisIdY2	2	If the secondary *y* axis is affected.
VtChAxisIdZ	3	If the *z* axis is affected.
VtChAxisIDNone	4	No axis.

The following table lists the constants for *updateFlags*.

Constants	Description
VtChNoDisplay	Absence of update flags; the chart display is not affected. (Defined as 0.)
VtChDisplayPlot	Update will cause the plot to repaint.
VtChLayoutPlot	Update will cause the plot to lay out.
VtChDisplayLegend	Update will cause the legend to repaint.
VtChLayoutLegend	Update will cause the legend to lay out.
VtChLayoutSeries	Update will cause the series to lay out.
VtChPositionSection	A chart section has been moved or resized.

See Also

UpdateFlags Constants

Applies To: MSChart Control

ChartActivated Event

Occurs when the user double clicks the Microsoft Chart control, but not on a specific element in the chart.

Syntax

Private Sub *object*_**ChartActivated** (*mouseFlags* **As Integer**, *cancel* **As Integer**)

The ChartActivated event syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
mouseFlags	Integer. Indicates whether a key is held down when the mouse button is clicked, as described in Settings.
cancel	Integer. This argument is not used at this time.

Settings

The event handler determines if a key is held down when the mouse button is clicked and sets *mouseFlags* to:

Constants	Description
VtChMouseFlagShiftKeyDown	If the SHIFT key is held down.
VtChMouseFlagControlKeyDown	If the CONTROL key is held down.

See Also

ChartSelected Event | ChartUpdated Event

Applies To: MSChart Control

ChartSelected Event

Occurs when the user clicks the Microsoft Chart control, but not on a specific element in the chart.

Syntax

Private Sub *object*_**ChartSelected** (*mouseFlags* **As Integer**, *cancel* **As Integer**)

The ChartSelected event syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
mouseFlags	Integer. Indicates whether a key is held down when the mouse button is clicked, as described in S ettings.
cancel	Integer. This argument is not used at this time.

Settings

The event handler determines if a key is held down when the mouse button is clicked and sets *mouseFlags* to:

Constants	Description
VtChMouseFlagShiftKeyDown	If the SHIFT key is held down.
VtChMouseFlagControlKeyDown	If the CONTROL key is held down.

See Also

ChartActivated Event | ChartUpdated Event

Applies To: MSChart Control

ChartUpdated Event

Occurs when the chart has changed.

Syntax

Private Sub *object*_**ChartUpdated** (*updateFlags* **As Integer**)

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
updateFlags	Integer. Provides information about the update of the chart, as described in Settings .

Settings

The following table lists the constants for *updateFlags*.

Constant	Description
VtChNoDisplay	Absence of update flags; the chart display is not affected. (Defined as 0.)
VtChDisplayPlot	Update will cause the plot to repaint.
VtChLayoutPlot	Update will cause the plot to lay out.
VtChDisplayLegend	Update will cause the legend to repaint.
VtChLayoutLegend	Update will cause the legend to lay out.
VtChLayoutSeries	Update will cause the series to lay out.
VtChPositionSection	A chart section has been moved or resized.

See Also

ChartActivated Event | ChartSelected Event | UpdateFlags Constants

Applies To: MSChart Control

DataUpdated Event

Occurs when the chart data grid has changed.

Syntax

Private Sub *object*_**DataUpdated** (*row* **As Integer**, *column* **As Integer**, *labelRow* **As Integer**, *labelColumn* **As Integer**, *labelSetIndex* **As Integer**, *updateFlags* **As Integer**)

The DataUpdated event syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
row	Integer. Indicates the row in the data grid.
column	Integer. Indicates the column in the datagrid.
labelRow	Integer. Indicates the row label.
labelColumn	Integer. Indicates the column label.
labelSetIndex	Integer. Identifies the level of labels. Levels of labels are numbered from the axis out, beginning with 1.
updateFlags	Integer. Provides information about the update of the data, as described in Settings.

Settings

The following table lists the constants for *updateFlags*.

Constant	Description
VtChNoDisplay	Absence of update flags; the chart display is not affected. (Defined as 0.)
VtChDisplayPlot	Update will cause the plot to repaint.
VtChLayoutPlot	Update will cause the plot to lay out.
VtChDisplayLegend	Update will cause the legend to repaint.
VtChLayoutLegend	Update will cause the legend to lay out.
VtChLayoutSeries	Update will cause the series to lay out.
VtChPositionSection	A chart section has been moved or resized.

Remarks

If row and column are nonzero, the change occurs to the indicated data cell. If *labelRow* or *labelColumn*, along with *labelSetIndex*, are nonzero, the indicated row or column label changes. If none of these are nonzero, no specific information about the change is available.

See Also

UpdateFlags Constants

Applies To: MSChart Control

DonePainting Event

Occurs immediately after the chart repaints or redraws.

Syntax

Private Sub *object*_**DonePainting** ()

The object placeholder represents an object expression that evaluates to an object in the Applies To list.

See Also

Applies To: MSChart Control

FootnoteActivated Event

Occurs when the user double clicks the chart footnote.

Syntax

Private Sub *object*_**FootnoteActivated**(*mouseFlags* **As Integer**, *cancel* **As Integer**)

The FootnoteActivated event syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
mouseFlags	Integer. Indicates whether a key is held down when the mouse button is clicked, as described in Settings.
cancel	This argument is not used at this time.

Settings

The event handler determines if a key is held down when the mouse button is clicked and sets *mouseFlags* to:

Constants	Description
VtChMouseFlagShiftKeyDown	If the SHIFT key is held down.
VtChMouseFlagControlKeyDown	If the CONTROL key is held down.

See Also

FootnoteSelected Event | FootnoteUpdated Event

Applies To: MSChart Control

FootnoteSelected Event

Occurs when the user clicks the chart footnote.

Syntax

Private Sub *object*_**FootnoteSelected** (*mouseFlags* **As Integer**, *cancel* **As Integer**)

The FootnoteSelected event syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
mouseFlags	Integer. Indicates whether a key is held down when the mouse button is clicked, as described in Settings.
cancel	This argument is not used at this time.

Settings

The event handler determines if a key is held down when the mouse button is clicked and sets *mouseFlags* to:

Constants	Description
VtChMouseFlagShiftKeyDown	If the SHIFT key is held down.
VtChMouseFlagControlKeyDown	If the CONTROL key is held down.

See Also

FootnoteActivated Event | FootnoteUpdated Event

Applies To: MSChart Control

FootnoteUpdated Event

Occurs when the chart footnote changes.

Syntax

Private Sub *object*_**FootnoteUpdated** (*updateFlags* **As Integer**)

The FootnoteUpdated event syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
updateFlags	Integer. Provides information about the update of the footnote, as described in Settings .

Settings

The following table lists the constants for *updateFlags*.

Constant	Description
VtChNoDisplay	Absence of update flags; the chart display is not affected. (Defined as 0.)
VtChDisplayPlot	Update will cause the plot to repaint.
VtChLayoutPlot	Update will cause the plot to lay out.
VtChDisplayLegend	Update will cause the legend to repaint.
VtChLayoutLegend	Update will cause the legend to lay out.
VtChLayoutSeries	Update will cause the series to lay out.
VtChPositionSection	A chart section has been moved or resized.

See Also

FootnoteActivated Event | FootnoteSelected Event | UpdateFlags Constants

Applies To: MSChart Control

LegendActivated Event

Occurs when the user double clicks on the chart legend.

Syntax

Private Sub *object*_**LegendActivated** (*mouseFlags* **As Integer**, *cancel* **As Integer**)

The LegendActivated event syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
mouseFlags	Integer. Indicates whether a key is held down when the mouse button is clicked, as described in S ettings.
cancel	Integer. This argument is not used at this time.

Settings

The event handler determines if a key is held down when the mouse button is clicked and sets *mouseFlags* to:

Constants	Description
VtChMouseFlagShiftKeyDown	If the SHIFT key is held down.
VtChMouseFlagControlKeyDown	If the CONTROL key is held down .

See Also

Applies To: MSChart Control

LegendSelected Event

Occurs when the user clicks on the chart legend.

Syntax

Private Sub *object*_**LegendSelected** (*mouseFlags* **As Integer**, *cancel* **As Integer**)

The LegendSelected event syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
mouseFlags	Integer. Indicates whether a key is held down when the mouse button is clicked .
cancel	Integer. This argument is not used at this time.

Settings

The event handler determines if a key is held down when the mouse button is clicked and sets *mouseFlags* to:

Constants	Description
VtChMouseFlagShiftKeyDown	If the SHIFT key is held down.
VtChMouseFlagControlKeyDown	If the CONTROL key is held down .

See Also

Applies To: MSChart Control

LegendUpdated Event

Occurs when the chart legend has changed.

Syntax

Private Sub *object*_**LegendUpdated** (*updateFlags* **As Integer**)

The LegendUpdated event syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
updateFlags	Integer. Provides information about the update of the legend, as described in Settings .

Settings

The following table lists the constants for *updateFlags*.

Constant	Description
VtChNoDisplay	Absence of update flags; the chart display is not affected. (Defined as 0.)
VtChDisplayPlot	Update will cause the plot to repaint.
VtChLayoutPlot	Update will cause the plot to lay out.
VtChDisplayLegend	Update will cause the legend to repaint.
VtChLayoutLegend	Update will cause the legend to lay out.
VtChLayoutSeries	Update will cause the series to lay out.
VtChPositionSection	A chart section has been moved or resized.

See Also

UpdateFlags Constants

Applies To: MSChart Control

PlotActivated Event

Occurs when the user double clicks the chart plot.

Syntax

Private Sub *object*_**PlotActivated** (*mouseFlags* **As Integer**, *cancel* **As Integer**)

The PlotActivated event syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
mouseFlags	Integer. Indicates whether a key is held down when the mouse button is clicked, as described in Settings.
cancel	Integer. This argument is not used at this time.

Settings

The event handler determines if a key is held down when the mouse button is clicked and sets *mouseFlags* to:

Constants	Description
VtChMouseFlagShiftKeyDown	If the SHIFT key is held down.
VtChMouseFlagControlKeyDown	If the CONTROL key is held down.

See Also

Applies To: MSChart Control

PlotSelected Event

Occurs when the user clicks the chart plot.

Syntax

Private Sub *object*_**PlotSelected** (*mouseFlags* **As Integer**, *cancel* **As Integer**)

The PlotSelected event syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
mouseFlags	Integer. Indicates whether a key is held down when the mouse button is clicked, as described in S ettings.
cancel	Integer. This argument is not used at this time.

Settings

The event handler determines if a key is held down when the mouse button is clicked and sets *mouseFlags* to:

Constants	Description
VtChMouseFlagShiftKeyDown	If the SHIFT key is held down.
VtChMouseFlagControlKeyDown	If the CONTROL key is held down.

See Also

Applies To: MSChart Control

PlotUpdated Event

Occurs when the chart plot has changed.

Syntax

Private Sub *object*_**PlotUpdated** (*updateFlags* **As Integer**)

The PlotUpdated event syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
updateFlags	Integer. Provides information about the update of the plot, as described in Settings.

Settings

The following table lists the constants for *updateFlags*.

Constant	Description
VtChNoDisplay	Absence of update flags; the chart display is not affected. (Defined as 0.)
VtChDisplayPlot	Update will cause the plot to repaint.
VtChLayoutPlot	Update will cause the plot to lay out.
VtChDisplayLegend	Update will cause the legend to repaint.
VtChLayoutLegend	Update will cause the legend to lay out.
VtChLayoutSeries	Update will cause the series to lay out.
VtChPositionSection	A chart section has been moved or resized.

See Also

UpdateFlags Constants

Applies To: MSChart Control

PointActivated Event

Occurs when the user double clicks on a data point.

Syntax

Private Sub *object*_**PointActivated** (*series* **As Integer**, *dataPoint* **As Integer**, *mouseFlags* **As Integer**, *cancel* **As Integer**)

The PointActivated event syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
series	Integer. Identifies the series containing the data point. Series are numbered in the order that their columns appear in the data grid, beginning with 1.
dataPoint	Integer. Identifies the data point's position in the series. Points are numbered in the order that their rows appear in the data grid, beginning with 1.
mouseFlags	Integer. Indicates whether a key is held down when the mouse button is clicked, as described in Settings.
cancel	Integer. This argument is not used at this time.

Settings

The event handler determines if a key is held down when the mouse button is clicked and sets *mouseFlags* to:

Constants	Description
VtChMouseFlagShiftKeyDown	If the SHIFT key is held down.
VtChMouseFlagControlKeyDown	If the CONTROL key is held down.

See Also

Applies To: MSChart Control

PointLabelActivated Event

Occurs when the user double clicks a data point label.

Syntax

Private Sub *object*_**PointLabelActivated** (*series* **As Integer**, *dataPoint* **As Integer**, *mouseFlags* **As Integer**, *cancel* **As Integer**)

The PointLabelActivated event syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
series	Integer. Identifies the series containing the data point. Series are numbered in the order that their columns appear in the data grid, beginning with 1.
dataPoint	Integer. Identifies the data point's position in the series. Points are numbered in the order that their rows appear in the data grid, beginning with 1.
mouseFlags	Integer. Indicates whether a key is held down when the mouse button is clicked, as described in Settings.
cancel	Integer. This argument is not used at this time.

Settings

The event handler determines if a key is held down when the mouse button is clicked and sets *mouseFlags* to:

Constants	Description
VtChMouseFlagShiftKeyDown	If the SHIFT key is held down.
VtChMouseFlagControlKeyDown	If the CONTROL key is held down.

See Also

Applies To: MSChart Control

PointLabelSelected Event

Occurs when the user clicks a data point label.

Syntax

Private Sub *object*_**PointLabelSelected** (*series* **As Integer**, *dataPoint* **As Integer**, *mouseFlags* **As Integer**, *cancel* **As Integer**)

The PointLabelSelected event syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
series	Integer. Identifies the series containing the data point. Series are numbered in the order that their column s appear in the data grid, beginning with 1.
dataPoint	Integer. Identifies the data point's position in the series. Points are numbered in the order that their rows appear in the data grid, beginning with 1.
mouseFlags	Integer. Indicates whether a key is held down when the mouse button is clicked, as described in Settings.
cancel	This argument is not used at this time.

Settings

The event handler determines if a key is held down when the mouse button is clicked and sets *mouseFlags* to:

Constants	Description
VtChMouseFlagShiftKeyDown	If the SHIFT key is held down.
VtChMouseFlagControlKeyDown	If the CONTROL key is held down.

See Also

Applies To: MSChart Control

PointLabelUpdated Event

Occurs when a data point label has changed.

Syntax

Private Sub *object*_**PointLabelUpdated** (*series* **As Integer**, *dataPoint* **As Integer**, *updateFlags* **As Integer**)

The PointLabelUpdated event syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
series	Integer. Identifies the series containing the data point. Series are numbered in the order that their columns appear in the data grid, beginning with 1.
dataPoint	Integer. Identifies the data point's position in the series. Points are numbered in the order that their rows appear in the data grid, beginning with 1.
updateFlags	Integer. Provides information about the update of the data point label, as described in Settings.

Settings

The following table lists the constants for *updateFlags*.

Constant	Description
VtChNoDisplay	Absence of update flags; the chart display is not affected. (Defined as 0.)
VtChDisplayPlot	Update will cause the plot to repaint.
VtChLayoutPlot	Update will cause the plot to lay out.
VtChDisplayLegend	Update will cause the legend to repaint.
VtChLayoutLegend	Update will cause the legend to lay out.
VtChLayoutSeries	Update will cause the series to lay out.
VtChPositionSection	A chart section has been moved or resized.

See Also

UpdateFlags Constants

Applies To: MSChart Control

PointSelected Event

Occurs when the user clicks a data point.

Syntax

Private Sub *object*_**PointSelected** (*series* **As Integer**, *dataPoint* **As Integer**, *mouseFlags* **As Integer**, *cancel* **As Integer**)

The PointSelected event syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
series	Integer. Identifies the series containing the data point. Series are numbered in the order that their columns appear in the data grid, beginning with 1.
dataPoint	Integer. Identifies the data point's position in the series. Points are numbered in the order that their rows appear in the data grid, beginning with 1.
mouseFlags	Integer. Indicates whether a key is held down when the mouse button is clicked, as described in Settings.
cancel	Integer. This argument is not used at this time.

Settings

The event handler determines if a key is held down when the mouse button is clicked and sets *mouseFlags* to:

Constants	Description
VtChMouseFlagShiftKeyDown	If the SHIFT key is held down.
VtChMouseFlagControlKeyDown	If the CONTROL key is held down.

See Also

Applies To: MSChart Control

PointUpdated Event

Occurs when a data point has changed.

Syntax

Private Sub *object*_**PointUpdated** (*series* **As Integer**, *dataPoint* **As Integer**, *updateFlags* **As Integer**)

The PointUpdated event syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
series	Integer. Identifies the series containing the data point. Series are numbered in the order that their columns appear in the data grid, beginning with 1.
dataPoint	Integer. Identifies the data point's position in the series. Points are numbered in the order that their rows appear in the data grid, beginning with 1.
updateFlags	Integer. Provides information about the update of the data point, as described in Settings.

Settings

The following table lists the constants for *updateFlags*.

Constant	Description
VtChNoDisplay	Absence of update flags; the chart display is not affected. (Defined as 0.)
VtChDisplayPlot	Update will cause the plot to repaint.
VtChLayoutPlot	Update will cause the plot to lay out.
VtChDisplayLegend	Update will cause the legend to repaint.
VtChLayoutLegend	Update will cause the legend to lay out.
VtChLayoutSeries	Update will cause the series to lay out.
VtChPositionSection	A chart section has been moved or resized.

See Also

UpdateFlags Constants

Applies To: MSChart Control

SeriesActivated Event

Occurs when the user double clicks a chart series. You can replace the standard user interface by canceling the event and displaying your own dialog box.

Syntax

Private Sub *object*_**SeriesActivated** (*series* **As Integer**, *mouseFlags* **As Integer**, *cancel* **As Integer**)

The SeriesActivated event syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
series	Integer. Identifies the series containing the data point. Series are numbered in the order that their columns appear in the data grid, beginning with 1.
mouseFlags	Integer. Indicates whether a key is held down when the mouse button is clicked, as described in Settings.
cancel	Integer. This argument is not used at this time.

Settings

The event handler determines if a key is held down when the mouse button is clicked and sets *mouseFlags* to:

Constants	Description
VtChMouseFlagShiftKeyDown	If the SHIFT key is held down.
VtChMouseFlagControlKeyDown	If the CONTROL key is held down.

See Also

Applies To: MSChart Control

MouseFlag Constants

VtChMouseFlag indicates which keyboard key is being held down while the mouse button is clicked.

Constant	Description
VtChMouseFlagShiftKeyDown	The SHIFT key is held down when the mouse button is clicked.
VtChMouseFlagControlKeyDown	The CONTROL key is held down when the mouse button is clicked.

SeriesSelected Event

Occurs when the user clicks a chart series.

Syntax

Private Sub *object*_**SeriesSelected** (*series* **As Integer**, *mouseFlags* **As Integer**, *cancel* **As Integer**)

The SeriesSelected event syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
series	Integer. Identifies the series containing the data point. Series are numbered in the order that thei r columns appear in the data grid, beginning with 1.
mouseFlags	Integer. Indicates whether a key is held down when the mouse button is clicked, as described in S ettings.
cancel	Integer. This argument is not used at this time.

Settings

The event handler determines if a key is held down when the mouse button is clicked and sets *mouseFlags* to:

Constants	Description
VtChMouseFlagShiftKeyDown	If the SHIFT key is held down.
VtChMouseFlagControlKeyDown	If the CONTROL key is held down.

See Also

Applies To: MSChart Control

SeriesUpdated Event

Occurs when a chart series has changed.

Syntax

Private Sub *object*_**SeriesUpdated** (*series* **As Integer**, *updateFlags* **As Integer**)

The SeriesUpdated event syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
series	Integer. Identifies the series containing the data point. Series are numbered in the order that thei r columns appear in the data grid, beginning with 1.
updateFlags	Integer. Provides information about the update of the series, as described in Settings.

Settings

The following table lists the constants for *updateFlags*.

Constant	Description
VtChNoDisplay	Absence of update flags; the chart display is not affected. (Defined as 0.)
VtChDisplayPlot	Update will cause the plot to repaint.
VtChLayoutPlot	Update will cause the plot to lay out.
VtChDisplayLegend	Update will cause the legend to repaint.
VtChLayoutLegend	Update will cause the legend to lay out.
VtChLayoutSeries	Update will cause the series to lay out.
VtChPositionSection	A chart section has been moved or resized.

See Also

UpdateFlags Constants

Applies To: MSChart Control

UpdateFlags Constants

The **UpdateFlags** constants listed below can be used in various events to determine what part of the chart is being updated.

Constant	Value	Description
VtChNoDisplay	0	Absence of update flags; the chart display is not affected .
VtChDisplayPlot	1	Update will cause the plot to repaint.
VtChLayoutPlot	2	Update will cause the plot to layout.
VtChDisplayLegend	4	Update will cause the legend to repaint.
VtChLayoutLegend	8	Update will cause the legend to layout.
VtChLayoutSeries	16	Update will cause the series to lay out.
VtChPositionSection	32	A chart section has been moved or resized.

Remarks

The constants can be used in the following events:

- AxisLabelUpdated Event
- AxisTitleUpdated Event
- AxisUpdated
- ChartUpdated Event
- DataUpdated Event
- FootnoteUpdated Event
- LegendUpdated Event
- PlotUpdated Event
- PointLabelUpdated Event
- PointUpdated Event
- SeriesUpdated Event
- TitleUpdated Event

See Also

AxisLabelActivated Event | AxisTitleUpdated Event | AxisUpdated Event | ChartUpdated Event | DataUpdated Event | FootnoteUpdated Event | LegendUpdated Event | PlotUpdated Event | PointLabelUpdated Event | PointUpdated Event | SeriesUpdated Event | TitleUpdated Event

TitleActivated Event

Occurs when the user double clicks the chart title. You can replace the standard user interface by canceling the event and displaying your own dialog box.

Syntax

Private Sub *object*_**TitleActivated** (*mouseFlags* **As Integer**, *cancel* **As Integer**)

The TitleActivated event syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
mouseFlags	Integer. Indicates whether a key is held down when the mouse button is clicked, as described in Setti ngs.
cancel	Integer. This argument is not used at this time.

Settings

The event handler determines if a key is held down when the mouse button is clicked and sets *mouseFlags* to:

Constants	Description
VtChMouseFlagShiftKeyDown	If the SHIFT key is held down.
VtChMouseFlagControlKeyDown	If the CONTROL key is held down.

See Also

Applies To: MSChart Control

TitleSelected Event

Occurs when the user clicks the chart title.

Syntax

Private Sub *object*_**TitleSelected** (*mouseFlags* **As Integer**, *cancel* **As Integer**)

The TitleSelected event syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
mouseFlags	Integer. Indicates whether a key is held down when the mouse button is clicked, as described in Settings.
cancel	Integer. This argument is not used at this time.

Settings

The event handler determines if a key is held down when the mouse button is clicked and sets *mouseFlags* to:

Constants	Description
VtChMouseFlagShiftKeyDown	If the SHIFT key is held down.
VtChMouseFlagControlKeyDown	If the CONTROL key is held down.

See Also

Applies To: MSChart Control

TitleUpdated Event

Occurs when the chart title has changed.

Syntax

Private Sub *object*_**TitleUpdated** (*updateFlags* **As Integer**)

The TitleUpdated event syntax has these parts:

Part	Description
object	An object expression that evaluates to an object in the Applies To list.
updateFlags	Provides information about the update of the title, as described in Settings.

Settings

The following table lists the constants for *updateFlags*.

Constant	Description
VtChNoDisplay	Absence of update flags; the chart display is not affected. (Defined as 0.)
VtChDisplayPlot	Update will cause the plot to repaint.
VtChLayoutPlot	Update will cause the plot to lay out.
VtChDisplayLegend	Update will cause the legend to repaint.
VtChLayoutLegend	Update will cause the legend to lay out.
VtChLayoutSeries	Update will cause the series to lay out.
VtChPositionSection	A chart section has been moved or resized.

See Also

UpdateFlags Constants

Applies To: MSChart Control

Samples

This topic links you to sample application and code example topics for Visual Basic and Visual C# programmers.

In This Section

Visual Basic Code Example Topics
Shows quickstart-like code examples that help you perform common tasks in Visual Basic .NET.
Visual C# Code Example Topics
Shows quickstart-like code examples that help you perform common tasks in Visual C# .NET.
Visual Basic Sample Applications
Describes how you can access sample code, and provides links to descriptions of sample Visual Basic programs.
Visual C# Samples
Describes how you can access sample code, and provides links to descriptions of sample Visual C# programs.

Related Sections

Walkthroughs in Visual Basic and Visual C#
Provides links to topics that walk you through complete scenarios covering different types of applications.
Programming with Office
Provides information on using Microsoft Office and Visual Studio .NET as part of your business application.

Samples	Visual Basic Samples
Samples	
Samples	Visual Basic Concepts
Samples	
Samples	
Samples	Writing C# Applications
Samples	
Samples	
Visual C# Samples	C# Sample Applications
Visual C# Samples	

Code: Redirecting to Another Page (Visual Basic)

This example uses the HttpResponse.Redirect method to redirect the browser to the specified URL.

Example

```
Response.Redirect("http://www.microsoft.com")
```

Compiling the Code

This example requires:

- A Web Forms page.

Robust Programming

The **Redirect** method is executed by the browser and therefore results in an extra round trip between server and browser.

It is up to you to make sure that the target page exists.

Security

Because the method is executed by the client, it is subject to spoofing or other tampering.

See Also

Code: Redirecting to Another Page in the Same Application (Visual Basic) | HttpResponse Class | HttpResponse.Redirect Method | Web Forms Pages | Web Applications Example Topics

Code: Overriding Control Validation on Windows Forms in the Closing Event (Visual Basic)

This example shows how to close a form even if there is validation logic in the form and one or more of the controls contain invalid data. In the example, there is validation logic for a TextBox control. Validation is overridden in the form's **Closing** event.

Example

```
Private Sub TextBox1_Validating(ByVal sender As Object, _
    ByVal e As System.ComponentModel.CancelEventArgs) _
    Handles TextBox1.Validating
    If TextBox1.Text.Length < 8 Then
            ("Value you entered is not valid;" _
            & " it must be at least 8 characters.")
        e.Cancel = True
    End If
End Sub

Private Sub Form1_Closing(ByVal sender As Object, _
    ByVal e As System.ComponentModel.CancelEventArgs) _
    Handles MyBase.Closing
    e.Cancel = False
End Sub
```

Compiling the Code

This project requires:

- A reference to the **System** namespace.
- A Windows Form.
- A **TextBox** control named TextBox1.
- Another control on the form that can receive focus to illustrate validation being overridden.

Security

User input into a Windows Forms can include potentially invalid, malformed, or malicious information. Always check that the values that a user enters conform to the requirements for your application before using the values.

See Also

Code: Validating User Input on Windows Forms (Visual Basic) | Validation of Control Data on Windows Forms | Form.Closing Event | CancelEventArgs Class | Windows Applications Example Topics

Code: Databinding a DataTable of String Values to a Windows Forms DataGrid Control (Visual Basic)

This example creates a data table with two columns and two rows and then binds it to an existing **DataGrid** control.

Example

```
Public Sub BindTableToDataGrid()
    Dim dt As New DataTable()
    dt.Columns.Add(New DataColumn("Item", GetType(String)))
    dt.Columns.Add(New DataColumn("Color", GetType(String)))
    dt.Rows.Add(New String() {"table", "brown"})
    dt.Rows.Add(New String() {"chair", "white"})
    DataGrid1.DataSource = dt
End Sub
```

Compiling the Code

This example requires:

- A reference to the **System** namespace.
- A **DataGrid** control named DataGrid1.

See Also

Code: Creating a DataTable (Visual Basic) | DataGrid Control (Windows Forms) | DataGrid.DataSource Property | DataTable Class | Windows Forms Controls Example Topics

Code: Hiding a DataColumn in a Windows Forms DataGrid Control Where the DataSource is a DataTable (Visual Basic)

This example hides the "X" column of a **DataTable** object that is displayed in an existing Windows Forms **DataGrid** control.

Example

```
Sub HideColumnOfDataSet()
    Dim points As New System.Data.DataTable("Points")
    points.Columns.Add(New DataColumn("X", GetType(Integer)))
    points.Columns.Add(New DataColumn("Y", GetType(Integer)))
    points.Rows.Add(New Object() {1, 2})
    points.Rows.Add(New Object() {3, 5})
    DataGrid1.DataSource = points

    Dim tableStyle As New DataGridTableStyle
    tableStyle.MappingName = "Points"
    DataGrid1.TableStyles.Add(tableStyle)
    DataGrid1.TableStyles("Points").GridColumnStyles("X").Width = 0
End Sub
```

Compiling the Code

This example requires:

- A Windows Form.
- A **DataGrid** control named DataGrid1.

If the data source is a **DataSet** object, set the **DataMember** property of the **DataGrid** to the table's name.

DataTable and **DataColumn** objects in typed datasets also have string-typed names. To find the name of the table, examine the table's **TableName** property. To find the name of the **DataColumn**, examine the column's **ColumnName** property.

Another method for hiding data columns is to create the **DataGridColumnStyle** objects and add them to the **GridColumnStyles** collection. With this method, you can hide a column by not adding a column style for it. For an example, see Formatting the Windows Forms DataGrid Control.

Robust Programming

The following conditions may cause an exception:

- The **MappingName** property does not match the name of the **DataTable** (NullReferenceException Class).
- Retrieving an item from the **GridColumnStyles** collection before the **TableStyle** is added to the **DataGrid.TableStyles** collection. The **GridColumnStyles** collection is filled when the **TableStyle** object is added to the **TableStyles** collection (NullReferenceException Class).
- A **TableStyle** added to the **DataGrid**.**TableStyles** collection does not have a unique **MappingName** (ArgumentException Class).
- The **DataSource** property of the **DataGrid** is not set before accessing the **GridColumnStyles** collection (NullReferenceException Class).

See Also

Windows Forms Controls Example Topics | Hiding a DataColumn in a DataGrid Where the DataSource is an Array | Hiding a DataColumn in a DataGrid Where the DataSource is an ArrayList | DataGrid Control (Windows Forms) | Formatting the Windows Forms DataGrid Control | DataGrid.TableStyles Property | DataGridTableStyle.MappingName Property | DataGridTableStyle.GridColumnStyles Property

Code: Hiding a DataColumn in a Windows Forms DataGrid Control Where the DataSource is an Array (Visual Basic)

This example hides the "X" column of a **DataGrid** that uses an array of **Point** objects as the data source.

Example

```
Sub HideColumnOfArray()
    Dim points() As System.Drawing.Point = _
        {New Point(1, 2), New Point(3, 5), New Point(5, 6)}
    DataGrid1.DataSource = points

    Dim tableStyle As New DataGridTableStyle
    Dim mappingName As String = DataGrid1.DataSource.GetType().Name
    tableStyle.MappingName = mappingName
    DataGrid1.TableStyles.Add(tableStyle)
    DataGrid1.TableStyles(mappingName).GridColumnStyles("X").Width = 0
End Sub
```

Compiling the Code

This example requires:

- A Windows Form.
- A **DataGrid** control named DataGrid1.

Another method for hiding data columns is to create the **DataGridColumnStyle** objects and add them to the **GridColumnStyles** collection. With this method, you can hide a column by not adding a column style for it. For an example, see Formatting the Windows Forms DataGrid Control.

Robust Programming

The following conditions may cause an exception:

- The **MappingName** property is not property formatted (NullReferenceException Class).
- Retrieving an item from the **GridColumnStyles** collection before the **TableStyle** is added to the **DataGrid.TableStyles** collection. The **GridColumnStyles** collection is filled when the **TableStyle** object is added to the **TableStyles** collection (NullReferenceException Class).
- A **TableStyle** added to the **DataGrid**.**TableStyles** collection does not have a unique **MappingName** (ArgumentException Class).
- The **DataSource** property of the **DataGrid** is not set before accessing the **GridColumnStyles** collection (NullReferenceException Class).

See Also

Windows Forms Controls Example Topics | Hiding a DataColumn in a DataGrid Where the DataSource is a DataTable | Hiding a DataColumn in a DataGrid Where the DataSource is an ArrayList | DataGrid Control (Windows Forms) | Formatting the Windows Forms DataGrid Control | DataGrid.TableStyles Property | DataGridTableStyle.MappingName Property | DataGridTableStyle.GridColumnStyles Property

Code: Hiding a DataColumn in a Windows Forms DataGrid Control Where the DataSource is an ArrayList (Visual Basic)

This example hides one column of a **DataGrid** that uses an **ArrayList** of **Point** objects as a data source.

Example

```
Sub HideColumnOfArrayList()
    Dim list As New System.Collections.ArrayList
    list.AddRange(New System.Drawing.Point() _
        {New Point(1, 2), New Point(3, 5), New Point(5, 6)})
    DataGrid1.DataSource = list

    Dim tableStyle As New DataGridTableStyle
    tableStyle.MappingName = "ArrayList"
    DataGrid1.TableStyles.Add(tableStyle)
    DataGrid1.TableStyles("ArrayList").GridColumnStyles("X").Width = 0
End Sub
```

Compiling the Code

This example requires:

- A Windows Form.
- A **DataGrid** control named DataGrid1.

When the data source is an **ArrayList** object, the **MappingName** is "ArrayList".

Another method for hiding data columns is to create the **DataGridColumnStyle** objects and add them to the **GridColumnStyles** collection. With this method, you can hide a column by not adding a column style for it. For an example, see Formatting the Windows Forms DataGrid Control.

Robust Programming

The following conditions may cause an exception:

- The **MappingName** property is not "ArrayList" (NullReferenceException Class).
- Retrieving an item from the **GridColumnStyles** collection before the **TableStyle** is added to the **DataGrid.TableStyles** collection. The **GridColumnStyles** collection is filled when the **TableStyle** object is added to the **TableStyles** collection (NullReferenceException Class).
- A **TableStyle** added to the **DataGrid**.**TableStyles** collection does not have a unique **MappingName** (ArgumentException Class).
- All the objects in the **ArrayList** are not of the same type of the first item or a derived type of the first item (TargetInvocationException Class).
- The **DataSource** property of the **DataGrid** is not set before accessing the **GridColumnStyles** collection (NullReferenceException Class).

See Also

Windows Forms Controls Example Topics | Hiding a DataColumn in a DataGrid Where the DataSource is a DataTable | Hiding a DataColumn in a DataGrid Where the DataSource is an Array | DataGrid Control (Windows Forms) | Formatting the Windows Forms DataGrid Control | DataGrid.TableStyles Property | DataGridTableStyle.MappingName Property | DataGridTableStyle.GridColumnStyles Property

Code: Determining Whether a Registry Key Exists (Visual Basic)

This example checks to see if the specified key, SOFTWARE\\Microsoft\\TestApp\\1.0, exists and creates it if it does not.

Example

```
Private Sub DoesKeyExist
    Dim regVersion As Microsoft.Win32.RegistryKey
    regVersion = _
Microsoft.Win32.Registry.CurrentUser.OpenSubKey("SOFTWARE\\Microsoft\\TestApp\\1.0", True)
    If regVersion Is Nothing Then
        regVersion = _
Microsoft.Win32.Registry.CurrentUser.CreateSubKey("SOFTWARE\\Microsoft\\TestApp\\1.0")
    End If
End Sub
```

Compiling the Code

This example requires:

- A reference to the **Win32** namespace.

Security

Registry calls fail if either sufficient runtime permissions are not granted (RegistryPermission Class) or if the user does not have the correct ACLs for creating or writing to settings. For example, a local application that has the code access security permission might not have operating system permission.

See Also

Code: Creating a Key In the Registry (Visual Basic) | Registry Access Changes in Visual Basic .NET | Registry Class | Registry Summary | System Input/Output Example Topics

Code: Displaying Variables in a String (Visual Basic)

This example declares the variable number and then displays it and its square in the string result.

Example

```
Private Sub DisplayTheSquare()
    Dim number As Integer = 4
    Dim result As String = _
        String.Format("{0} times {0} = {1}", _
        number, number * number)
    MsgBox(result)
End Sub
```

Compiling the Code

This example requires:

- A reference to the **System** namespace.

Robust programming

The following conditions may create an exception:

- Number is too large (OverflowException Class)
- Format string is incorrect, such as using {2} instead of {1} when there is no third element. (FormatException Class)

See Also

String Manipulation | Strings | String.Format Method | System Input/Output Example Topics

Visual C# Code Example Topics

Code example topics in the Visual C# .NET documentation are designed to present concise, real-world examples of common tasks. These topics are organized by feature area.

Code Topic Categories

Code example topics are available for the following features:

General Language Example Topics
Presents a series of specific example code topics for common tasks using language elements of Visual C# .NET.
Graphics Programming Example Topics
Presents a series of specific example code topics for common tasks in graphics and GDI+ programming with Visual C# .NET.
Web Applications Example Topics
Presents a series of specific example code topics for common tasks in Web Forms applications with Visual C# .NET.
Windows Forms Example Topics
Presents a series of specific example code topics for common tasks in Windows Forms applications with Visual C# .NET.

Other Sources For Sample Code

There are also many other places to find C# code examples for common features and tasks in the Visual Studio and .NET Framework help system. You can also find sample code in the following locations:

C# Programmer's Reference
The topics in this section contain code showing how to use basic language features such as threading, inheritance, and file IO.
Visual C# Samples
Lists compiled sample applications available to you in the documentation set.
Walkthroughs in Visual Basic and Visual C#
Lists tutorial topics that walk you through the process of creating applications and learning to use specific features.
Quickstarts and Other Samples
Lists samples available to you when you install the .NET Framework SDK.
.NET Framework Reference Information
Many topics in the .NET Framework reference contain code examples of how to use classes and their members.

General Language Example Topics

This topic is designed to help you find code that demonstrates how to perform common programming tasks in applications using Visual C#.NET.

In This Section

- Breaking Out of an Iterative Statement
- Calling a Method on an Object
- Catching an Exception
- Declaring an Array
- Declaring a Property
- Determining the Span Between Two Dates
- Displaying a Web Page in Internet Explorer
- Finding the Names of Resources in an Assembly
- Generating Multiline String Literals
- Inheriting From a Class
- Initializing an Array
- Iterating Through an Array
- Iterating Through a Collection
- Passing Object Arrays to Methods
- Reading a Text File One Line at a Time
- Reading Class Data from an XML File
- Reading From a Text File
- Reading XML From a File
- Running a Program
- Running a Program Associated with a File Type
- Searching for a String in an Array of Strings
- Searching Within a String
- Setting a Property on an Object
- Simulating Default Parameters
- Throwing an Exception
- Using PInvoke
- Writing Class Data to an XML File
- Writing to a Text File

Related Code Examples

You can also find sample code for C# applications in the following locations:

- C# Language Tour
- Visual C# Samples
- Walkthroughs in Visual Basic and Visual C#
- Quickstarts and other Samples
- .NET Framework Reference Information

Code: Breaking Out of an Iterative Statement (Visual C#)

This example uses the **break** statement to break out of a **for** loop when an integer counter reaches 10.

Example

```
for (int counter = 1; counter <= 1000; counter++)
{
    if (counter == 10)
       break;
    Console.WriteLine(counter);
}
```

Compiling the Code

Copy the code and paste it into the **Main** method of a console application.

See Also

break | Jump Statements | General Language Example Topics

Code: Calling a Method on an Object (Visual C#)

This example calls the **WriteLine** method on the **Console** class.

Example

```
Console.WriteLine("This line will appear in the console window");
```

Compiling the Code

Copy the code and paste it into the **Main** method of a console application.

Robust Programming

Use the fully qualified name of the method, unless it is accessible from the same scope.

Unless the method is static, it must be referenced through an instance of the class.

When assigning an expression to a property, make sure of the following:

- That it is of a compatible data type.
- That it has a valid value, especially if the value is derived from user input.

See Also

Console Class | Console.WriteLine Method (String) | General Language Example Topics

Code: Catching an Exception (Visual C#)

This example uses the **try**-**catch** blocks to catch division by zero as an exception.

Example

```
int top = 0, bottom = 0, result = 0;
try
{
    result = top / bottom;
}
catch (System.DivideByZeroException ex)
{
    Console.WriteLine("{0} Exception caught.", ex);
}
```

Compiling the Code

Copy the code and paste it into the **Main** method of a console application.

See Also

Code: Throwing an Exception (Visual C#) | try-catch | try-catch-finally | DivideByZeroException Class | General Language Example Topics

Code: Declaring a Property (Visual C#)

This example declares an instance property. For more information and examples on instance and static properties, see Properties.

Example

```
private string name;
// A read-write instance property:
public string NameProperty
{
   get
   {
      return name;
   }
   set
   {
      name = value;
   }
}
```

Compiling the Code

The code must appear within a class or a struct.

Robust Programming

- You can use the **get** accessor to either return the field value or compute the value and return it, for example:

```
get
{
    return (name != null) ? name : "NA";
}
```

- Do not use the **get** accessor to change the state of the object, for example:

```
get
{
    return myNumericField++;
}
```

See Also

Property Declaration | Accessors | Interface Properties | General Language Example Topics

Code: Declaring an Array (Visual C#)

This example shows three different ways to declare different kinds of arrays: single-dimensional, multidimensional, and jagged.

Example

```
// Single-dimensional arrays.
int[] myArray1 = new int [5];
string[] myArray2 = new string[6];

// Multidimensional arrays.
int[,] myArray3 = new int[4,2];
int[,,] myArray4 = new int [4,2,3];

// Jagged array.
int[][] myArray5 = new int[3][];
```

Compiling the Code

Copy the code and paste it into the **Main** method of a console application.

Robust Programming

Make sure you initialize the elements of the jagged array before using it, as follows:

```
myArray5[0] = new int[7];
myArray5[1] = new int[5];
myArray5[2] = new int[2];
```

See Also

Arrays | Arrays Tutorial | Array Class | Reference Types | General Language Example Topics

Code: Determining the Span Between Two Dates (Visual C#)

This example calculates the difference in days between two dates and constructs a **TimeSpan** value for that difference.

Example

```
DateTime oldDate = new DateTime(2002,7,15);
DateTime newDate = DateTime.Now;

// Difference in days, hours, and minutes.
TimeSpan ts = newDate - oldDate;
// Difference in days.
int differenceInDays = ts.Days;

Console.WriteLine("Difference in days: {0} ", differenceInDays);
```

Compiling the Code

Copy the code and paste it into the **Main** method of a console application.

See Also

TimeSpan Structure | DateTime Members | DateTime Subtraction Operator | DateTime.Subtract Method| TimeSpan.Days Property | General Language Example Topics

Code: Displaying a Web Page in Internet Explorer (Visual C#)

This example displays www.microsoft.com in Internet Explorer using the **Process.Start** method.

Example

```
System.Diagnostics.Process.Start(
"C:\\Program Files\\Internet Explorer\\IExplore.exe",
"www.microsoft.com");
```

Compiling the Code

Copy the code and paste it into the **Main** method of a console application.

Replace the first parameter of **Start** with the path to IExplore.exe, if the program is installed in a different location.

See Also

Process.Start Method | Starting Processes | General Language Example Topics

Code: Finding the Names of Resources in an Assembly (Visual C#)

This example creates a string that contains the names of all the resources in an assembly.

Example

```
System.Reflection.Assembly thisExe;
thisExe = System.Reflection.Assembly.GetExecutingAssembly();
string [] resources = thisExe.GetManifestResourceNames();
string list = "";

// Build the string of resources.
foreach (string resource in resources)
   list += resource + "\r\n";
```

Compiling the Code

Copy the code and paste it into the **Main** method of a console application.

See Also

Assembly Class | General Language Example Topics

Code: Generating Multiline String Literals (Visual C#)

This example demonstrates two ways of constructing a multiline string literal.

Example

```
string myString1 = "This is the first line of my string.\n" +
                   "This is the second line of my string.\n" +
                   "This is the third line of the string.\n";
string myString2 = @"This is the first line of my string.
This is the second line of my string.
This is the third line of the string.";
```

Compiling the Code

Copy the code and paste it into the **Main** method of a console application.

See Also

string | General Language Example Topics.

Code: Inheriting From a Class (Visual C#)

This example defines the **Circle** and **Rectangle** classes, which both inherit from the **Shape** class, and the **Square** class, which inherits from the **Rectangle** class.

Example

```
public class Shape
{
    // Definitions of properties, methods, fields, and events.
}
public class Circle : Shape
{
    // Specialized properties, methods, fields, events for Circle.
}
public class Rectangle : Shape
{
    // Specialized properties, methods, fields, events for Rectangle.
}
public class Square : Rectangle
{
    // Specialized properties, methods, fields, events for Square.
}
```

Compiling the Code

- Start a new console application.
- Copy the code and paste it right before or after **Class1** declaration.

Robust Programming

Make sure the class you want to inherit is not sealed.

See Also

class | sealed | General Language Example Topics

Code: Initializing an Array (Visual C#)

This example shows three different ways to initialize different kinds of arrays: single-dimensional, multidimensional, and jagged.

Example

```
// Single-dimensional array (numbers).
int[] n1 = new int[4] {2, 4, 6, 8};
int[] n2 = new int[] {2, 4, 6, 8};
int[] n3 = {2, 4, 6, 8};
// Single-dimensional array (strings).
string[] s1 = new string[3] {"John", "Paul", "Mary"};
string[] s2 = new string[] {"John", "Paul", "Mary"};
string[] s3 = {"John", "Paul", "Mary"};

// Multidimensional array.
int[,] n4 = new int[3, 2] { {1, 2}, {3, 4}, {5, 6} };
int[,] n5 = new int[,] { {1, 2}, {3, 4}, {5, 6} };
int[,] n6 = { {1, 2}, {3, 4}, {5, 6} };

// Jagged array.
int[][] n7 = new int[2][] { new int[] {2,4,6}, new int[] {1,3,5,7,9} };
int[][] n8 = new int[][] { new int[] {2,4,6}, new int[] {1,3,5,7,9} };
int[][] n9 = { new int[] {2,4,6}, new int[] {1,3,5,7,9} };
```

Compiling the Code

Copy the code and paste it into the **Main** method of a console application.

Robust Programming

Array members are automatically initialized to the default initial value for the array type if the array is not initialized at the time it is declared. If the array declaration is a field of a type, then when the type is instantiated, the array will be set to its default value of **null**. For more information on default values of C# types, see Default Values Table.

See Also

Arrays | Arrays Tutorial | Array Class | Reference Types | General Language Example Topics

Code: Iterating Through a Collection (Visual C#)

This example uses the **Add** method on the **Hashtable** class to add entries to a **Hashtable** collection. It then uses a **foreach** statement to iterate through the collection.

Example

```
Hashtable phones = new Hashtable();
// Add items.
phones.Add("John", "123-4567");
phones.Add("Enju", "351-8765");
phones.Add("Molly", "221-5678");
phones.Add("James", "010-4077");
phones.Add("Nelly", "110-5699");
phones.Add("Leah", "922-5699");

// Iterate through the collection.
Console.WriteLine("Name\t\tNumber");
foreach (string name in phones.Keys)
{
    Console.WriteLine(name +"\t"+ phones[name]);
}
```

Compiling the Code

- Copy the code and paste it into the **Main** method of a console application.
- Add a using directive to the **System.Collections** namespace.

See Also

Iteration Statements | foreach, in | System.Collections Namespace | Hashtable Class | Using foreach with Collections| General Language Example Topics

Code: Iterating Through an Array (Visual C#)

This example uses the **foreach** statement to access and display items of an array.

Example

```
int[] numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9};
foreach (int element in numbers)
{
    System.Console.WriteLine(element);
}
```

Compiling the Code

Copy the code and paste it into the **Main** method of a console application.

See Also

Iteration Statements | foreach, in | Using foreach with Arrays | Using foreach with Collections | Arrays | Arrays Tutorial | General Language Example Topics

Code: Passing Object Arrays to Methods (Visual C#)

This example shows how an array of objects is passed to the **DisplayMyCollection** method, which uses the **params** keyword to accept any number of arguments.

Example

```
class MyBoxingClass
{
   public static void DisplayMyCollection(params object[] anArray)
   {
      foreach (object obj in anArray)
      {
         System.Console.Write(obj + "\t");
      }

      // Suspend the screen.
      System.Console.ReadLine();
   }

   static void Main()
   {
      DisplayMyCollection(101, "Visual C# Basics", 2002);
   }
}
```

Compiling the Code

You can compile the example directly using the command line, or paste it into a console application using the Visual Studio IDE.

See Also

params Keyword | General Language Example Topics

Code: Reading a Text File One Line at a Time (Visual C#)

This example reads the contents of a text file, one line at a time, into a string using the **ReadLine** method of the **StreamReader** class. Each text line is stored into the string `line` and displayed on the screen.

Example

```
int counter = 0;
string line;

// Read the file and display it line by line.
System.IO.StreamReader file =
   new System.IO.StreamReader("c:\\test.txt");
while((line = file.ReadLine()) != null)
{
   Console.WriteLine (line);
   counter++;
}

file.Close();

// Suspend the screen.
Console.ReadLine();
```

Compiling the Code

Copy the code and paste it into the **Main** method of a console application.

Replace `"c:\\test.txt"` with the actual file name.

Robust Programming

The following conditions may cause an exception:

- The file may not exist (IOException).

Security

Do not make decisions about the contents of the file based on the name of the file. For example, the file myFile.cs may not be a C# source file.

See Also

Code: Writing to a Text File (Visual C#) | Code: Reading From a Text File (Visual C#) |
File Access Through StreamReader and StreamWriter Classes | StreamReader Class | General Language Example Topics

Code: Reading Class Data from an XML File (Visual C#)

This example reads the data stored in an object in the IntroToVCS.xml file using the **Deserialize** method of the **XmlSerializer** class.

Example

```
public class Book
{
   public string title;

   static void Main()
   {
      Book introToVCS  = new Book();
      System.Xml.Serialization.XmlSerializer reader = new
         System.Xml.Serialization.XmlSerializer(introToVCS.GetType());

      // Read the XML file.
      System.IO.StreamReader file=
         new System.IO.StreamReader("c:\\IntroToVCS.xml");

      // Deserialize the content of the file into a Book object.
      introToVCS = (Book) reader.Deserialize(file);
      System.Windows.Forms.MessageBox.Show(introToVCS.title,
         "Book Title");
   }
}
```

Compiling the Code

You can compile the example directly using the command line, or paste the code into a console application using the Visual Studio IDE. In the latter case, you must reference the System.Windows.Forms.dll file.

Replace `"c:\\IntroToVCS.xml"` with the actual file name.

Robust Programming

The following conditions may cause an exception:

- The path name may be too long (PathTooLongException).

See Also

Code: Writing Class Data to an XML File (Visual C#) | File Access Through StreamReader and StreamWriter Classes | StreamWriter Class | General Language Example Topics

Code: Reading From a Text File (Visual C#)

This example reads the contents of a text file into a string using the **ReadToEnd** method of the **StreamReader** class.

Example

```
// Read the file as one string.
System.IO.StreamReader myFile =
   new System.IO.StreamReader("c:\\test.txt");
string myString = myFile.ReadToEnd();

myFile.Close();

// Display the file contents.
Console.WriteLine(myString);
// Suspend the screen.
Console.ReadLine();
```

Compiling the Code

Copy the code and paste it into the **Main** method of a console application.

Replace `"c:\\test.txt"` with the actual file name.

Robust Programming

The following conditions may cause an exception:

- The file may not exist (IOException).

Security

Do not make decisions about the contents of the file based on the name of the file. For example, the file myFile.cs may not be a C# source file.

See Also

Code: Writing to a Text File (Visual C#) | Code: Reading a Text File One Line at a Time (Visual C#) | File Access Through StreamReader and StreamWriter Classes | StreamReader Class | | General Language Example Topics

Code: Reading XML From a File (Visual C#)

This example uses the **XMLTextReader** class to extract the element names and text strings from the IntroToVCS.xml file and store the information in a string variable.

Example

```
System.Xml.XmlTextReader reader = new System.Xml.XmlTextReader("c:\\IntroToVCS.xml");
         string contents = "";
         while (reader.Read())
         {
            reader.MoveToContent();
            if (reader.NodeType == System.Xml.XmlNodeType.Element)
               contents += "<"+reader.Name + ">\n";
            if (reader.NodeType == System.Xml.XmlNodeType.Text)
               contents += reader.Value + "\n";
         }
Console.Write(contents);
```

Compiling the Code

Copy the code and paste it into the **Main** method of a console application.

Replace `"c:\\IntroToVCS.xml"` with the actual file name.

> **Note** The XML document must be well-formed.

Robust Programming

The following conditions may cause an exception:

- The path name may be too long (PathTooLongException).

See Also

Code: Writing Class Data to an XML File (Visual C#) | Reading XML with the XmlReader | General Language Example Topics

Code: Running a Program (Visual C#)

This example loads and runs the MSPaint application using the **Process.Start** method.

Example

```
System.Diagnostics.Process.Start("mspaint.exe");
```

Compiling the Code

Copy the code and paste it into the **Main** method of a console application.

Replace `"mspaint.exe"` with the path to the application you want to run.

Robust Programming

The following conditions may cause an exception:

- The file may not exist (IOException).

Security

The call to **Process.Start** requires full trust (SecurityException Class).

See Also

Process.Start Method | Starting Processes | General Language Example Topics

Code: Running a Program Associated With a File Type (Visual C#)

This example opens the test.txt file in Notepad using the **Process.Start** method.

Example

```
System.Diagnostics.Process.Start("c:\\test.txt");
```

Compiling the Code

Copy the code and paste it into the **Main** method of a console application.

Replace the file "`c:\\test.txt`" with the file you want to open.

Robust Programming

The following conditions may cause an exception:

- The file may not exist (IOException).

Security

The call to **Process.Start** requires full trust (SecurityException Class).

See Also

Process.Start Method | Starting Processes | General Language Example Topics

Code: Searching for a String in an Array of Strings (Visual C#)

This example calls the **IndexOf** method on an array of strings to report the string number and index of the first occurrence of a substring.

Example

```
string[] strArray = {"ABCDEFG", "HIJKLMNOP"};
string findThisString = "JKL";
int strNumber;
int strIndex = 0;
for (strNumber = 0; strNumber < strArray.Length; strNumber++)
{
    strIndex = strArray[strNumber].IndexOf(findThisString);
    if (strIndex >= 0)
        break;
}
Console.WriteLine("String number: {0}\nString index: {1}",
    strNumber, strIndex);
```

Compiling the Code

Copy the code and paste it into the **Main** method of a console application.

Robust Programming

The **IndexOf** method reports the location of the first character of the first occurrence of the substring. The index is 0-based, which means the first character of a string has an index of 0.

If **IndexOf** does not find the substring, it returns -1.

The **IndexOf** method is case-sensitive and uses the current culture.

If you want more control over possible exceptions, enclose the string search in a try-catch statement.

See Also

Code: Searching Within a String (Visual C#) | Exception Handling Statements | string | String.IndexOf Method (String) | General Language Example Topics

Code: Searching Within a String (Visual C#)

This example calls the **IndexOf** method on a **String** object to report the index of the first occurrence of a substring.

Example

```
string searchWithinThis = "ABCDEFGHIJKLMNOP";
string searchForThis = "DEF";
int firstCharacter = searchWithinThis.IndexOf(searchForThis);

Console.WriteLine("First occurrence: {0}", firstCharacter);
```

Compiling the Code

Copy the code and paste it into the **Main** method of a console application.

Robust Programming

The **IndexOf** method reports the location of the first character of the first occurrence of the substring. The index is 0-based, which means the first character of a string has an index of 0.

If **IndexOf** does not find the substring, it returns -1.

The **IndexOf** method is case-sensitive and uses the current culture.

If you want more control over possible exceptions, enclose the string search in a try-catch statement.

See Also

Exception Handling Statements| string | String.IndexOf Method (String) | General Language Example Topics

Code: Setting a Property on an Object (Visual C#)

This example sets the **CurrentDirectory** property on the **Environment** class to C:\Public.

Example

```
Environment.CurrentDirectory = "C:\\Public";
```

-or-

```
Environment.CurrentDirectory = @"C:\Public";
```

Compiling the Code

Copy the code and paste it into the **Main** method of a console application.

Robust Programming

Use the fully qualified name of the property, unless it is accessible from the same scope.

Unless the property is static, it must be referenced through an instance of the class.

When assigning an expression to a property, make sure of the following:

- That it is of a compatible data type.
- That it has a valid value, especially if the value is derived from user input.

If you want more control over possible exceptions, enclose the property assignment in a try-catch statement.

See Also

Property Declaration | Exception Handling Statements | Environment Class | Environment.CurrentDirectory Property | General Language Example Topics

Code: Simulating Default Parameters (Visual C#)

This example demonstrates the use of method overloading to simulate default parameters, which is not allowed in C#.

Example

```
class MyClass
{
    static string myMethod(string precip, string country, string location)
    {
        return string.Format("The {0} in {1} stays mainly in the {2}.",
            precip, country, location );
    }

    static string myMethod(string precip, string country )
    {
        return myMethod(precip, country, "plain");
    }

    static string myMethod()
    {
        return myMethod("rain", "Spain", "plain");
    }

    static void Main(string[] args)
    {
        Console.WriteLine(myMethod());
        Console.WriteLine(myMethod("snow", "Walla Walla"));
    }
}
```

Compiling the Code

Copy the class and paste it over `Class1` in a console application.

See Also

Comparison Between C++ and C# | Compiler Error CS0241 | General Language Example Topics

Code: Throwing an Exception (Visual C#)

This example throws the exception **System.ApplicationException**.

Example

```
throw new ApplicationException();
```

Compiling the Code

Copy the code and paste it into the **Main** method of a console application.

Robust Programming

Unhandled exceptions cause application execution to stop.

See Also

Code: Catching an Exception (Visual C#) | throw | try-catch | ApplicationException Class | Exception Class | General Language Example Topics

Code: Using PInvoke (Visual C#)

This example shows you how to use the **DllImport** attribute to create a message box by calling **MessageBoxA** from user32.dll.

Example

```
class PlatformInvokeTest
{
   [DllImport("user32.dll")]
   public static extern int MessageBoxA(
      int h, string m, string c, int type);

   public static int Main()
   {
      return MessageBoxA(0, "Hello World!", "My Message Box", 0);
   }
}
```

Compiling the Code

- Copy the class and paste it over `Class1` in a console application.
- Add a using directive to the **System.Runtime.InteropServices** namespace.

Robust Programming

- Make sure to include the correct name of the DLL that contains the required method (user32.dll in this case).
- Make sure to use the appropriate parameters of the **MessageBoxA** function.

See Also

Platform Invoke Tutorial | Visual C# Code Example Topics

Code: Writing Class Data to an XML File (Visual C#)

This example writes the data stored in an object to the IntroToVCS.xml by means of the **Serialize** method of the **XmlSerializer** class.

Example

```
public class Book
{
   public string title;

   static void Main()
   {
      Book introToVCS = new Book();
      introToVCS.title = "Intro to Visual CSharp";
      System.Xml.Serialization.XmlSerializer writer =
         new System.Xml.Serialization.XmlSerializer(introToVCS.GetType());
      System.IO.StreamWriter file =
         new System.IO.StreamWriter("c:\\IntroToVCS.xml");

      writer.Serialize(file, introToVCS);
      file.Close();
   }
}
```

Compiling the Code

You can compile the example directly using the command line, or paste the code into a console application using the Visual Studio IDE.

Replace `"c:\\IntroToVCS.xml"` with the actual file name.

Robust Programming

The following conditions may cause an exception:

- The file exists and is read-only (IOException).
- The path name may be too long (PathTooLongException).
- The disk may be full (IOException).

Security

This example creates a new file, if the file does not already exist. If the file already exists, the application overwrites it.

See Also

Code: Reading Class Data from an XML File (Visual C#) | File Access Through StreamReader and StreamWriter Classes | StreamWriter Class | General Language Example Topics

Code: Writing to a Text File (Visual C#)

This example writes a string to a text file using the **WriteLine** method of the **StreamWriter** class.

Example

```
// Compose a string that consists of three lines.
string lines = "First line.\r\nSecond line.\r\nThird line.";

// Write the string to a file.
System.IO.StreamWriter file = new System.IO.StreamWriter("c:\\test.txt");
file.WriteLine(lines);

file.Close();
```

Compiling the Code

Copy the code and paste it into the **Main** method of a console application.

Replace `"c:\\test.txt"` with the actual file name.

Robust Programming

The following conditions may cause an exception:

- The file exists and is read-only (IOException).
- The path name may be too long (PathTooLongException).
- The disk may be full (IOException).

Security

This example creates a new file, if one does not already exist. If the file already exists, the application will overwrite it.

In order to append an existing file, set the Boolean parameter to **true**, as follows:

```
System.IO.StreamWriter file =
   new System.IO.StreamWriter("c:\\test.txt", true);
```

See Also

Code: Reading a Text File One Line at a Time (Visual C#) | Code: Reading From a Text File (Visual C#) | File Access Through StreamReader and StreamWriter Classes | StreamWriter Class | General Language Example Topics

Graphics Programming Example Topics

This topic is designed to help you find code that demonstrates how to perform common graphics programming tasks using Visual C#.NET.

In This Section

- Creating a Bitmap at Run Time
- Creating a Pen
- Creating a Solid Brush
- Drawing a Filled Ellipse on a Form
- Drawing a Filled Rectangle on a Form
- Drawing a Line on a Form
- Drawing Outlined Shapes
- Drawing Text on a Form
- Drawing Vertical Text on a Form
- Print Previewing the Form
- Printing a DataGrid
- Printing a Text File
- Printing the Form
- Setting the Color of a Pen

Related Code Examples

You can also find sample code for Web applications in the following locations:

- C# Language Tour
- Visual C# Samples
- Walkthroughs in Visual Basic and Visual C#
- Quickstarts and other Samples
- .NET Framework Reference Information

Code: Creating a Bitmap at Run Time (Visual C#)

This example creates and fills a **Bitmap** object, then displays it in an existing Windows Forms **PictureBox** control.

Example

```
void CreateBitmap()
{
    System.Drawing.Bitmap flag = new System.Drawing.Bitmap(10, 10);
    for( int x = 0; x <  flag.Height; ++x )
        for( int y = 0; y < flag.Width; ++y )
            flag.SetPixel(x, y, Color.White);
    for( int x = 0; x < flag.Height; ++x )
        flag.SetPixel(x, x, Color.Red);
    pictureBox1.Image = flag;
}
```

Compiling the Code

This example requires:

- A reference to the **System** namespace.

Robust Programming

The following conditions may cause an exception:

- Trying to set a pixel outside the bounds of the bitmap (ArgumentException Class).

See Also

Graphics Programming Example Topics | Bitmap Class

Code: Creating a Pen (Visual C#)

This example demonstrates how to create a **Pen** object.

Example

```
System.Drawing.Pen myPen;
myPen = new System.Drawing.Pen(System.Drawing.Color.Black);
```

Robust Programming

You should always call **Dispose** on any objects that consume system resources, such as **Pen** objects.

See Also

Graphics Programming Example Topics | Pen Class | Pens, Brushes, and Colors

Code: Creating a Solid Brush (Visual C#)

This example demonstrates how to create a **SolidBrush** object.

Example

```
System.Drawing.SolidBrush myBrush;
myBrush = new System.Drawing.SolidBrush(System.Drawing.Color.Black);
```

Robust Programming

You should always call **Dispose** on any objects that consume system resources, such as **Brush** objects.

See Also

Graphics Programming Example Topics | SolidBrush Class | Brush Class | Pens, Brushes, and Colors

Code: Drawing a Filled Ellipse on a Form (Visual C#)

This example demonstrates how to draw a filled ellipse on a form.

Example

```
System.Drawing.SolidBrush myBrush = new System.Drawing.SolidBrush(System.Drawing.Color.Red);
System.Drawing.Graphics formGraphics = this.CreateGraphics();
formGraphics.FillEllipse(myBrush, new Rectangle(0,0,200,300));
myBrush.Dispose();
formGraphics.Dispose();
```

Compiling the Code

This example requires:

- A Windows Application project.

The code needs to be within the scope of the **form** class. The instance of the form is represented by **this**.

Robust Programming

You should always call **Dispose** on any objects that consume system resources, such as **Brush** and **Graphics** objects.

See Also

Graphics Programming Example Topics | Drawing Lines and Shapes with GDI+ | GDI+ Graphics | Graphics.FillEllipse Method

Code: Drawing a Filled Rectangle on a Form (Visual C#)

This example demonstrates how to draw a filled rectangle on a form.

Example

```
System.Drawing.SolidBrush myBrush = new System.Drawing.SolidBrush(System.Drawing.Color.Red);
System.Drawing.Graphics formGraphics = this.CreateGraphics();
formGraphics.FillRectangle(myBrush, new Rectangle(0,0,200,300));
myBrush.Dispose();
formGraphics.Dispose();
```

Compiling the Code

This example requires:

- A Windows Application project.

The code needs to be within the scope of the **form** class. The instance of the form is represented by **this**.

Robust Programming

You should always call **Dispose** on any objects that consume system resources, such as **Brush** and **Graphics** objects.

See Also

Graphics Programming Example Topics | Drawing Lines and Shapes with GDI+ | GDI+ Graphics | Graphics.FillRectangle Method

Code: Drawing a Line on a Form (Visual C#)

This example demonstrates how to draw a line on a form.

Example

```
System.Drawing.Pen myPen;
myPen = new System.Drawing.Pen(System.Drawing.Color.Red);
System.Drawing.Graphics formGraphics = this.CreateGraphics();
formGraphics.DrawLine(myPen, 0, 0, 200, 200);
myPen.Dispose();
formGraphics.Dispose();
```

Compiling the Code

This example requires:

- A Windows Application project.

The code needs to be within the scope of the form class. The instance of the form is represented by **this**.

Robust Programming

You should always call **Dispose** on any objects that consume system resources, such as **Pen** and **Graphics** objects.

See Also

Graphics Programming Example Topics | Drawing Lines and Shapes with GDI+ | GDI+ Graphics | Graphics.DrawLine Method

Code: Drawing Outlined Shapes (Visual C#)

This example demonstrates how to draw outlined ellipses and rectangles on a form.

Example

```
private void DrawEllipse()
{
   System.Drawing.Pen myPen;
   myPen = new System.Drawing.Pen(System.Drawing.Color.Red);
   System.Drawing.Graphics formGraphics = this.CreateGraphics();
   formGraphics.DrawEllipse(myPen, new Rectangle(0,0,200,300));
   myPen.Dispose();
   formGraphics.Dispose();
}
private void DrawRectangle()
{
   System.Drawing.Pen myPen;
   myPen = new System.Drawing.Pen(System.Drawing.Color.Red);
   System.Drawing.Graphics formGraphics = this.CreateGraphics();
   formGraphics.DrawRectangle(myPen, new Rectangle(0,0,200,300));
   myPen.Dispose();
   formGraphics.Dispose();
}
```

Compiling the Code

This example requires

- A Windows Application project.

The code needs to be within the scope of the **form** class. The instance of the form is represented by **this**.

Robust Programming

You should always call **Dispose** on any objects that consume system resources, such as **Pen** and **Graphics** objects.

See Also

Graphics Programming Example Topics | Drawing Lines and Shapes with GDI+ | GDI+ Graphics | Graphics.DrawEllipse Method | Graphics.DrawRectangle Method

Code: Drawing Text on a Form (Visual C#)

This example demonstrates drawing text on a form.

Example

```
private void DrawString()
{
   System.Drawing.Graphics formGraphics = this.CreateGraphics();
   string drawString = "Sample Text";
   System.Drawing.Font drawFont = new System.Drawing.Font("Arial", 16);
   System.Drawing.SolidBrush drawBrush = new System.Drawing.SolidBrush(System.Drawing.Color.B
lack);
   float x = 150.0f;
   float y = 50.0f;
   formGraphics.DrawString(drawString, drawFont, drawBrush, x, y);
   drawFont.Dispose();
   drawBrush.Dispose();
   formGraphics.Dispose();
}
```

Compiling the Code

This example requires:

- A Windows Application project.

The code needs to be within the scope of the **form** class. The instance of the form is represented by **this**.

Robust Programming

You should always call **Dispose** on any objects that consume system resources, such as **Font** and **Graphics** objects.

The following conditions may cause an exception:

- The Arial font is not installed.

See Also

Graphics Programming Example Topics | Graphics.DrawString Method | StringFormat.FormatFlags Property | StringFormatFlags Enumeration | Drawing Text with GDI+

Code: Drawing Vertical Text on a Form (Visual C#)

This example demonstrates drawing text on a form in a vertical orientation.

Example

```
private void DrawVerticalText()
{
   System.Drawing.Graphics formGraphics = this.CreateGraphics();
   string drawString = "Sample Text";
   System.Drawing.Font drawFont = new System.Drawing.Font("Arial", 16);
   System.Drawing.SolidBrush drawBrush = new System.Drawing.SolidBrush(System.Drawing.Color.B
lack);
   float x = 150.0f;
   float y = 50.0f;
   System.Drawing.StringFormat drawFormat = new System.Drawing.StringFormat(StringFormatFlags
.DirectionVertical);
   formGraphics.DrawString(drawString, drawFont, drawBrush, x, y, drawFormat);
   drawFont.Dispose();
   drawBrush.Dispose();
   formGraphics.Dispose();
}
```

Compiling the Code

This example requires:

- A Windows Application project.

The code needs to be within the scope of the **form** class. The instance of the form is represented by **this**.

Robust Programming

You should always call **Dispose** on any objects that consume system resources, such as **Font** and **Graphics** objects.

The following conditions may cause an exception:

- The Arial font is not installed.

See Also

Graphics Programming Example Topics | Graphics.DrawString Method | StringFormat.FormatFlags Property | StringFormatFlags Enumeration | Drawing Text with GDI+

Code: Print Previewing the Form (Visual C#)

This example demonstrates print previewing a copy of the current form.

Example

```
[System.Runtime.InteropServices.DllImport("gdi32.dll")]
public static extern long BitBlt (IntPtr hdcDest, int nXDest, int nYDest, int nWidth, int nHe
ight, IntPtr hdcSrc, int nXSrc, int nYSrc, int dwRop);
private Bitmap memoryImage;
private void CaptureScreen()
{
   Graphics mygraphics = this.CreateGraphics();
   Size s = this.Size;
   memoryImage = new Bitmap(s.Width, s.Height, mygraphics);
   Graphics memoryGraphics = Graphics.FromImage(memoryImage);
   IntPtr dc1 = mygraphics.GetHdc();
   IntPtr dc2 = memoryGraphics.GetHdc();
   BitBlt(dc2, 0, 0, this.ClientRectangle.Width, this.ClientRectangle.Height, dc1, 0, 0, 1336
9376);
   mygraphics.ReleaseHdc(dc1);
   memoryGraphics.ReleaseHdc(dc2);
}
private void printDocument1_PrintPage(System.Object sender, System.Drawing.Printing.PrintPage
EventArgs e)
{
   e.Graphics.DrawImage(memoryImage, 0, 0);
}
private void printButton_Click(System.Object sender, System.EventArgs e)
{
   CaptureScreen();
   printPreviewDialog1.Show();
}
```

Compiling the Code

This example requires:

- A **PrintDocument** component named printDocument1 with a PrintPage event handler.
- A **PrintPreviewDialog** component named printPreviewDialog1 with its Document property set to printDocument1.
- A **Button** named printButton with a Click event handler.

The example code replaces the existing event handlers. A print preview of the form is displayed when printButton is clicked.

Robust Programming

The following conditions may cause an exception:

- You do not have permission to access the printer.
- You do not have permission to use unmanaged code.
- There is no printer installed.
- The print preview dialog box was previously disposed. This occurs after the print preview dialog box is closed.

Security

In order to run this example, you must have permission to execute unmanaged code and to access the printer.

See Also

Graphics Programming Example Topics | Printing with the PrintDocument Component | Rendering Images with GDI+ | Printing Graphics in Windows Forms | PrintDocument Class

Code: Printing a DataGrid (Visual C#)

This example demonstrates printing a **DataGrid** control.

Example

```
private void printGrid_Click(System.Object sender, System.EventArgs e)
{
   printDocument1.Print();
}
private void printDocument1_PrintPage(System.Object sender, System.Drawing.Printing.PrintPage
EventArgs e)
{
   PaintEventArgs myPaintArgs = new PaintEventArgs(e.Graphics, new Rectangle(new Point(0, 0),
this.Size));
   this.InvokePaint(dataGrid1, myPaintArgs);
}
```

Compiling the Code

This example requires:

- A **Button** control named printGrid with a Click event handler.
- A **DataGrid** control named dataGrid1.
- A **PrintDocument** component named printDocument1 with a PrintPage event handler.

The example code replaces the existing event handlers.

Robust Programming

The following conditions may cause an exception:

- You do not have permission to access the printer.
- There is no printer installed.

Security

In order to run this example, you must have permission to access the printer.

See Also

Graphics Programming Example Topics | Code Example: Printing the Form |
Introduction to the Windows Forms PrintDocument Component | Printing Graphics in Windows Forms | PrintDocument Class

Code: Printing a Text File (Visual C#)

This example demonstrates printing a text file.

Example

```
System.IO.StreamReader fileToPrint;
System.Drawing.Font printFont;
private void printButton_Click(object sender, EventArgs e)
{
   string printPath = System.Environment.GetFolderPath(Environment.SpecialFolder.Desktop);
   fileToPrint = new System.IO.StreamReader(printPath + @"\myFile.txt");
   printFont = new System.Drawing.Font("Arial", 10);
   printDocument1.Print();
   fileToPrint.Close();
}

private void printDocument1_PrintPage(object sender, System.Drawing.Printing.PrintPageEventAr
gs e)
{
   float yPos = 0f;
   int count = 0;
   float leftMargin = e.MarginBounds.Left;
   float topMargin = e.MarginBounds.Top;
   string line = null;
   float linesPerPage = e.MarginBounds.Height/printFont.GetHeight(e.Graphics);
   while (count < linesPerPage)
   {
      line = fileToPrint.ReadLine();
      if (line == null)
      {
         break;
      }
      yPos = topMargin + count * printFont.GetHeight(e.Graphics);
      e.Graphics.DrawString(line, printFont, Brushes.Black, leftMargin, yPos, new StringForma
t());
      count++;
   }
   if (line != null)
   {
      e.HasMorePages = true;
   }
}
```

Compiling the Code

This example requires:

- A **Button** control named printButton with a Click event handler.
- A **PrintDocument** component named printDocument1 with a PrintPage event handler.

This example further assumes that you have a text file named myFile.txt on your desktop. The example code replaces the existing event handlers.

Robust Programming

File operations should be enclosed within appropriate **try...catch...finally** blocks. Always call **StreamReader.Close** when you have finished using it.

The following conditions may cause an exception:

- You do not have permission to access the printer.
- You do not have permission to access the file system.

- There is no printer installed.
- The Arial font is not installed.

Security

In order to run this example, you must have permission to access the file system and the printer.

See Also

Graphics Programming Example Topics | Introduction to the Windows Forms PrintDocument Component | StreamReader Class | PrintDocument Class

Code: Printing the Form (Visual C#)

This example demonstrates printing a copy of the current form.

Example

```
[System.Runtime.InteropServices.DllImport("gdi32.dll")]
public static extern long BitBlt (IntPtr hdcDest, int nXDest, int nYDest, int nWidth, int nHe
ight, IntPtr hdcSrc, int nXSrc, int nYSrc, int dwRop);
private Bitmap memoryImage;
private void CaptureScreen()
{
   Graphics mygraphics = this.CreateGraphics();
   Size s = this.Size;
   memoryImage = new Bitmap(s.Width, s.Height, mygraphics);
   Graphics memoryGraphics = Graphics.FromImage(memoryImage);
   IntPtr dc1 = mygraphics.GetHdc();
   IntPtr dc2 = memoryGraphics.GetHdc();
   BitBlt(dc2, 0, 0, this.ClientRectangle.Width, this.ClientRectangle.Height, dc1, 0, 0, 1336
9376);
   mygraphics.ReleaseHdc(dc1);
   memoryGraphics.ReleaseHdc(dc2);
}
private void printDocument1_PrintPage(System.Object sender, System.Drawing.Printing.PrintPage
EventArgs e)
{
   e.Graphics.DrawImage(memoryImage, 0, 0);
}
private void printButton_Click(System.Object sender, System.EventArgs e)
{
   CaptureScreen();
   printDocument1.Print();
}
```

Compiling the Code

This example requires:

- A **PrintDocument** component named printDocument1 with a PrintPage event handler.
- A **Button** named printButton with a Click event handler.

The example code replaces the existing event handlers. The form is printed when `printButton` is clicked.

Robust Programming

The following conditions may cause an exception:

- You do not have permission to access the printer.
- You do not have permission to use unmanaged code.
- There is no printer installed.

Security

In order to run this example, you must have permission to execute unmanaged code and to access the printer.

See Also

Graphics Programming Example Topics | Printing with the PrintDocument Component | Rendering Images with GDI+ | Printing Graphics in Windows Forms | PrintDocument Class

Code: Setting the Color of a Pen (Visual C#)

This example demonstrates how to change the color of a pre-existing **Pen** object.

Example

```
myPen.Color = System.Drawing.Color.PeachPuff;
```

Compiling the Code

This example requires:

- A **Pen** object named *myPen*.

Robust Programming

You should always call **Dispose** on any objects that consume system resources, such as **Pen** objects.

See Also

Graphics Programming Example Topics | Pen Class | Code: Creating a Pen (Visual C#) | Pens, Brushes, and Colors

Web Applications Example Topics

This topic is designed to help you find code that demonstrates how to perform common programming tasks in Web applications using Visual C#.NET.

In This Section

- Adding Controls at Run Time
- Adding Items to a List Control
- Detecting the Browser Version
- Determining the Selected Item in a List Control
- Displaying Data in a Web Forms DataGrid Control
- Passing Data from One Web Forms Page to Another
- Reading a Cookie
- Reading Values From Application State
- Reading Values From Session State
- Reading a Value From View State
- Removing Items From a List Control
- Retrieving a Dataset from View State
- Redirecting to Another Page
- Redirecting to Another Page in the Same Application
- Saving Values in Application State
- Saving Values in Session State
- Saving a Dataset to View State
- Saving a Value in View State
- Selecting an Item in a List Control
- Writing a Cookie

Related Code Examples

You can also find sample code for Web applications in the following locations:

- C# Language Tour
- Visual C# Samples
- Walkthroughs in Visual Basic and Visual C#
- Quickstarts and other Samples
- .NET Framework Reference Information

Code: Adding Controls at Run Time (Visual C#)

This example adds a text box and button to a Web Forms page at run time. It also dynamically binds an event handler to the button's **Click** event. The handler displays the values of the dynamically generated text box.

The controls are added into a Panel Web server control, which acts as a placeholder. The controls are separated in the panel with line breaks (HTML <BR> elements), which are added to the panel using the LiteralControl control.

Example

```
private void Page_Load(object sender, System.EventArgs e)
{
    if (!IsPostBack)
        AddControls();
}

protected override void LoadViewState(object savedState)
{
    base.LoadViewState(savedState);
    if (ViewState["controsladded"] == null)
    AddControls();
}

private void AddControls()
{
    TextBox dynamictextbox = new TextBox();
    dynamictextbox.Text = "(Enter some text)";
    dynamictextbox.ID = "dynamictextbox";
    Button dynamicbutton = new Button();
    dynamicbutton.Click += new System.EventHandler(dynamicbutton_Click);
    dynamicbutton.Text = "Dynamic Button";
    Panel1.Controls.Add(dynamictextbox);
    Panel1.Controls.Add(new LiteralControl("<BR>"));
    Panel1.Controls.Add(new LiteralControl("<BR>"));
    Panel1.Controls.Add(dynamicbutton);
    ViewState["controlsadded"] = true;
}

private void dynamicbutton_Click(Object sender, System.EventArgs e)
{
    TextBox tb = new TextBox();
    tb = (TextBox) (Panel1.FindControl("dynamictextbox"));
    Label1.Text = tb.Text;
}
```

Compiling the Code

This example requires:

- A Web Forms page.
- A **Panel** control called Panel1.
- A **Label** control called Label1.

Security

User input in a Web Forms page can include potentially malicious client script. By default, the Web Forms page validates that user input does not include script or HTML elements. For more information, see Scripting Exploits and Protecting Against Script Exploits in a Web Application.

See Also

Adding Controls to a Web Forms Page Programmatically | Code: Reading a Value From View State (Visual C#) | Code: Saving a Value in View State (Visual C#) | Introduction to Web Forms State Management |

Code: Adding Items to a List Control (Visual C#)

This example uses the **Add** method of the ListBox.Items property to add items to the list.

Example

```
// Create a string array and assign items.
string[] ListContents = {"Enju", "Sue", "Mary"};

// Add the array items to the list box.
for (int i = 0; i <= ListContents.Length - 1; i++)
   ListBox1.Items.Add(ListContents[i]);
```

Compiling the Code

Start a new ASP.NET Web application and do the following:

- Add a list box named `ListBox1`.
- Copy the code and paste it into the **Page_Load** method.

See Also

ListBox.ObjectCollection Class | Code: Removing Items From a List Control (Visual C#) | Web Applications Example Topics

Code: Detecting the Browser Version (Visual C#)

This example examines the Request.Browser property to determine the type and version of the client's Web browser. There are many other browser attributes that can be accessed by means of the **Request.Browser** property, such as **Url**, **ServerVariables**, and **User**.

Example

```
HttpBrowserCapabilities brObject = Request.Browser;

// Display the type and version.
Response.Write("Browser Type: "+ brObject.Type);
Response.Write("<p>"+"Browser Version: "+ brObject.Version);
```

Compiling the Code

Start a new ASP.NET Web application and paste the code into the **Page_Load** method.

See Also

Detecting Browser Types in Web Forms | Request.Browser | BrowserCapabilities Class | Web Applications Example Topics

Code: Determining the Selected Item in a List Control (Visual C#)

This example uses the **SelectedIndex** and **SelectedItem** properties of a List control to access information about the selected item.

Example

```
// The index:
int ListItemIndex;
// Value of the item:
string ListItemValue;

ListItemIndex = ListBox1.SelectedIndex;
ListItemValue = ListBox1.SelectedItem.Value.ToString();
```

Compiling the Code

Start a new ASP.NET Web application and do the following:

- Add a list box named `ListBox1` and populate it with items. For details, see Code: Adding Items to a List Control (Visual C#).
- Copy the code and paste it into the **Page_Load** method.

See Also

ListBox.ObjectCollection Class | Code: Adding Items to a List Control (Visual C#) | Web Applications Example Topics

Code: Displaying Data in a Web Forms DataGrid Control (Visual C#)

This example uses the DataBind method to bind the contents of a dataset to a **DataGrid** control.

Example

```
if (!IsPostBack)
{
    sqlDataAdapter1.Fill(dataSet11);
    DataGrid1.DataBind();
}
```

Compiling the Code

This example requires:

- A Web Forms page.
- A **DataGrid** control named `DataGrid1`.
- A typed dataset named `dataSet11`, which is an instance of the typed dataset class `DataSet1`.
- A properly configured data adapter named `sqlDataAdapter1`. You can bind the **DataGrid** control to any Web Forms page data source.
- The application must have permission to access the database. For details, see Using SQL Server in Web Applications.

Robust Programming

You can specify the **DataGrid** control's **DataSource** property at design time in the Properties window. You do not have to specify it at run time, as is shown in the previous example.

You should always wrap data access calls in a **try-catch** block:

```
try
{
    sqlDataAdapter1.Fill(dataSet11);
    DataGrid1.DataBind();
}
catch
{
    Label1.Text = "Unable to connect to the database.";
}
```

You should bind the **DataGrid** control to its data source only when there is a change to the data or to the control's state. Do not data bind the control on each round trip. In the previous example, the control is bound only the first time the page runs, not on every postback.

Web Forms pages often benefit by caching a dataset rather than requerying the database. For details, see Code: Saving a Dataset to ViewState (Visual C#).

Security

If you display error information, do not include such information as a user name that a malicious user might be able to take advantage of. For details, see Displaying Safe Error Messages.

See Also

Adding DataGrid Web Server Controls to a Web Forms Page | Data Access in Web Forms Pages | Data Sources for Web Forms Pages | Introduction to Data Access in Web Forms Pages | State Management Recommendations | Web Applications Example Topics

Code: Passing Data from One Web Forms Page to Another (Visual C#)

This example stores information in session state and navigates to another page in the same application, then retrieves the stored information in the second page.

Example

```
// First part: Saves info on first page.
string firstName = "John";
string lastName = "Smith";
string city = "Seattle";
Session.Add("First", firstName);
Session.Add("Last", lastName);
Session.Add("City", city);

// Redirect to second page.
Server.Transfer("WebForm2.aspx");

// Second part: Reads info into variables on the second page.
string firstName = (string)(Session["First"]);
string lastName = (string)(Session["Last"]);
string city = (string)(Session["City"]);
```

Compiling the Code

- Start an ASP.NET application with two Web Forms pages, WebForm1.aspx and WebForm2.aspx.
- Paste the first part of the code into the **Page_Load** method (or a **Click** method) of WebForm1.aspx.
- Paste the second part of the code into the **Page_Load** method of WebForm2.aspx.

Robust Programming

Session state information is available only to pages in the same application. Storing information in session state uses server memory. For more information, see State Management Recommendations.

See Also

Code: Saving Values in Session State (Visual C#)| Introduction to Web Forms State Management |
Saving Web Forms Page Values Using View State | State Management Recommendations | ASP.NET State Management |
Web Applications Example Topics

Code: Reading a Cookie (Visual C#)

The following example uses the HttpCookie class and its properties to read a cookie with a specific name.

Example

```
HttpCookie myCookie = new HttpCookie("MyTestCookie");
myCookie = Request.Cookies["MyTestCookie"];

// Read the cookie information and display it.
if (myCookie != null)
   Response.Write("<p>"+ myCookie.Name + "<p>"+ myCookie.Value);
else
   Response.Write("not found");
```

Compiling the Code

Start a new ASP.NET Web application and paste the code into the **Page_Load** method.

Robust Programming

Replace the name `"MyTestCookie"` with the name of the cookie to be read.

Values returned from a cookie are always of type **string.**

Security

The browser can only send the data back to the server that originally created the cookie. However, malicious users can access cookies and read their contents.

See Also

Code: Writing a Cookie (Visual C#) | HttpCookie Class | Web Applications Example Topics

Code: Reading Values From Application State (Visual C#)

This example retrieves a value from the HttpApplicationState object.

Example

```
string appMsg;
appMsg = (string)(Application["Message"]);
```

Compiling the Code

Start a new ASP.NET application and paste the code into the **Page_Load** method.

Robust Programming

You must cast the data to the appropriate type, as shown in the example.

No exception is thrown if you attempt to get a value out of application state that does not exist. To be sure that the value you want is in application state, check first for the existence of the object with a test such as the following:

```
if (Application["Message"] == null)
    Response.Write("No such value in application state.");
```

If you attempt to use a non-existent application state entry in some other way (for example, to examine its type), a NullReferenceException exception is thrown.

See Also

Code: Saving Values in Application State (Visual C#) | Application State | Introduction to Web Forms State Management | State Management Recommendations | ASP.NET State Management | Web Applications Example Topics

Code: Reading Values From Session State (Visual C#)

This example accesses the HttpSessionState.Item property to retrieve the values in session state.

Example

```
string firstName = (string)(Session["First"]);
string lastName = (string)(Session["Last"]);
string city = (string)(Session["City"]);
```

Compiling the Code

Start a new ASP.NET application and paste the code into the **Page_Load** method.

Robust Programming

Session values are of type **Object**. You should cast to the appropriate type when reading them.

No exception is thrown if you attempt to get a value out of session state that does not exist. To be sure that the value you want is in session state, check first for the existence of the object with a test such as the following:

```
if (Session["City"] == null)
    Response.Write("No such value in session state.");
```

If you attempt to use a non-existent session state entry in some other way (for example, to examine its type), a NullReferenceException exception is thrown.

See Also

Code: Reading a Value From View State (Visual C#) | Introduction to Web Forms State Management | Saving Web Forms Page Values Using View State | State Management Recommendations | ASP.NET State Management | Web Applications Example Topics

Code: Reading a Value From View State (Visual C#)

This example saves the value `"yellow"` to the **Control.ViewState** property and then, when the page posts back to the server, retrieves a value of the property. For more information on **ViewState**, see Saving Web Forms Page Values Using View State.

Example

```
string strColor;
if (Page.IsPostBack)
{
   // Retrieve and display the property value.
   strColor = (string)ViewState["color"];
   Response.Write(strColor);
}
else
   // Save the property value.
   ViewState["color"] = "yellow";
```

Compiling the Code

Copy and paste the code into the **Page_Load** method of an ASP.NET Web application.

Robust Programming

As shown in the example, retrieving the **ViewState** property occurs when the Web page is posted back to the server. Otherwise, the string will be **null**.

For more information on usage recommendations, see State Management Recommendations.

See Also

Code: Saving a Value in View State (Visual C#) | State Management Recommendations | Developing High-Performance ASP.NET Applications | Web Applications Example Topics

Code: Removing Items From a List Control (Visual C#)

This example uses the **Remove** method of the ListBox.Items property to remove an item from the list.

Example

```
ListBox1.Items.Remove("Mary");
```

Compiling the Code

Start a new ASP.NET Web application and do the following:

- Add a list box named `ListBox1` and populate it with items. For details, see Code: Adding Items to a List Control (Visual C#).
- Copy the code and paste it into the **Page_Load** method.

Robust Programming

Replace `"Mary"` with the value of the item to be removed.

See Also

ListBox.ObjectCollection Class | Code: Adding Items to a List Control (Visual C#) | Web Applications Example Topics

Code: Retrieving a Dataset from View State (Visual C#)

This example retrieves the **DataSet** from a view state.

Example

```
if (Page.IsPostBack)
{
   System.IO.StringReader sr =
      new System.IO.StringReader((string)(ViewState["dSet"]));
   dSet.ReadXml(sr);
}
```

Compiling the Code

Start a new ASP.NET Web application and do the following:

- Add a **DataAdapter** named *SqlDataAdapter1*.
- Generate a **DataSet** (the default name of the **DataSet** class is **DataSet1**).
- Create an instance of **DataSet1** named *dSet*.
- Copy the code and paste it into the **Page_Load** method.

Robust Programming

As shown in the example, retrieving the **DataSet** occurs when the Web page is posted back to the server.

Security

For information about how to use a SQL Server database on other machines, see Using SQL Server in Web Applications.

See Also

Code: Saving a Dataset to ViewState (Visual C#) | Web Applications Example Topics

Code: Redirecting to Another Page (Visual C#)

This example uses the HttpResponse.Redirect method to redirect the browser to the specified URL.

Example

```
Response.Redirect("http://www.microsoft.com");
```

Compiling the Code

Start a new ASP.NET application and paste the code into the **Page_Load** method.

Robust Programming

The **Redirect** method is executed by the browser and therefore results in an extra round trip between server and browser.

Make sure that the target page exists.

Security

The method is subject to spoofing or tampering because it is executed by the client.

See Also

Code: Redirecting to Another Page in the Same Application (Visual C#) | HttpResponse Class | HttpResponse.Redirect Method | Web Forms Pages | Web Applications Example Topics

Code: Redirecting to Another Page in the Same Application (Visual C#)

This example uses the Server.Transfer method to redirect the browser to the specified page.

Example

```
Server.Transfer("WebForm2.aspx");
```

Compiling the Code

- Start an ASP.NET application with two Web Forms pages, WebForm1.aspx and WebForm2.aspx.
- Paste the code into the **Page_Load** method (or a **Click** method) of WebForm1.aspx.

Robust Programming

The page transferred to should be another Web Forms page (.aspx page) in the same application. You cannot use **Server.Transfer** to redirect to an .asp or .asmx page.

Make sure that the target page exists.

Because **Server.Transfer** is executed on the server, the browser does not have a record of the page change. Therefore, if the user refreshes the page, unexpected results can occur.

See Also

Code: Redirecting to Another Page (Visual C#) | HttpServerUtility Class | HttpServerUtility.Transfer Method | Web Forms Pages | Web Applications Example Topics

Code: Saving Values in Application State (Visual C#)

This example uses the HttpApplicationState class to save global information.

Example

```
Application["Message"] = "AppMsg";
Application["AppStartTime"] = DateTime.Now;
```

Compiling the Code

Start a new ASP.NET application and paste the code into the **Page_Load** method.

Robust Programming

Storing information in application state uses server memory. An alternative is to store information in view state. For more information, see Code: Saving a Value in View State (Visual C#) and State Management Recommendations.

See Also

Code: Reading Values From Application State (Visual C#) | Application State | Introduction to Web Forms State Management | State Management Recommendations | ASP.NET State Management | Web Applications Example Topics

Code: Saving Values in Session State (Visual C#)

This example uses the Session.Add method to persist values within an individual session.

Example

```
string firstName = "Jeff";
string lastName = "Smith";
string city = "Seattle";

Session.Add("First", firstName);
Session.Add("Last", lastName);
Session.Add("City", city);
```

Compiling the Code

Start a new ASP.NET application and paste the code into the **Page_Load** method.

Robust Programming

Storing information in session state uses server memory. An alternative is to store information in view state. For more information, see Code: Saving a Value in View State (Visual C#) and State Management Recommendations.

Server state can expire after 20 minutes of inactivity (by default) and the information that you store there can be lost.

See Also

HttpSessionState Class | Introduction to Web Forms State Management | Saving Web Forms Page Values Using View State | State Management Recommendations | ASP.NET State Management | Web Applications Example Topics

Code: Saving a Dataset to View State (Visual C#)

This example uses the **Control.ViewState** property to save a dataset.

Example

```
sqlDataAdapter1.Fill(dSet);
System.IO.StringWriter sw = new System.IO.StringWriter();

// Write the DataSet to the ViewState property.
dSet.WriteXml(sw);
ViewState["dSet"] = sw.ToString();
```

Compiling the Code

Start a new ASP.NET Web application and do the following:

- Add a **DataAdapter** named *SqlDataAdapter1*.
- Generate a dataset and then create an instance named *dSet*.
- Copy the code and paste it into the **Page_Load** method.

Security

For information about how to use a SQL Server database on other machines, see Using SQL Server in Web Applications.

See Also

Code: Retrieving a Dataset from ViewState (Visual C#) | Web Applications Example Topics

Code: Saving a Value in View State (Visual C#)

This example saves the string value `"yellow"` to the **Control.ViewState** property. For more information, see Saving Web Forms Page Values Using View State.

Example

```
ViewState["color"] = "yellow";
```

Compiling the Code

Start a new ASP.NET Web application and paste the code into the **Page_Load** method.

Robust Programming

Replace the property string with an appropriate string.

For more information on usage recommendations, see State Management Recommendations.

See Also

Code: Reading a Value From View State (Visual C#) | State Management Recommendations | Developing High-Performance ASP.NET Applications | Web Applications Example Topics

Code: Selecting an Item in a List Control (Visual C#)

This example uses the **IndexOf** method to search a list box for a specific text.

Example

```
ListBox1.SelectedIndex = ListBox1.Items.IndexOf(ListBox1.Items.FindByText("ValueToFind"));
```

Compiling the Code

Start a new ASP.NET Web application and do the following:

- Add a list box named `ListBox1` and populate it with items. For details, see Code: Adding Items to a List Control (Visual C#).
- Copy the code and paste the code into the **Page_Load** method.
- Replace "*ValueToFind*" with the text you are searching for.

See Also

ListBox.ObjectCollection Class | Code: Adding Items to a List Control (Visual C#) | Web Applications Example Topics

Code: Writing a Cookie (Visual C#)

The following example uses the HttpCookie class and its properties to write a cookie that expires after one minute.

Example

```
HttpCookie myCookie = new HttpCookie("MyTestCookie");
DateTime now = DateTime.Now;

// Set the cookie value.
myCookie.Value = now.ToString();
// Set the cookie expiration date.
myCookie.Expires = now.AddMinutes(1);

// Add the cookie.
Response.Cookies.Add(myCookie);

Response.Write("<p> The cookie has been written.");
```

Compiling the Code

Start a new ASP.NET Web application and paste the code into the **Page_Load** method.

Robust Programming

Replace the name `"MyTestCookie"` and the cookie value with a name and value of your choice.

Security

Information stored in a cookie can be lost because the user has full control over cookies and can disable cookies entirely.

See Also

Code: Reading a Cookie (Visual C#) | HttpCookie Class | Web Applications Example Topics

Windows Forms Example Topics

This topic is designed to help you find code that demonstrates how to perform common programming tasks in Windows Forms applications using Visual C#.NET.

In This Section

- Adding a Node to the Selected Node of a TreeView Control
- Adding Items to a ListBox Control
- Adding Nested Nodes to a TreeView Control
- Adding Nodes to a TreeView Control
- Changing the Background Color of a Form
- Converting the Text in a TextBox Control to an Integer
- Creating a Context Menu and Attaching it to a Control
- Creating a Group of Radio Buttons from a String Array
- Creating a Shaped Form
- Creating a ToolTip for a Control
- Databinding a DataTable of String Values to a Windows DataGrid Control
- Databinding an Array of String Values to a Windows ListBox Control
- Determining the Selected Cell in a Windows DataGrid Control
- Determining the Selected Item in a ListBox Control
- Determining the Selected Node in a TreeView Control
- Determining the Selected Text in a TextBox Control
- Determining Which Control on a Form is Selected
- Displaying a Web Page from a LinkLabel Control
- Displaying an OpenFileDialog Dynamically
- Displaying One Form from Another
- Displaying the Time in a StatusBar Control
- Drawing Graphics on a Windows Form
- Formatting Characters in Bold in a RichTextBox Control
- Getting a Value from Another Form
- Hiding a DataColumn in a DataGrid Where the DataSource is a DataTable
- Hiding a DataColumn in a DataGrid Where the DataSource is an Array
- Hiding a DataColumn in a DataGrid Where the DataSource is an ArrayList
- Populating a ListBox Control with an Array of Strings
- Retrieving an Image that is an Embedded Resource
- Retrieving the Data from the Selected Cell in a Windows DataGrid Control
- Searching for an Item in a ListBox Control
- Selecting a Range of Dates in a Calendar Control
- Selecting an Item in a ListBox Control

Related Code Examples

You can also find sample code for C# applications in the following locations:

- C# Language Tour
- Visual C# Samples
- Walkthroughs in Visual Basic and Visual C#
- Quickstarts and other Samples
- .NET Framework Reference Information

Code: Adding a Node to the Selected Node of a TreeView Control (Visual C#)

This example adds one node to the selected node of an existing **TreeView** control.

Example

```
TreeNode node = treeView1.SelectedNode;
node.Nodes.Add("New node below selected node");
```

Compiling the Code

This example requires:

- A **TreeView** control named treeView1.

Robust Programming

The following conditions may cause an exception:

- The **TreeView** control has no nodes or none of the nodes is selected (NullReferenceException Class).

See Also

Windows Forms Example Topics | Code: Adding Nodes to a TreeView Control (Visual C#) | Code: Adding Nested Nodes to a TreeView Control (Visual C#) | TreeView.SelectedNode Property | TreeView.Nodes Property | TreeView Control (Windows Forms)

Code: Adding Items to a ListBox Control (Visual C#)

This example adds the contents of a Windows Forms **TextBox** control to a **ListBox** control when the **TextBox** control loses focus.

Example

```
private void textBox_Leave(object sender, System.EventArgs e)
{
    listBox1.Items.Add(((TextBox)sender).Text);
}
```

Compiling the Code

This example requires:

- A form with a **ListBox** control named listBox1 and three **TextBox** controls named textBox1, textBox2, and textBox3. Set each TextBox's **Leave** event handler to textBox_Leave.

  Note This can also be used with a **ComboBox** control by substituting a **ComboBox** control named comboBox1 for the **ListBox** control and changing the code from listBox1 to comboBox1.

Robust Programming

To avoid duplicate entries in the list, you should search for a match before adding the contents of the **TextBox** to the list. For an example, see Code: Searching for an Item in a ListBox Control.

See Also

Windows Forms Example Topics

Code: Adding Nested Nodes to a TreeView Control (Visual C#)

This example adds a first-level node and a second-level node to an existing **TreeView** control.

Example

```
TreeNode node = treeView1.Nodes.Add("Level one node");
node.Nodes.Add("Level two node");
```

Compiling the Code

This example requires:

- A **TreeView** control named treeView1.

See Also

Windows Forms Example Topics | Code: Adding a Node to the Selected Node of a TreeView Control (Visual C#) | Code: Adding Nodes to a TreeView Control (Visual C#) | TreeView.Nodes Property | TreeView Control (Windows Forms)

Code: Adding Nodes to a TreeView Control (Visual C#)

This example adds three first-level nodes to an existing **TreeView** control.

Example

```
treeView1.Nodes.Add("First Node");
treeView1.Nodes.Add("Second Node");
treeView1.Nodes.Add("Third Node");
```

Compiling the Code

This example requires:

- A **TreeView** control named treeView1.

See Also

Windows Forms Example Topics | Code: Adding a Node to the Selected Node of a TreeView Control (Visual C#) | Code: Adding Nested Nodes to a TreeView Control (Visual C#) | TreeView.Nodes Property | TreeView Control (Windows Forms)

Code: Changing the Background Color of a Form (Visual C#)

This example changes the background color of a Windows Form programmatically.

Example

```
private void Form1_Click(object sender, System.EventArgs e)
{
    this.BackColor = System.Drawing.Color.DarkBlue;
}
```

Compiling the Code

This example requires:

- A form named Form1. Set Form1's **Click** event handler to Form1_Click.

See Also

Windows Forms Example Topics | Color Structure

Code: Converting the Text in a TextBox Control to an Integer (Visual C#)

This example demonstrates two methods of converting text data to integer data.

Example

```
int anInteger;
anInteger = Convert.ToInt32(textBox1.Text);
anInteger = int.Parse(textBox1.Text);
```

Compiling the Code

This example requires:

- A **TextBox** control named textBox1.

Robust Programming

The following conditions may cause an exception:

- The text converts to a number that is too large or too small to store as an **int** (OverflowException Class).
- The text may not represent a number (FormatException Class).

See Also

Windows Forms Example Topics | Convert Class | Int32.Parse Method

Code: Creating a Context Menu and Attaching it to a Control (Visual C#)

This example programmatically creates a Windows Forms context menu and associates it with a control.

Example

```
private void Form1_Load(object sender, System.EventArgs e)
{
    System.Windows.Forms.ContextMenu contextMenu1;
    contextMenu1 = new System.Windows.Forms.ContextMenu();
    System.Windows.Forms.MenuItem menuItem1;
    menuItem1 = new System.Windows.Forms.MenuItem();
    System.Windows.Forms.MenuItem menuItem2;
    menuItem2 = new System.Windows.Forms.MenuItem();
    System.Windows.Forms.MenuItem menuItem3;
    menuItem3 = new System.Windows.Forms.MenuItem();

    contextMenu1.MenuItems.AddRange(new System.Windows.Forms.MenuItem[] {menuItem1, menuItem2
, menuItem3});
    menuItem1.Index = 0;
    menuItem1.Text = "MenuItem1";
    menuItem2.Index = 1;
    menuItem2.Text = "MenuItem2";
    menuItem3.Index = 2;
    menuItem3.Text = "MenuItem3";

    textBox1.ContextMenu = contextMenu1;
}
```

Compiling the Code

This example requires:

- A Windows Form named Form1 and a **TextBox** control named textBox1. Set Form1's **Load** event handler to Form1_Load.

See Also

Windows Forms Example Topics | ContextMenu Component (Windows Forms)

Code: Creating a Group of Radio Buttons from a String Array (Visual C#)

This example programmatically creates a group of Windows Forms radio buttons and set their **Text** properties to values from an array of strings.

Example

```
private void button1_Click(object sender, System.EventArgs e)
{
    string [] stringArray = new string[3];

    stringArray[0] = "Yes";
    stringArray[1] = "No";
    stringArray[2] = "Maybe";

    System.Windows.Forms.RadioButton [] radioButtons = new System.Windows.Forms.RadioButton[3
];

    for(int i=0; i<3; ++i)
    {
        radioButtons[i] = new RadioButton();
        radioButtons[i].Text = StringArray[i];
        radioButtons[i].Location = new System.Drawing.Point(10, 10+i*20);
        this.Controls.Add(radioButtons[i]);
    }
}
```

Compiling the Code

This example requires:

- A Windows Form with a **Button** control named button1. Set button1's **Click** event handler to button1_Click.

See Also

Windows Forms Example Topics | RadioButton Control (Windows Forms) | RadioButton Class

Code: Creating a Shaped Form (Visual C#)

This example gives a form an elliptical shape.

Example

```
protected override void OnPaint( System.Windows.Forms.PaintEventArgs e )
{
    System.Drawing.Drawing2D.GraphicsPath shape = new System.Drawing.Drawing2D.GraphicsPath()
;
    shape.AddEllipse(0, 0, this.Width, this.Height);
    this.Region = new System.Drawing.Region(shape);
}
```

Compiling the Code

This example overrides the **OnPaint** method to change the shape of the form. To use this code, copy the method declaration as well as the drawing code inside the method.

The **Region** property of the **Form** class is an advanced member. To use this property in IntelliSense, see General, Basic, Text Editor, Options Dialog Box.

See Also

Windows Forms Example Topics | Control.OnPaint Method | Region Class | System.Drawing Namespace | GraphicsPath.AddEllipse Method | Control.Region Property

Code: Creating a ToolTip for a Control (Visual C#)

This example programmatically creates a ToolTip for a Windows Forms control.

Example

```
private void Form1_Load(object sender, System.EventArgs e)
{
    System.Windows.Forms.ToolTip ToolTip1 = new System.Windows.Forms.ToolTip();
    ToolTip1.SetToolTip(this.textBox1, "Hello");
}
```

Compiling the Code

This example requires:

- A form named Form1 with a **TextBox** control named textBox1. Set Form1's **Load** event handler to Form1_Load.

See Also

Windows Forms Example Topics

Code: Databinding a DataTable of String Values to a Windows DataGrid Control (Visual C#)

This example creates a data table with two columns and two rows and then binds it to an existing **DataGrid** control.

Example

```
DataTable dt = new DataTable();
dt.Columns.Add(new DataColumn("Item", typeof(string)));
dt.Columns.Add(new DataColumn("Color", typeof(string)));
dt.Rows.Add(new string[] {"table", "brown"});
dt.Rows.Add(new string[] {"chair", "white"});
dataGrid1.DataSource = dt;
```

Compiling the Code

This example requires:

- A **DataGrid** control named dataGrid1.

See Also

Windows Forms Example Topics | DataGrid Control (Windows Forms) | DataGrid.DataSource Property | DataTable Class

Code: Databinding an Array of String Values to a Windows ListBox Control (Visual C#)

This example adds three strings to an existing **ListBox** control.

Example

```
listBox1.DataSource = new string[] { "one", "two", "three" };
```

Compiling the Code

This example requires:

- A **ListBox** control named listBox1.

See Also

Windows Forms Example Topics | ListBox Control (Windows Forms) | ListControl.DataSource Property | ListBox.Items Property

Code: Determining the Selected Cell in a Windows DataGrid Control (Visual C#)

This example sets a reference to the selected cell of an existing **DataGrid** control.

Example

```
System.Windows.Forms.DataGridCell selectedCell = dataGrid1.CurrentCell;
```

Compiling the Code

This example requires:

- A **DataGrid** control named dataGrid1.

Robust Programming

If the **DataGrid** is not bound to a data source, or if no cell is selected, then the selected cell is the cell in row 0 and column 0.

See Also

Windows Forms Example Topics | Code: Retrieving the Data from the Selected Cell in a Windows DataGrid Control | DataGrid.CurrentCell Property | DataGrid Control (Windows Forms)

Code: Determining the Selected Item in a ListBox Control (Visual C#)

This example determines which item has been selected in a Windows Forms **ListBox** control.

Example

```
private void Form1_Load(object sender, System.EventArgs e)
{
    listBox1.Items.Add("One");
    listBox1.Items.Add("Two");
    listBox1.Items.Add("Three");
}

private void listBox1_SelectedIndexChanged(object sender, System.EventArgs e)
{
    if ((string)listBox1.SelectedItem == "Two")
        MessageBox.Show((string)listBox1.SelectedItem);
}
```

Compiling the Code

This example requires:

- A form named Form1 with a **ListBox** control named listBox1. Set Form1's **Load** event handler to Form1_Load. Set listBox1's **SelectedIndexChanged** event handler to listBox1_SelectedIndexChanged.

  Note This can also be used with a **ComboBox** control by substituting a **ComboBox** control named comboBox1 for the **ListBox** control and changing the code from listBox1 to comboBox1.

See Also

Windows Forms Example Topics

Code: Determining the Selected Node in a TreeView Control (Visual C#)

This example sets a reference to the selected node of existing **TreeView** control.

Example

```
TreeNode node = treeView1.SelectedNode;
```

Compiling the Code

This example requires:

- A **TreeView** control named treeView1.

Robust Programming

The following conditions may cause an exception:

- The **TreeView** control has no nodes or none of the nodes is selected (NullReferenceException Class).

See Also

Windows Forms Example Topics | Code: Adding Nodes to a TreeView Control (Visual C#) | Code: Adding Nested Nodes to a TreeView Control (Visual C#) | TreeView.SelectedNode Property | TreeView.Nodes Property | TreeView Control (Windows Forms)

Code: Determining the Selected Text in a TextBox Control (Visual C#)

This example programmatically selects text in a Windows Forms **TextBox** control and retrieves the selected text.

Example

```
private void button1_Click(object sender, System.EventArgs e)
{
    textBox1.Text = "Hello World";
    textBox1.Select(6, 5);
    MessageBox.Show(textBox1.SelectedText);
}
```

Compiling the Code

This example requires:

- A form with a **TextBox** control named textBox1 and a **Button** control named button1. Set button1's **Click** event handler to button1_Click.

  Note This can also be used with a **RichTextBox** control by substituting a **RichTextBox** control named richTextBox1 for the **TextBox** control and changing the code from textBox1 to richTextBox1.

Robust Programming

In this example you are setting the **Text** property before retrieving the **SelectedText** value. In most cases, you will be retrieving text entered by the user, so you will want to add error-handling code in case the text is too short.

See Also

Windows Forms Example Topics

Code: Determining Which Control on a Form is Selected (Visual C#)

This example displays the name of the currently selected Windows Forms control in a **Label** control.

Example

```
private void DisplayNameOfActiveControl()
{
    label1.Text = this.ActiveControl.Name;
}
```

Compiling the Code

This example requires:

- A form with a **Label** control named label1.

See Also

Windows Forms Example Topics

Code: Displaying a Web Page from a LinkLabel Control (Visual C#)

This example displays a Web page in the default browser when a user clicks a Windows Forms **LinkLabel** control.

Example

```
private void Form1_Load(object sender, System.EventArgs e)
{
    linkLabel1.Text = "Click here to get more info.";
    linkLabel1.Links.Add(6, 4, "www.microsoft.com");
}
private void linkLabel1_LinkClicked(object sender, System.Windows.Forms.LinkLabelLinkClickedE
ventArgs e)
{
    System.Diagnostics.Process.Start(e.Link.LinkData.ToString());
}
```

Compiling the Code

This example requires:

- A form named Form1 with a **LinkLabel** control named linkLabel1.
- An active Internet connection.

Secure programming

The call to the **Start** method requires full trust. For more information, see SecurityException Class.

See Also

Windows Forms Example Topics | LinkLabel Control (Windows Forms) | LinkLabel Class

Code: Displaying an OpenFileDialog Dynamically (Visual C#)

This example instantiates and displays an **OpenFileDialog** control.

Example

```
OpenFileDialog openFile = new OpenFileDialog();
openFile.DefaultExt = "doc";
// The Filter property requires a search string after the pipe ( | )
openFile.Filter = "Word documents (*.doc)|*.doc";
openFile.ShowDialog();
if( openFile.FileNames.Length > 0 )
{
    foreach( string filename in openFile.FileNames )
    {
        // Insert code here to process the files.
    }
}
```

Robust Programming

You can use the **FileOK** event of the **OpenFileDialog** control to be certain that the user selected a file and clicked the OK button on the dialog box.

Use the **CheckFileExists**, **CheckPathExists**, **DefaultExtension**, **Filter**, **Multiselect**, and **ValidateNames** properties of the **OpenFileDialog** control to limit run-time errors.

See Also

Windows Forms Example Topics | OpenFileDialog Class

Code: Displaying One Form from Another (Visual C#)

This example displays a second form from another Windows Form.

Example

```
private void button1_Click(object sender, System.EventArgs e)
{
    Form2 frm = new Form2();
    frm.Show();
}
```

Compiling the Code

This example requires:

- Two Windows Forms named Form1 and Form2. Form1 contains a **Button** control named button1. Set button1's **Click** event handler to button1_Click.

See Also

Windows Forms Example Topics | Windows Forms

Code: Displaying the Time in a StatusBar Control (Visual C#)

This example displays the current time in hh:mm format in a **StatusBar** control.

Example

```
private void timer1_Tick(object sender, System.EventArgs e)
{
    statusBar1.Panels[0].Text = DateTime.Now.ToShortTimeString();
}
```

Compiling the Code

The form must contain a **Timer** control with **Enabled** set to **True**, and a **StatusBar** with a **Panel** added to the **Panels** property and the **ShowPanels** property set to **True**.

The code is part of the event handler for the **Timer** control's **Tick** event, as shown.

Robust Programming

The following conditions may cause an exception:

- The **StatusBar** has no **Panel** controls (ArgumentOutOfRangeException Class).

See Also

Windows Forms Example Topics | Panel.Text Property | DateTime.ToShortTimeString Method | Timer.Tick Event

Code: Drawing Graphics on a Windows Form (Visual C#)

This example draws a circle and a square on a form.

Example

```
System.Drawing.Graphics graphics = this.CreateGraphics();
System.Drawing.Rectangle rectangle = new System.Drawing.Rectangle(100, 100, 200, 200);
graphics.DrawEllipse(System.Drawing.Pens.Black, rectangle);
graphics.DrawRectangle(System.Drawing.Pens.Red, rectangle);
```

Compiling the Code

This code is to be added to a class that derives from **System.Windows.Forms.Form**. The **this** refers to the instance of the form.

Security

To run this process, your assembly requires a privilege level granted by the UIPermission Class. If you are running in a partial-trust context, the process might throw an exception due to insufficient privileges. For more information, see Code Access Security Basics.

See Also

Windows Forms Example Topics | Graphics Class | Graphics.DrawEllipse Method | Graphics.DrawRectangle Method

Code: Formatting Characters in Bold in a RichTextBox Control (Visual C#)

This example adds the text "This text is in **bold**." to an existing **RichTextBox** control.

Example

```
richTextBox1.Rtf = @"{\rtf1\ansi This is in \b bold\b0.}";
```

Compiling the Code

This example requires:

- A **RichTextBox** control named richTextBox1.

Robust Programming

The rich text format is extremely flexible, but any errors in the format lead to errors in the displayed text. For more information, see Rich Text Format (RTF) Specification, version 1.6.

See Also

Windows Forms Example Topics | RichTextBox Control (Windows Forms)

Code: Getting a Value from Another Form (Visual C#)

This example retrieves a value from a text box on a Windows Form and displays it in a text box on another form.

Example

In Form1.cs:

```
private Form2 otherForm;
private void GetOtherFormTextBox()
{
    textBox1.Text = otherForm.TextBox1.Text;
}
```

In Form2.cs:

```
public TextBox TextBox1
{
    get
    {
        return textBox1;
    }
}
```

Compiling the Code

This example requires:

- Two forms named Form1 and Form2, each containing a **TextBox** control named textBox1. Form1 should create an instance of Form2 and assign it to otherForm; **GetOtherFormTextBox** will copy the text in Form2's textBox1 to Form1's textBox1.

See Also

Windows Forms Example Topics | Displaying One Form from Another

Code: Hiding a DataColumn in a DataGrid Where the DataSource is a DataTable (Visual C#)

This example hides the "X" column of a **DataTable** object that is displayed in an existing Windows Forms **DataGrid** control.

Example

```
private void HideColumnOfDataSet()
{
    System.Data.DataTable points = new System.Data.DataTable("Points");
    points.Columns.Add(new DataColumn("X", typeof(int)));
    points.Columns.Add(new DataColumn("Y", typeof(int)));
    points.Rows.Add(new object[]{1, 2});
    points.Rows.Add(new object[]{3, 5});
    dataGrid1.DataSource = points;

    DataGridTableStyle tableStyle = new DataGridTableStyle();
    tableStyle.MappingName = "Points";
    dataGrid1.TableStyles.Add(tableStyle);
    dataGrid1.TableStyles["Points"].GridColumnStyles["X"].Width = 0;
}
```

Compiling the Code

This example requires:

- A Windows Form with a **DataGrid** control named dataGrid1.

If the data source is a **DataSet** object, set the **DataMember** property of the **DataGrid** to the table's name.

DataTable and **DataColumn** objects in typed datasets also have string-typed names. To find the name of the table, example the table's **Name** property. To find the name of the **DataColumn**, examine the column's **Name** property.

Robust Programming

The following conditions may cause an exception:

- The **MappingName** property does not match the name of the **DataTable** (NullReferenceException Class).
- Retrieving an item from the **GridColumnStyles** collection before the **TableStyle** is added to the **DataGrid.TableStyles** collection. The **GridColumnStyles** collection is filled when the **TableStyle** object is added to the **TableStyles** collection (NullReferenceException Class).
- A **TableStyle** added to the **DataGrid**.**TableStyles** collection does not have a unique **MappingName** (ArgumentException Class).
- The **DataSource** property of the **DataGrid** is not set before accessing the **GridColumnStyles** collection (NullReferenceException Class).

See Also

Windows Forms Example Topics | Hiding a DataColumn in a DataGrid Where the DataSource is an Array | Hiding a DataColumn in a DataGrid Where the DataSource is an ArrayList | DataGrid Control (Windows Forms) | Formatting the Windows Forms DataGrid Control | DataGrid.TableStyles Property | DataGridTableStyle.MappingName Property | DataGridTableStyle.GridColumnStyles Property

Code: Hiding a DataColumn in a DataGrid Where the DataSource is an Array (Visual C#)

This example hides the "X" column of a **DataGrid** that uses an array of **Point** objects as the data source.

Example

```
private void HideColumnOfArray()
{
    System.Drawing.Point [] points = { new Point(1, 2), new Point(3, 5), new Point(5, 6)};
    dataGrid1.DataSource = points;

    DataGridTableStyle tableStyle = new DataGridTableStyle();
    tableStyle.MappingName = "Point[]";
    dataGrid1.TableStyles.Add(tableStyle);
    dataGrid1.TableStyles["Point[]"].GridColumnStyles["X"].Width = 0;
}
```

Compiling the Code

This example requires:

- A Windows Form with a **DataGrid** control named dataGrid1.

The **MappingName**, a string, is the type of the array with "[]" appended. Thus, if the data source is an array of **Point** objects, the mapping name is "Point[]".

Robust Programming

The following conditions may cause an exception:

- The **MappingName** property is not property formatted (NullReferenceException Class).
- Retrieving an item from the **GridColumnStyles** collection before the **TableStyle** is added to the **DataGrid.TableStyles** collection. The **GridColumnStyles** collection is filled when the **TableStyle** object is added to the **TableStyles** collection (NullReferenceException Class).
- A **TableStyle** added to the **DataGrid**.**TableStyles** collection does not have a unique **MappingName** (ArgumentException Class).
- The **DataSource** property of the **DataGrid** is not set before accessing the **GridColumnStyles** collection (NullReferenceException Class).

See Also

Windows Forms Example Topics | Hiding a DataColumn in a DataGrid Where the DataSource is a DataTable | Hiding a DataColumn in a DataGrid Where the DataSource is an ArrayList | DataGrid Control (Windows Forms) | Formatting the Windows Forms DataGrid Control | DataGrid.TableStyles Property | DataGridTableStyle.MappingName Property | DataGridTableStyle.GridColumnStyles Property

Code: Hiding a DataColumn in a DataGrid Where the DataSource is an ArrayList (Visual C#)

This example hides one column of a **DataGrid** that uses an **ArrayList** of **Point** objects as a data source.

Example

```
private void HideColumnOfArrayList()
{
    System.Collections.ArrayList list = new System.Collections.ArrayList();
    list.AddRange( new Point[]{ new Point(1, 2), new Point(3, 5), new Point(5, 6) } );
    dataGrid1.DataSource = list;
    DataGridTableStyle tableStyle = new DataGridTableStyle();
    tableStyle.MappingName = "ArrayList";
    dataGrid1.TableStyles.Add(tableStyle);
    object o = dataGrid1.TableStyles["ArrayList"];
    dataGrid1.TableStyles["ArrayList"].GridColumnStyles["X"].Width = 0;
}
```

Compiling the Code

This example requires:

- A Windows Form with a **DataGrid** control named dataGrid1.

When the data source is an **ArrayList** object, the **MappingName** is "ArrayList".

Robust Programming

The following conditions may cause an exception:

- The **MappingName** property is not "ArrayList"(NullReferenceException Class).
- Retrieving an item from the **GridColumnStyles** collection before the **TableStyle** is added to the **DataGrid.TableStyles** collection. The **GridColumnStyles** collection is filled when the **TableStyle** object is added to the **TableStyles** collection (NullReferenceException Class).
- A **TableStyle** added to the **DataGrid**.**TableStyles** collection does not have a unique **MappingName** (ArgumentException Class).
- All the objects in the **ArrayList** are not of the same type of the first item or a derived type of the first item (TargetInvocationException Class).
- The **DataSource** property of the **DataGrid** is not set before accessing the **GridColumnStyles** collection (NullReferenceException Class).

See Also

Windows Forms Example Topics | Hiding a DataColumn in a DataGrid Where the DataSource is a DataTable | Hiding a DataColumn in a DataGrid Where the DataSource is an Array | DataGrid Control (Windows Forms) | Formatting the Windows Forms DataGrid Control | DataGrid.TableStyles Property | DataGridTableStyle.MappingName Property | DataGridTableStyle.GridColumnStyles Property

Code: Populating a ListBox Control with an Array of Strings (Visual C#)

This example adds an array of strings to a Windows Forms **ListBox** control.

Example

```
private void Form1_Load(object sender, System.EventArgs e)
{
    string [] myList = new string[4];

    myList[0] = "One";
    myList[1] = "Two";
    myList[2] = "Three";
    myList[3] = "Four";

    listBox1.Items.AddRange(myList);
}
```

Compiling the Code

This example requires:

- A form named Form1 with a **ListBox** control named listBox1. Set Form1's **Load** event handler to Form1_Load.

  Note This can also be used with a **ComboBox** control by substituting a **ComboBox** control named comboBox1 for the **ListBox** control and changing the code from listBox1 to comboBox1.

Robust Programming

The following conditions may cause an exception:

- The array contains one or more null values.

See Also

Windows Forms Example Topics

Code: Retrieving an Image that is an Embedded Resource (Visual C#)

This example retrieves an image that is an embedded resource of the assembly.

Example

```
System.Reflection.Assembly thisExe;
thisExe = System.Reflection.Assembly.GetExecutingAssembly();
System.IO.Stream file =
    thisExe.GetManifestResourceStream("AssemblyName.ImageFile.jpg");
this.pictureBox1.Image = Image.FromStream(file);
```

Compiling the Code

This example requires:

- A Windows Form with a **PictureBox** control named pictureBox1.

Add the image file to the project, and set the **Build Action** property to **Embedded Resource** in Solution Explorer.

Replace `"AssemblyName.ImageFile.jpg"` with the name of the resource, as it is known in the assembly. Use the **GetManifestResourceNames** method of the Assembly object to find the resource name. See Code: Finding the Names of Resources in an Assembly.

Robust Programming

The following conditions may cause an exception:

- The embedded resource does not exist in the assembly, and the call to **GetManifestResourceStream** returns **Nothing**.
- There may not be an application associated with the file type (the file extension).

See Also

Windows Forms Example Topics | Code: Finding the Names of Resources in an Assembly | Code: Running a Program | Process Class

Code: Retrieving the Data from the Selected Cell in a Windows DataGrid Control (Visual C#)

This example retrieves the selected data in an existing **DataGrid** control filled with integer values.

Example

```
System.Windows.Forms.DataGridCell selectedCell = dataGrid1.CurrentCell;
object selectedItem = dataGrid1[selectedCell.RowNumber, selectedCell.ColumnNumber];
int cellValue = Convert.ToInt32(selectedItem);
```

Compiling the Code

This example requires:

- A **DataGrid** control named dataGrid1.

Robust Programming

If the **DataGrid** is not bound to a data source, or if no cell is selected, then the selected cell is the cell in row 0 and column 0.

The following conditions may cause an exception:

- The **DataGrid** control is not bound to a data source (InvalidOperationException Class).
- The data type of the cell does not contain the type of data to which you are trying to convert (InvalidCastException Class).

See Also

Windows Forms Example Topics | Code: Determining the Selected Cell in a Windows DataGrid Control | DataGrid.CurrentCell Property | DataGrid Control (Windows Forms)

Code: Searching for an Item in a ListBox Control (Visual C#)

This example searches for a string in a Windows Forms **ListBox** control.

Example

```
private void button1_Click(object sender, System.EventArgs e)
{
    listBox1.Items.Add("One");
    listBox1.Items.Add("Two");
    listBox1.Items.Add("Three");

    if (listBox1.FindString("Two") != -1)
        MessageBox.Show("Found it!");
}
```

Compiling the Code

This example requires:

- A form with a **ListBox** control named listBox1 and a **Button** control named button1. Set button1's **Click** event handler to button1_Click.

  Note This can also be used with a **ComboBox** control by substituting a **ComboBox** control named comboBox1 for the **ListBox** control and changing the code from listBox1 to comboBox1.

See Also

Windows Forms Example Topics

Code: Selecting a Range of Dates in a Calendar Control (Visual C#)

This example selects a range of dates in a Windows Forms **MonthCalendar** control. In this example, when the user selects a date the week is selected.

Example

```
private void monthCalendar1_DateSelected(object sender, System.Windows.Forms.DateRangeEventAr
gs e)
{
    DateTime startDate = e.Start;
    startDate = startDate.AddDays(-(double)startDate.DayOfWeek);
    monthCalendar1.SelectionStart = startDate;
    monthCalendar1.SelectionEnd = startDate.AddDays(6);
}
```

Compiling the Code

This example requires:

- A Windows Form with a **MonthCalendar** control named monthCalendar1. Set monthCalendar1's **DateSelected** event handler to monthCalendar1_DateSelected.

See Also

Windows Forms Example Topics | MonthCalendar Control (Windows Forms)

Code: Selecting an Item in a ListBox Control (Visual C#)

This example selects and highlights an item in a Windows Forms **ListBox** control.

Example

```
private void button1_Click(object sender, System.EventArgs e)
{
    listBox1.Items.Add("One");
    listBox1.Items.Add("Two");
    listBox1.Items.Add("Three");
    listBox1.SelectedIndex = listBox1.FindString("Two");
}
```

Compiling the Code

This example requires:

- A form with a **ListBox** control named listBox1 and a **Button** control named button1. Set button1's **Click** event handler to button1_Click.

  Note This can also be used with a **ComboBox** control by substituting a **ComboBox** control named comboBox1 for the **ListBox** control and changing the code from listBox1 to comboBox1.

See Also

Windows Forms Example Topics

Visual Basic Sample Applications

You can access sample code in two ways:

- Browse the sample abstracts in this section. Each abstract contains a link to open or copy the sample's files.
- Browse the Visual Basic samples directory structure on the product CD or DVD. For the specific location of sample files for your version of Visual Basic .NET, see Locating Sample Files.

In the Visual Basic documentation, "sample" means a complete application, including project files, while "example" means a small block of code that demonstrates a single, limited programming task.

In This Section

Visual Basic Sample Abstracts
Provides links to sample abstracts. Each abstract contains a link to open or copy the sample's files.
Visual Basic QuickStarts and Other Samples
Lists other Visual Basic samples available when you install the .NET Framework SDK.
Code for Visual Basic Programmers
Lists sample code available in Visual Basic code example topics.

Related Sections

QuickStarts (.NET Framework)
Includes over 300 samples designed to introduce you to various aspects of the .NET Framework.

Samples	Visual Basic Samples
Samples	
Samples	Visual Basic Concepts
Samples	
Samples	
Samples	Writing C# Applications
Samples	
Samples	
Visual C# Samples	C# Sample Applications
Visual C# Samples	

Visual Basic Sample Abstracts

In This Section

Console: Localized Hello World Sample
Demonstrates localizing your applications for different cultures.
Web Forms: DHTML Sample
Demonstrates that Dynamic HTML (DHTML) is still a useful tool in the world of .NET programming.
Web Forms: User Control Sample
Demonstrates how to build a custom ASP.NET User Control in Visual Basic .NET and access it from a Web Forms page.
XML Web Services: Windows Forms Client/Asynchronous XML Web Service Sample
Demonstrates how to call an XML Web service asynchronously.
Windows Forms: Calc Sample
Demonstrates how to create the equivalent of a control array in Visual Basic .NET by using common methods to handle the individual controls' click events.
Windows Forms: Localization Sample
Demonstrates how to save culture preferences to your computer that persist from run to run of your application.
Windows Forms: Multi-Threaded TCP/IP Listener Sample
Demonstrates a technique for creating a multi-threaded application that generates and listens for TCP/IP traffic.
Windows Forms: Multilingual Form Sample
Demonstrates using different culture information in a Windows Forms application.
Windows Forms: Performance Monitor Sample
Demonstrates how to monitor system performance using performance counters. This sample is also an introduction to performance counters.
Windows Forms: Process Controller Sample
Demonstrates how to build a Visual Basic Windows application for processes and services monitoring.
Windows Forms: Registry Client Sample
Demonstrates how to use the Registry classes to add a new registry key and manipulate the values associated with that key.
Windows Forms: Scribble Sample
Demonstrates a Windows Forms MDI drawing application.
Windows Forms: System Tray Sample
Demonstrates how to use the status area of the Windows taskbar in a Visual Basic .NET application.
Windows Forms: Vehicle Sample
Demonstrates inheritance by implementing a vehicle class for cars and boats.
Windows Forms: Weather Sample
Demonstrates how to use the .NET Framework's XML classes to query an XML document and retrieve the weather forecast for a specified city.
Windows Forms: Winlogger Sample
Demonstrates using a logging component to write to a text file and the Windows Application Event Log.

Related Sections

Visual Basic Sample Applications
Provides links to Visual Basic code samples that demonstrate .NET concepts.
Visual Basic Walkthroughs
Provides a list of topics that give step-by-step instruction on doing various kinds of Visual Basic programming.
Decision Chart
Provides a graphical, linked guide to decisions about designing your application.

Console: Localized Hello World Sample

The Localized Hello World sample demonstrates creating and using satellite assemblies for console applications. It shows how to create resources for Visual Basic console applications, localize the resources, and build localized satellite assemblies.

> **Note** To install and run this sample, you must be a member of your computer's Debugger Users group. If installing this application by means of a remote session, you must be a member of the Administrators group.

To run this sample

1. Copy all files to your local hard drive by means of the **Copy All Files** link.
2. Open the solution **LocalizedHelloWorld.sln**.
3. From the **Build** menu, click **Build Solution**.
4. From the **Debug** menu, click **Start Without Debugging**.
5. Type in a culture supported by the application to see the resources in that culture. This sample supports English (en), German (de), Spanish (es) French (fr) and Italian (it).

See Also

Visual Basic Sample Applications | Visual Basic Walkthroughs | Security and Localized Satellite Assemblies | LocalizedHelloWorld Sample: Demonstrates Satellite Assemblies for Console Applications (Visual C#)

Web Forms: DHTML Sample

The Web Forms DHTML sample demonstrates using client-side script and Dynamic HTML (DHTML) within a Web Forms application. A majority of the Web Forms samples available focus on the new server-side event-handling capabilities available with ASP.NET. This sample demonstrates that client-side script is still a viable and useful tool for building Web applications in ASP.NET.

> **Note** To install and run this sample, you must be a member of your computer's Debugger Users and VS Developers groups. To debug this sample, you must be a member of the Administrators group. If installing this application by means of a remote session, you must be a member of the Administrators group.

To run this sample

1. Use the **Copy** Link to copy the files to your hard drive. This sample will not function properly if the **Load** link is used.
2. Run the **WebFormDHTMLSetup.msi** file and follow the directions.
3. Open the **.sln** file.

   > **Note** If Visual Studio cannot open the project by means of Front Page Server Extensions, a warning dialog box will be displayed asking you to switch to file-based access. Choose **OK** to continue opening the project.

4. Press F5 to run the sample.

   > **Note** If you are prompted to set the Start Page for the application, right-click the **WebFormDHTMLDemo.aspx** item in Solution Explorer and select **Set As Start Page**.

See Also

Visual Basic Sample Applications | Visual Basic Walkthroughs | Client Script in Web Forms Pages

Web Forms: User Control Sample

The Web Forms user control sample demonstrates how to build a custom ASP.NET User Control in Visual Basic .NET and access it from a Web Forms page. This sample includes a custom paired list box user control that allows the user to move values between the available and selected list boxes. The Web Forms page that hosts the control can specify and retrieve the values from the available and selected lists.

This sample features Web Forms pages and user controls.

> **Note** This sample will not run on Windows NT 4.0; you must be running Windows 2000 or Windows XP. To install and run this sample, you must be a member of your computer's Debugger Users and VS Developers groups. To debug this sample, you must be a member of the Administrators group. If installing this application by means of a remote session, you must be a member of the Administrators group.

To run this sample

1. Use the **Copy** Link to copy the files to your hard drive.
2. Double-click the **Setup.msi** icon, and follow the steps to install the sample on your computer.
3. Open the .sln file.
4. Set a start page by right-clicking WebForm1.aspx in Solution Explorer and choosing **Set As Start Page** on the shortcut menu.
5. Press F5.

See Also

Visual Basic Sample Applications | Visual Basic Walkthroughs

XML Web Services: Windows Forms Client/Asynchronous XML Web Service Sample

The Asynchronous XML Web service sample demonstrates how to call an XML Web service from a Visual Basic .NET Windows Form client application by using the built-in asynchronous support of the .NET Framework. The sample XML Web service takes an input parameter — milliseconds to delay before returning to the client — and returns a random number to the caller. The Windows Form client application allows the user to specify the milliseconds to delay and demonstrates calling the XML Web service synchronously and asynchronously.

> **Note** To install and run this sample, you must be a member of your computer's Debugger Users and VS Developers groups. To debug this sample, you must be a member of the Administrators group. If installing this application by means of a remote session, you must be a member of the Administrators group.

To run this sample

1. Use the **Copy** Link to copy the files to your hard drive. This sample will not function properly if the **Load** link is used.
2. Run the **AsyncWebServiceSetup.msi** and follow the directions.
3. After the installation completes, launch **AsyncWebService.sln**.

   > **Note** If Visual Studio cannot open the project by means of Front Page Server Extensions, a warning dialog box will be displayed asking you to switch to file-based access. Choose **OK** to continue opening the project.

4. In **Solution Explorer**, right-click **AsyncTestClient** and choose **Set as Startup Project**.
5. Press F5 to run the sample.
6. Click the **Call Web Service Sync** button.
7. Click the **Call Web Service Async** button. Do not clear to the 'Async Call Started' message in order to see the asynchronous behavior of the call.

See Also

Visual Basic Sample Applications | Visual Basic Walkthroughs | XML Web Services in Managed Code

Windows Forms Sample Abstracts for Visual Basic

The samples listed on this page demonstrate features that can be used in Windows-based applications.

In This Section

Windows Forms: Calc Sample
Demonstrates how to create the equivalent of a control array in Visual Basic .NET by using common methods to handle the individual controls' click events.
Windows Forms: Localization Sample
Demonstrates how to save culture preferences to your computer that persist from run to run of your application.
Windows Forms: Multi-Threaded TCP/IP Listener Sample
Demonstrates a technique for creating a multi-threaded application that generates and listens for TCP/IP traffic.
Windows Forms: Multilingual Form Sample
Demonstrates using different culture information in a Windows Forms application.
Windows Forms: Performance Monitor Sample
Demonstrates how to monitor system performance using performance counters. This sample is also an introduction to performance counters.
Windows Forms: Process Controller Sample
Demonstrates how to build a Visual Basic Windows application for processes and services monitoring.
Windows Forms: Registry Client Sample
Demonstrates how to use the Registry classes to add a new registry key and manipulate the values associated with that key.
Windows Forms: Scribble Sample
Demonstrates a Windows Forms MDI drawing application.
Windows Forms: System Tray Sample
Demonstrates how to use the status area of the Windows taskbar in a Visual Basic .NET application.
Windows Forms: Vehicle Sample
Demonstrates inheritance by implementing a vehicle class for cars and boats.
Windows Forms: Weather Sample
Demonstrates how to use the .NET Framework's XML classes to query an XML document and retrieve the weather forecast for a specified city.
Windows Forms: Winlogger Sample
Demonstrates using a logging component to write to a text file and the Windows Application Event Log.

Related Sections

Visual Basic Sample Applications
Provides links to Visual Basic code samples that demonstrate .NET concepts.
Visual Basic Walkthroughs
Provides a list of topics that give step-by-step instruction on doing various kinds of Visual Basic programming.
Decision Chart
Provides a graphical, linked guide to decisions about designing your application.

Windows Forms: Calc Sample

The Calc sample demonstrates how to create the equivalent of a control array in Visual Basic .NET by using common methods to handle the individual controls' click events.

This sample features Windows Forms and control arrays.

> **Note** To install and run this sample, you must be a member of your computer's Debugger Users group. If installing this application by means of a remote session, you must be a member of the Administrators group.

To run this sample

1. Copy all files to your local hard drive by means of the **Copy All Files** link.
2. Open the .sln file.
3. Press F5.

See Also

Visual Basic Sample Applications | Visual Basic Walkthroughs

Windows Forms: Localization Sample

The Windows Forms Localization sample demonstrates techniques for localizing a Windows Forms application with Visual Basic .NET. The sample contains a variety of controls and localized content (text string, menus, images, and so forth.). When the user selects a language from the combo box, the user interface will update itself using the locale of the language selected.

> **Note** To install and run this sample, you must be a member of your computer's Debugger Users group. If installing this application by means of a remote session, you must be a member of the Administrators group.

To run this sample

1. Copy all files to your local hard drive by means of the **Copy All Files** link.
2. Open the .sln file.
3. Press F5 to start the application.
4. Click the **Click Me** button.

   A greeting is displayed in English.

5. In the combo box, choose a different culture.

   The text of the application and the greeting displayed are now represented in the selected language.

See Also

Visual Basic Sample Applications | Visual Basic Walkthroughs | Localizing Applications

Windows Forms: Multilingual Form Sample

The Multilingual Form sample demonstrates creating and localizing Windows Forms. It shows how to use the Windows Forms Designer to create, edit, and build localized applications.

For more information about localized applications, see Globalizing and Localizing Applications.

> **Note** To install and run this sample, you must be a member of your computer's Debugger Users group. If installing this application by means of a remote session, you must be a member of the Administrators group.

To run this sample

1. Copy all files to your local hard drive by means of the **Copy All Files** link.
2. Open the solution **VBMultilingualForm.sln**.
3. On the **Build** menu, click **Build Solution**.
4. On the **Debug** menu, click **Start Without Debugging**.
5. To see a different culture's resources, type a valid RFC 1766 culture name into the text box and click the **switch** button. A list of available cultures will appear when the mouse pointer hovers over the text box.

To localize your own Windows Forms or Web Forms

1. Select the form to be localized in the appropriate forms designer.
2. Set the form's **Localizable** property to **true**.
3. For each language you want to localize to, do the following:
   a. Set the form's **Language** property to the desired language.
   b. Enter the new text in the new language for each control.

You can also change the appearance of the form's controls (size, position, and so on) for each language.

See Also

Visual Basic Sample Applications | Visual Basic Walkthroughs |
MultilingualForm Sample: Demonstrates Creating and Localizing Windows Forms (Visual C#)

Windows Forms: Multi-Threaded TCP/IP Listener Sample

The MultiThreaded TCP/IP Listener sample demonstrates a technique for creating a multi-threaded application that generates and listens for TCP/IP traffic. The TCPIPClient application submits multiple TCP/IP calls to the listener component. The TCPIPListener application creates threads that listen for TCP/IP requests. This sample shows how to use critical sync locks to update a single counter on the server.

This sample features Windows Forms, the .NET Framework, and threading.

> **Note** To install and run this sample, you must be a member of your computer's Debugger Users group. If installing this application by means of a remote session, you must be a member of the Administrators group.

To run this sample

1. Copy all files to your local hard drive by means of the **Copy All Files** link.
2. Open the .sln file.
3. In Solution Explorer, right-click the **TCPIPListener** solution and select **Properties**.

   The Solution Property Pages dialog box is displayed.

4. In the Startup Project page, click **Multiple Startup Projects**.
5. Set the Action to **Start without debugging** for both the Client and Listener projects.
6. Click **OK** to close the dialog box.
7. Press CTRL+F5.
8. Click **Start** in the Client form.

See Also

Visual Basic Sample Applications | Visual Basic Walkthroughs | Multithreading in Components

Windows Forms: Performance Monitor Sample

The PerfMon sample is an introduction to performance counters. It demonstrates how to monitor local or remote system performance. It shows how to dynamically create timers, context menus, and graphic displays. It also demonstrates the following activities:

- Including new Visual Basic classes derived from other components.
- Dynamically adding and removing event handlers and components.
- Using the **TreeView** control, **TreeNode** class, and **Timer** component.
- Using the **PerformanceCounterCategory** and **PerformanceCounter** components.

  For more information on performance counters, see PerformanceCounter Class.

  > **Note** This sample is functional only on Windows NT 4.0, Windows 2000, and Windows XP. To install and run this sample, you must be a member of your computer's Debugger Users group. If installing this application by means of a remote session, you must be a member of the Administrators group.

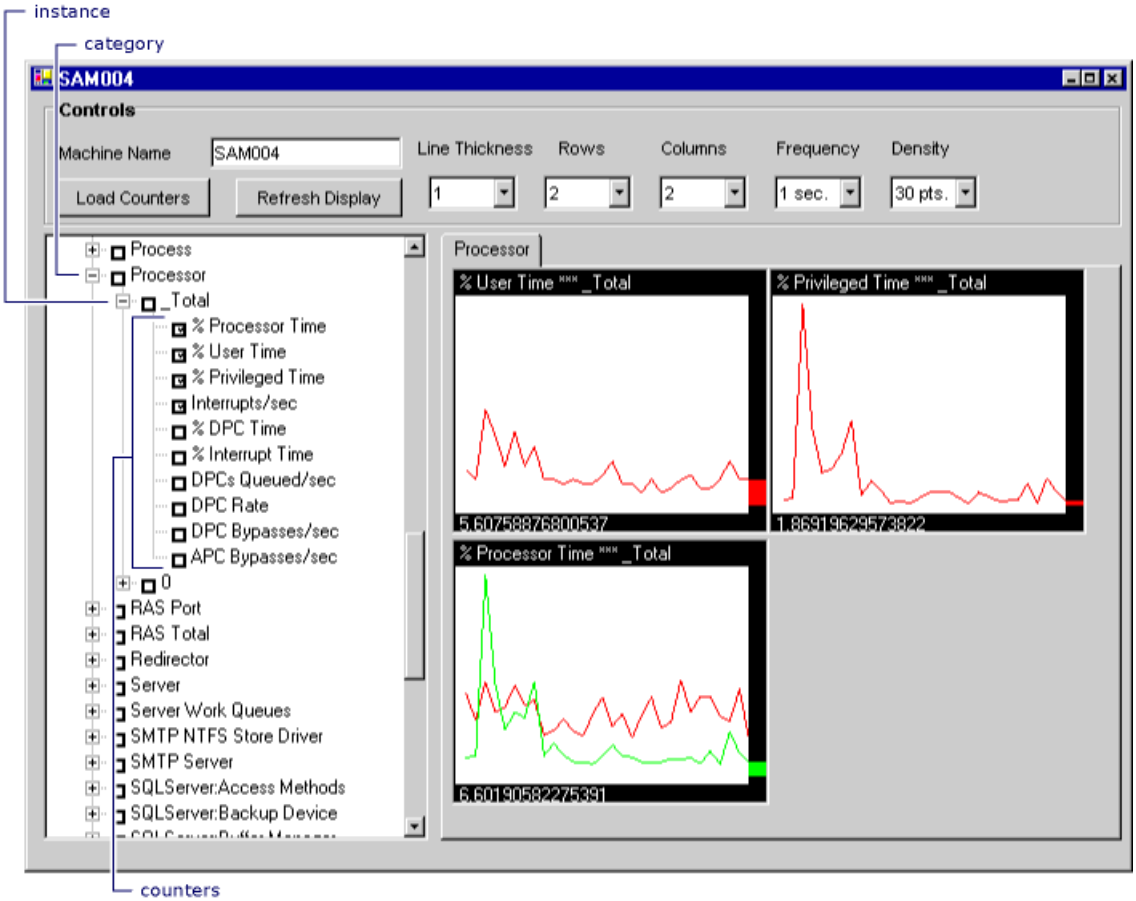
Building and Running the Sample

To build and run this sample

1. Copy all files to your local hard drive by means of the **Copy All Files** link.
2. Open the solution **PerfMon.sln**.
3. On the **Build** menu, click **Build**.
4. On the **Debug** menu, click **Start Without Debugging**.

   The main application window appears.

The following figure shows the performance monitor program running. Refer to the figure, as needed, while you go through the various performance monitoring procedures.



The Display Panel Properties

Each graphic display in the figure shows the following:

- The top of the display shows the name of the counter and the name of the instance separated by "***". If the category does not contain an instance name (for example, the ICMP category), the "***" is followed by a blank space.
- The bottom of the display shows the current counter value. This value may change periodically (the default is every second).
- The right side of the display shows a red bar. The height of the bar depends on the last value monitored on the counter.
- The graph plotted in the display shows the performance history for that counter.

This application is an MDI application. It starts with four windows that can be used to monitor four computers. You can open as many windows as you want by clicking the **New PerfMon** command on the **Options** menu. In the following procedures, only one window is used. Therefore, you can close all of the windows except one.

To load the performance counters of a system

1. In the **Machine Name** field, type the name of the system you want to monitor. If you do not type anything, the local computer will be monitored.
2. Click the **Load Counters** button (or press ALT+L). The application will load **PerformanceCounterCategory**, **InstanceName**, and **PerformanceCounter** components on the specified computer. The **Load Counters** button will be disabled until all of the counters on the system have been loaded.

   The tree view in the left pane will contain all of the performance categories and the performance counters available. Each leaf in the tree represents a performance counter. You can monitor the completion progress by watching the progress indicator on the bottom of the application window. After the tree has been completed, the **Load Counters** button will be enabled again.

   > **Note** To monitor a specific computer, you need administrator rights to the computer; otherwise, the **PerformanceCounterCategory** components will not be able to load the categories (security protection).

To monitor the performance of a system

1. Load the performance counters by performing the preceding steps.
2. Select one of the counters by expanding the corresponding category node. For example, to watch the performance of the processor, expand the **Processor** node, then select one of the **Processor** instances (for example, "**_Total**" or "**0**"). Expand the instance node as well to see the counters.
3. On the leaf level, the counters are represented by check boxes. Select one of the counters, for example, **%Processor Time**. A tab, which has the name **Processor**, will appear in the right pane, and a graphic display of the selected counter will appear on the tab page.
4. To remove a specific graphic display from the tab page, clear the check box for the corresponding counter. When all of the counters in a specific page are cleared, the corresponding tab disappears.

   > **Note** Each time you change any of the display properties, click the **Refresh Display** button to apply the changes. The changes will apply for current or future displays.

To manage the display properties

- To change the line thickness, select the thickness you want from the **Line Thickness** box.
- To change the number of rows per display, select the number you want from the **Rows** box. For example, selecting one row makes the display occupy the full height of the tab page.
- To change the number of columns per display, select the number you want from the **Columns** box. For example, selecting one column makes the display occupy the full width of the tab page.

  > **Note** If the number of graphic displays exceeds the visible area of the tab page, scroll bars will be created automatically.

- To change the time interval, select the number of seconds from the **Frequency** box.
- To change the number of points to display in the graph, select a number from the **Density** box.

To merge two or more displays

1. Select the display you want to move and drag it to the destination display. Notice that the background of the destination display becomes black when it is selected and is ready to receive the moved display. Both displays will merge, and two different graphic colors will appear (red and green in this case). For example, if you drop "**%Processor Time *** _total**" on "**%User Time *** _total**", the green color will represent the first and the red color will represent the second. The data shown in the display's top, bottom, and right side will only represent the **"%Processor Time *** _total**" counter.

2. To switch from one graph to another after a display merge, right-click the display and select the name of the graph from the shortcut menu. In the preceding example, if you want to change the display data from "**% Processor Time *** _total**" to "**%User Time *** _total**", right-click the display and select **"%User Time *** _total**" from the shortcut menu. The display will show the data of the selected counter, and the bar color on the right side will change to red (or the color representing the selected counter).
3. If you have more than two merged counters, each counter will be represented by a different color. To select the data view for any one of the counters, perform the preceding step.

To separate two or more merged displays

1. Locate the counter's check box of the display you want to separate.
2. Clear the counter's check box. The corresponding display will be removed.
3. Select the counter's check box again. A new display will appear on the tab page.

To use menu options

- To stop the graphics plotting:

  1. Click the **Option** menu (or press ALT+O).
  2. Click Stop Graphics.

     The graphics will freeze and no more lines will be plotted.

- To restart the graphics plotting:

  1. Click the **Option** menu.
  2. Click Start Graphics.

     The displays will continue plotting from the last position.

- To quit the application:

  1. Click the **Option** menu.
  2. Click **Exit**.

See Also

Visual Basic Sample Applications | Visual Basic Walkthroughs | Introduction to Monitoring Performance Thresholds | PerfMon Sample: Demonstrates How to Monitor System Performance Using Performance Counters (Visual C#)

Windows Forms: Process Controller Sample

The ProcessController sample shows how to build a Visual Basic Windows application for monitoring processes and services. The sample also demonstrates the following activities:

- Dynamically adding and removing event handlers.
- Using the **Process** component to:
  - Read the running processes on the local/remote computer.
  - Start a process on the local computer.
  - Stop a process on the local/remote computer.
  - Read the modules used by some running process.
  - Read all of the characteristics of some running process.
- Using the **ServiceController** component to:
  - Read the running services and device drivers on the local/remote computer.
  - Start/Stop/Pause a service on the local/remote computer.
  - Check properties of the services or device drivers on the local/remote computer.
- Handling security issues related to the use of the **Process** and the **ServiceController** components.

> **Note** To install and run this sample, you must be a member of your computer's Debugger Users group. If installing this application by means of a remote session, you must be a member of the Administrators group.

Building and Running the Sample

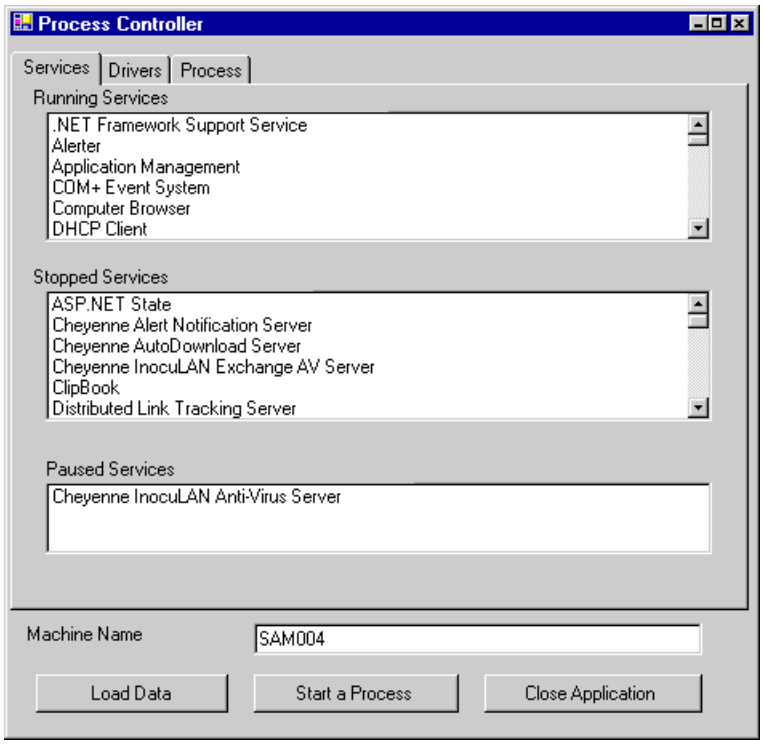
To build and run this sample

1. Copy all files to your local hard drive by means of the **Copy All Files** link.
2. Open the solution **ProcessController.sln**.
3. On the **Build** menu, click **Build**.
4. On the **Debug** menu, click **Start Without Debugging**.

   The main application window appears.

The following figure shows the ProcessController program running. Refer to the figure, as needed, while you perform the following test procedures:

- Load all services, device drivers, and processes running on some computer in the network.
- Start/Pause/Stop a service or a device driver running on the local/remote computer.
- Start a process running on the local computer.
- Terminate a process running on the local/remote computer.



To load all of the services, device drivers, and processes

1. Type the name of the computer you want in the **Machine Name** box.
2. Click the **Load Data** button (or press ALT+L).

To start a service

1. Click a service name in the **Stopped Services** list.
2. Right-click the service and click **Start Service** on the shortcut menu.

   After the service starts, its name should appear in the top-most list labeled **Running Services**.

   > **Note** You cannot start a service if you do not have the administrator rights on that computer.

To stop a service

1. Click a service name in the **Running Services** list.
2. Right-click the service and click **Stop Service** on the shortcut menu.

   After the service has been stopped, the name should appear in the **Stopped Services** list.

   > **Note** Some services do not accept the **Stopped** state. Also, you cannot stop a service if you do not have administrator rights on that computer.

To pause a running service

- Click a service name in the **Running Services** list.
- Right-click the service and click **Pause** on the shortcut menu.

  If the service is paused, the service name should appear in the **Paused Services** list at the bottom of the window.

  > **Note** Some services do not accept the **Paused** state. Also, you cannot pause a service if you do not have administrator rights on that computer.

To check some of the service's information

1. Click a service name in any of the three lists.
2. Right-click the service and click **Show Service Info** from the shortcut menu. A new window containing the service information will appear.

To verify and manage device drivers

1. Click the **Drivers** tab at the top of the window.
2. Go through the preceding steps to start, stop, and pause a service. Exactly the same rules mentioned for services apply to the device drivers.

To verify and manage processes

1. Click the **Process** tab at the top of the window.
2. Select a process name in the left-most list. At this point, information about that service should appear in the list on the right.

   Note If you are not an administrator on the target computer, you will not be able to see or manage any process.

To start a process

1. Click the **Start a Process** button (or press ALT+S).
2. Select the process you want to start in the **Open** dialog box.
3. Click the **Open** button in the dialog box. If your target computer is the local computer, the process name should appear in the left-most list.

   Note If the target computer is a remote computer, the process will start on the local computer, by default, if possible.

To stop a process

1. Click a process in the left-most list.
2. Right-click the process and click **Terminate Process** from the shortcut menu.

   After the process has been terminated, its name should disappear from the list.

To load the services, drivers, and processes from another computer

1. Type the name of the computer in the **Machine Name** box.
2. Click the **Load Data** button (or press ALT+L).

To stop the ProcessController application

- Click the **Close Application** button (or press ALT+C).

  After you click the button, the window will slowly fade away. This effect is obtained by handling the Window's Opacity property (file: MainForm.vb, method: cmdClose_Click).

Description of the Sample Classes

File	Description
MainForm.vb	Describes the MainForm class, which is a System.Windows.Forms.Form type class. Its designer contains most of the UI controls needed for this application.
ProcessControllerManager.vb	Contains the implementation of the ProcessControllerManager class. This class handles processes management.
DriverControllerManager.vb	Contains the implementation of the DriverControllerManager class. This class is in charge of managing the device drivers on a given computer.
ServiceControllerManager.vb	Contains the implementation of the ServiceControllerManager class. This class describes the logic to control the services on a given computer.
ServiceInfo.vb	Implements the ServiceInfo class, which is a System.Windows.Forms.Form class and is used to display services and drivers information only.

See Also

Visual Basic Sample Applications | Visual Basic Walkthroughs | Introduction to Monitoring and Managing Windows Processes | ProcessController Sample: Demonstrates Processes and Services Monitoring (Visual C#)

Windows Forms: Registry Client Sample

The Registry Client sample demonstrates the use of the Registry classes. It demonstrates how you can use these classes to add a new registry key and manipulate the values associated with that key.

This sample features Windows Forms, the .NET Framework, and the Windows registry.

> **Note** To install and run this sample, you must be a member of your computer's Debugger Users group. If installing this application by means of a remote session, you must be a member of the Administrators group.

To run this sample

1. Copy all files to your local hard drive by means of the **Copy All Files** link.
2. Open the .sln file.
3. Press F5.

See Also

Visual Basic Sample Applications | Visual Basic Walkthroughs

Windows Forms: Scribble Sample

The Scribble sample shows how to develop a Windows Forms MDI application using Visual Basic and the .NET Framework classes. Scribble is a small drawing application that lets you draw free-hand drawings using the mouse and save the images in a file.

> **Note** To install and run this sample, you must be a member of your computer's Debugger Users group. If installing this application by means of a remote session, you must be a member of the Administrators group.

To run this sample

1. Copy all files to your local hard drive by means of the **Copy All Files** link.
2. Open the solution **Scribble.sln**.
3. On the **Build** menu, click **Build**.
4. On the **Debug** menu, click **Start Without Debugging**.

When you run the sample, you get a drawing surface and the following menus:

- **File** menu for input and output operations.
- **Edit** menu to cut, copy, and paste.
- **Pen** menu to select the pen thickness.
- **View** menu to show or hide the toolbar and the status bar.
- **Window** menu to create, tile, or cascade windows.
- **Help** menu to display the help topics and the About Scribble text box.

In addition to the menu options, the user interface includes a toolbar with buttons to duplicate the menu commands: **New**, **Open**, **Save**, **Print Preview**, **Print**, and **Help**.

See Also

Visual Basic Sample Applications | Visual Basic Walkthroughs | Multiple-Document Interface (MDI) Applications | Scribble Sample: Visual C# MDI Drawing Application

Windows Forms: System Tray Sample

The System Tray sample demonstrates how to use the status area of the Windows taskbar in a Visual Basic .NET application. When run, the application adds a new icon to your system tray. Right-clicking the icon will display a shortcut menu with options to display a settings form, toggle the image in the tray, or quit the application.

This sample features Windows Forms, the status area of the taskbar, user controls, and context menus.

> **Note** To install and run this sample, you must be a member of your computer's Debugger Users group. If installing this application by means of a remote session, you must be a member of the Administrators group.

To run this sample

1. Copy all files to your local hard drive by means of the **Copy All Files** link.
2. Open the .sln file.
3. Press F5.

See Also

Visual Basic Sample Applications | Visual Basic Walkthroughs

Windows Forms: Vehicle Sample

The Vehicle sample demonstrates inheritance by implementing a vehicle class for cars and boats. The boat class overrides the base **ComputeRange** method to double the range for sailboats.

This sample features Windows Forms, language changes, and inheritance.

> **Note** To install and run this sample, you must be a member of your computer's Debugger Users group. If installing this application by means of a remote session, you must be a member of the Administrators group.

To run this sample

1. Copy all files to your local hard drive by means of the **Copy All Files** link.
2. Open the .sln file.
3. Press F5.

   > **Note** If prompted that the project cannot be started directly, right-click **VehicleSampleBaseClient** in Solution Explorer and choose **Set As Startup Project**. Then press F5 again.

4. Choose a vehicle from the drop down list.
5. Enter Mpg and capacity information and press the **Compute Range** button.

See Also

Visual Basic Sample Applications | Visual Basic Walkthroughs | Inheritance Basics

Windows Forms: Weather Sample

The Weather sample uses the .NET Framework's XML classes to query an XML document and retrieve the weather forecast for a specified city. This sample uses the **XML** and **XMLDocumentNavigator** classes to open an XML document and issue an XPath query to find the weather for the selected city. The forecast is then displayed in a message box.

This sample features Windows Forms, the .NET Framework, and XML.

> **Note** To install and run this sample, you must be a member of your computer's Debugger Users group. If installing this application by means of a remote session, you must be a member of the Administrators group.

To run this sample

1. Copy all files to your local hard drive by means of the **Copy All Files** link.
2. Open the .sln file.
3. Press F5.

See Also

Visual Basic Sample Applications | Visual Basic Walkthroughs

Windows Forms: Winlogger Sample

The WinLogger sample contains a logging component that supports writing to a text file and the Windows Application Event Log. The logging component contains two classes that each implement the custom ILog interface. The client application prompts the user for a message and then provides the ability to call either logging interface.

This sample features Windows Forms, .NET Framework, file I/O, event log, and component.

> **Note** To install and run this sample, you must be a member of your computer's Debugger Users group. If installing this application by means of a remote session, you must be a member of the Administrators group.

To run this sample

1. Copy all files to your local hard drive by means of the **Copy All Files** link.
2. Open the .sln file.
3. Press F5.

   > **Note** If prompted that the project cannot be started directly, right-click **Winlogger** in Solution Explorer and choose **Set As Startup Project**. Then press F5 again.

4. Enter a message you wish to log.
5. Click the **Write to file** button and check the \WinLogger\Bin\LogFile.txt file.
6. Click the **Write to event log** button and check the Application LogStep1.

See Also

Visual Basic Sample Applications | Visual Basic Walkthroughs | Introduction to the EventLog Component

Visual Basic QuickStarts and Other Samples

You can find Visual Basic and other general samples at Program Files\Microsoft Visual Studio .NET\FrameworkSDK\Samples if you install them with the .NET Framework SDK. If you do not want to install the samples on your computer, go to the StartSamples.htm file in the ...\Samples folder in the .NET Framework SDK installation directory on the product CD or DVD.

QuickStarts

The QuickStarts are a series of samples and supporting documentation designed to quickly acquaint you with the syntax, architecture, and power of Visual Studio .NET and the .NET Framework. To access the QuickStarts, open the Start menu, then click Programs, Microsoft .NET Framework SDK, and then Samples and QuickStart Tutorials. The main page of the "Microsoft .NET Framework SDK QuickStarts, Tutorials and Samples" application will appear. To run the samples, follow the directions to install the samples database and complete the installation. After you have installed the QuickStarts, the "QuickStarts Tutorials" link will be functional.

The following QuickStarts are specific to Visual Basic:

Windows Forms

Description	Location
Hello World	quickstart\winforms\samples\simplehelloworld\vb\SimpleHelloWorld.vbproj quickstart\winforms\samples\designablehelloworld\vb\DesignableHelloWorld.vbproj
The Windows Forms Model	quickstart\winforms\samples\helloworldform\vb\HelloWorldForm.vbproj
Dynamic layout: anchoring and docking	quickstart\winforms\samples\layout\anchoring\vb\Anchoring.vbproj quickstart\winforms\samples\layout\docking\vb\Docking.vbproj quickstart\winforms\samples\layout\dockman\vb\DockMan.vbproj
MDI applications	quickstart\winforms\samples\mdi\vb\MDI.vbproj
Handling errors	quickstart\howto\samples\winforms\errorhandler\vb\ErrorHandler.vbproj

Window Forms Controls

Description	Location
Button	quickstart\winforms\samples\controlreference\buttonctl\vb\ButtonCtl.vbproj
CheckedListBox	quickstart\winforms\samples\controlreference\checkedlistboxctl\vb\CheckedListBoxCtl.vbproj
ComboBox	quickstart\winforms\samples\controlreference\comboboxctl\vb\ComboBoxCtl.vbproj
MainMenu, ContextMenu	quickstart\winforms\samples\menus\vb\Menus.vbproj
DateTimePicker, ColorDialog, FontDialog, GroupBox	quickstart\winforms\samples\controlreference\datetimepickerctl\vb\DateTimePickerCtl.vbproj
DomainUpDown and Numeric UpDown	quickstart\winforms\samples\controlreference\updownctl\vb\UpDownCtl.vbproj
DataGrid	quickstart\winforms\samples\data\grid\vb\Grid.vbproj
ErrorProvider	quickstart\winforms\samples\data\update\vb\UpdateWebService.vbproj quickstart\winforms\samples\data\update\vb\client\Update.vbproj quickstart\winforms\samples\data\update\vb\customersdataset\CustomersDataSet.vbproj

PageSetupDialog, PrintDialog, PrintDocument, PrintPreviewControl, PrintPreviewDialog	quickstart\winforms\samples\printing\example1\vb\PrintingExample1.vbproj quickstart\winforms\samples\printing\example2\vb\PrintingExample2.vbproj quickstart\winforms\samples\printing\example3\vb\PrintingExample3.vbproj quickstart\winforms\samples\printing\example4\vb\PrintingExample4.vbproj quickstart\winforms\samples\printing\example5\vb\PrintingExample5.vbproj
HScrollBar, VScrollBar	quickstart\winforms\samples\controlreference\scrollbarctl\vb\ScrollBarCtl.vbproj
ImageList, ToolTip, TreeView	quickstart\winforms\samples\controlreference\treeviewctl\vb\TreeViewCtl.vbproj
LinkLabel, PropertyGrid	quickstart\winforms\samples\controlreference\linklabelctl\vb\LinkLabelCtl.vbproj
ListBox	quickstart\winforms\samples\controlreference\listboxctl\vb\ListBoxCtl.vbproj quickstart\howto\samples\winforms\ownerdrawlistbox\vb\OwnerDrawListBox.vbproj
ListView, ToolTip	quickstart\winforms\samples\controlreference\tooltipctl\vb\ToolTipCtl.vbproj
ProgressBar, TrackBar	quickstart\winforms\samples\controlreference\progressbarctl\vb\ProgressBarCtl.vbproj
RadioButton	quickstart\winforms\samples\layout\dockman\vb\DockMan.vbproj
StatusBar, Timer	quickstart\winforms\samples\controlreference\statusbarctl\vb\StatusBarCtl.vbproj
TabControl	quickstart\winforms\samples\controlreference\tabcontrolctl\vb\TabControlCtl.vbproj
TrackBar	quickstart\winforms\samples\controlreference\trackbarctl\vb\TrackBarCtl.vbproj

Data

Description	**Location**
Simple data binding	quickstart\winforms\samples\data\simplebinding\vb\SimpleBinding.vbproj
Data binding with the ComboBox	quickstart\winforms\samples\data\comboboxbinding\vb\ComboBoxBinding.vbproj
Data binding with the DataGrid	quickstart\winforms\samples\data\grid\vb\Grid.vbproj
Master/Detail: three projects working together	quickstart\winforms\samples\data\masterdetails\vb\MasterDetailsWebService.vbproj quickstart\winforms\samples\data\masterdetails\vb\client\MasterDetails.vbproj quickstart\winforms\samples\data\masterdetails\vb\dataset\CustomersAndOrdersDataSet.vbproj
Update	quickstart\winforms\samples\data\update\vb\UpdateWebService.vbproj quickstart\winforms\samples\data\update\vb\client\Update.vbproj quickstart\winforms\samples\data\update\vb\customersdataset\CustomersDataSet.vbproj
Data binding a DataGrid to an XML Web service	quickstart\winforms\samples\data\webservicebinding\vb\WebService.vbproj quickstart\winforms\samples\data\webservicebinding\vb\client\WebServiceClient.vbproj quickstart\winforms\samples\data\webservicebinding\vb\dataset\DataSet.vbproj

Web

Description	Location
Web Forms user controls	quickstart\aspplus\samples\vstudio\webforms\pagelets\vb\Pagelets.vbproj
Validation	quickstart\aspplus\samples\vstudio\webforms\validate\vb\Validate.vbproj

Other features

Description	Location
Accessibility	quickstart\winforms\samples\accessible\vb\Accessible.vbproj
Printing	quickstart\winforms\samples\printing\example1\vb\PrintingExample1.vbproj quickstart\winforms\samples\printing\example2\vb\PrintingExample2.vbproj quickstart\winforms\samples\printing\example3\vb\PrintingExample3.vbproj quickstart\winforms\samples\printing\example4\vb\PrintingExample4.vbproj quickstart\winforms\samples\printing\example5\vb\PrintingExample5.vbproj
Graphics	quickstart\winforms\samples\gdiplus\brushes\vb\GDIPBrushes.vbproj quickstart\winforms\samples\gdiplus\images\vb\GDIPImages.vbproj quickstart\winforms\samples\gdiplus\pens\vb\GDIPPens.vbproj quickstart\winforms\samples\gdiplus\text\vb\GDIPText.vbproj
Controls: writing and hosting	quickstart\winforms\samples\writingcontrols\helloworldcontrol\vb\control\HelloWorldControl.vbproj quickstart\winforms\samples\writingcontrols\helloworldcontrol\vb\hostapp\HostApp.vbproj quickstart\winforms\samples\writingcontrols\simplecontrol\vb\control\SimpleControl.vbproj quickstart\winforms\samples\writingcontrols\simplecontrol\vb\hostapp\HostApp.vbproj quickstart\winforms\samples\writingcontrols\customuitypeeditor\vb\control\FlashTrackBar.vbproj quickstart\winforms\samples\writingcontrols\customuitypeeditor\vb\hostapp\CustomUITypeEditor.vbproj
Controls: User control	quickstart\winforms\samples\writingcontrols\usercontrol\vb\control\CustomerControl.vbproj quickstart\winforms\samples\writingcontrols\usercontrol\vb\hostapp\UserControl.vbproj
Controls: Licensing	quickstart\winforms\samples\writingcontrols\licensedcontrol\vb
Threading and marshalling	quickstart\howto\samples\winforms\threadmarshal\vb\ThreadMarshal.vbproj

Additional Samples

When you install the .NET Framework SDK, you can also access the following samples that represent complete applications or feature various technologies. (You can also find them on the CD at the location specified below.) The application samples demonstrate more complete error handling and object-oriented development, while the technology samples provide instruction on one or more features.

Each sample also contains a Readme.htm file with a brief description of the sample. The readme file is in the parent directory of the Visual Basic directory. For example, the readme file for the TypeFinder application is located in Samples\applications\typefinder.

You can also access the readme files in the "Microsoft .NET Framework SDK QuickStarts, Tutorials and Samples" application. To access the samples, select Start, Programs, Microsoft .NET Framework SDK, and then Samples and QuickStart Tutorials. The main page of the "Microsoft .NET Framework SDK QuickStarts, Tutorials and Samples" application will appear. There are links on this page to the Visual Basic samples described in the following table.

The following samples are specific to Visual Basic:

Sample	Location on CD
TypeFinder: a console application providing a tool that a user with a command-line interface can use to find out about types in the environment	Samples\applications\typefinder\vb\TypeFinderVB.vbproj
WinTalk: a simple "Sockets Chat" Windows application using Windows Forms	Samples\applications\wintalk\vb\WinTalkVB.vbproj
WordCount: a console application that counts the words in a text file and organizes the results depending on the arguments you use in the command line	Samples\applications\wordcount\vb\WordCountVB.vbproj
Object Construction: a technology sample demonstrating object construction strings	Samples\technologies\componentservices\objectconstruction\VB\OCFormVB.vbproj Samples\technologies\componentservices\objectconstruction\VB\OCObjVB.vbproj
Object Pooling: a technology sample demonstrating multiple client programs using one pooled object in a server application	Samples\technologies\componentservices\objectpooling\VB\OPFormVB.vbproj Samples\technologies\componentservices\objectpooling\VB\OPObjVB.vbproj
Queued Components: a technology sample demonstrating COM + queued components	Samples\technologies\componentservices\queuedcomponents\VB\QCFormVB.vbproj Samples\technologies\componentservices\queuedcomponents\VB\QCObjVB.vbproj
Role-Based Security: a technology sample that uses a server application to track a client application's user account and report whether it belongs to the administrator's role	Samples\technologies\componentservices\RoleBasedSecurity\VB\RBSecFormVB.vbproj Samples\technologies\componentservices\RoleBasedSecurity\VB\RBSecObjVB.vbproj
Configuration: a technology sample demonstrating how to use the .NET Framework Configuration WMI provider with the **System.Management** classes to read and write configuration files	Samples\technologies\Configuration\ConfigTest\VB\ConfigTestVB.vbproj
Delegates and Events: a technology sample demonstrating a type-safe callback mechanism	Samples\technologies\delegatesandevents\vb\DelegatesAndEventsVB.vbproj
Exceptions: a technology sample demonstrating structured exception handling	Samples\technologies\exceptions\vb\ExceptionsVB.vbproj
Garbage Collection: a technology sample showing how the heap manages all class objects by monitoring their lifetimes and freeing the objects when they are no longer referenced	Samples\technologies\GarbageCollection\VB\GarbageCollectionVB.vbproj
Permissions: a technology sample demonstrating how to grant or deny access to resources within a method call	Samples\technologies\permissions\vb\PermissionsVB.vbproj
Custom Attributes: a technology sample demonstrating how to create customer attributes and associate them with types and members. The sample also uses reflection to find out if a custom attribute is associated with a type or member	Samples\technologies\reflection\customattribute\vb\CustomAttributeVB.vbproj
Dynamic Type Resolution: a technology sample demonstrating how a distributed application can increase flexibility and fault-tolerance using a type resolution process.	Samples\technologies\reflection\DynamicTypeResolve\VB\DynamicTypeResolveVB.vbproj
Reflection Invoke: a technology sample demonstrating how a distributed application can increase flexibility and fault-tolerance using a type resolution process	Samples\technologies\reflection\invoke\vb\InvokeVB.vbproj Samples\technologies\reflection\invoke\vb\TesterVB.vbproj
Reflection Emit: a technology sample demonstrating how to use a run-time feature that allows code to create dynamic assemblies, modules, and types	Samples\technologies\reflection\reflectionemit\vb\ReflectionEmitVB.vbproj
Serialization: a technology sample demonstrating how the common language runtime can serialize an object graph in memory to a stream using the SoapFormatter or the BinaryFormatter	Samples\technologies\serialization\vb\SerializationVB.vbproj

Join Thread: a technology sample demonstrating the common language runtime feature that allows one thread to either wait for another thread to terminate or "join' with another thread	Samples\technologies\threading\jointhread\vb\JoinThreadVB.vbproj
Monitor Synchronization: a technology sample demonstrating how to use the **Monitor** class to synchronize threading	Samples\technologies\threading\monitorsynchronization\vb\MonitorSynchronizationVB.vbproj
Thread Pools: a technology sample demonstrating how to use the **ThreadPool** class by queuing an asynchronous method call that is executed by a thread from the thread pool managed by the common language runtime	Samples\technologies\threading\pools\vb\PoolsVB.vbproj
Reader-Writer Synchronization: a technology sample demonstrating how to use the **ReaderWriterLock** class. This allows you to enforce exclusive access to a shared resource while a thread is writing or modifying a resource, but allows nonexclusive access when readying or querying the resource	Samples\technologies\threading\readerwritersynchronization\vb\ReaderWriterSynchronizationVB.vbproj
Thread Static: a technology sample demonstrating how to use the **ThreadStaticAttribute** type to create static variables backed by thread local storage	Samples\technologies\threading\threadstaticvariables\vb\ThreadStaticVariablesVB.vbproj
Timers: a technology sample demonstrating how to use timers in the common language runtime	Samples\technologies\threading\timers\vb\TimersVB.vbproj
Wait Synchronization: a technology sample demonstrating how to use the **WaitHandle** class for thread synchronization by enforcing mutually exclusive access to a protected code block	Samples\technologies\threading\waitsynchronization\vb\WaitSynchronizationVB.vbproj
Value and Enum Types: a technology sample demonstrating how to work with value and enumerated types	Samples\technologies\valueandenumtypes\vb\ValueAndEnumTypesVB.vbproj

See Also

Visual Basic Sample Applications | Visual Basic Walkthroughs | Windows Forms QuickStart (.NET Framework) | QuickStarts (.NET Framework)

Visual C# Samples

You can access sample code in two ways:

- Browse the sample abstracts in this section. Each abstract contains a link to open or copy the sample's files. For help on locating a particular sample, see Finding a Sample.
- Browse the Visual C# samples directory structure on the product CD or DVD (Samples\vcsharp).

In This Section

General Samples
Provides links to samples that use Visual Studio project templates and designers and demonstrate different kinds of applications such as Windows Forms and Web Forms.

Tutorial Samples
Provides links to samples that are used in the C# tutorials.

Cross-Language Samples
Provide links to samples that demonstrate cross-language interoperability between Visual C# and Visual C++.

QuickStarts and Other Samples
Provide an overview and installation procedure of many more samples which cover the most compelling features of the .Net Framework technology.

Related Sections

Topic	Location
Samples	Visual Basic Samples
Samples	
Samples	Visual Basic Concepts
Samples	
Samples	
Samples	Writing C# Applications
Samples	
Samples	
Visual C# Samples	C# Sample Applications
Visual C# Samples	

General Samples

This section contains sample abstracts for the Visual C# general samples. The samples in this section use Visual Studio project templates and designers and demonstrate different kinds of applications such as Windows Forms and Web Forms.

Sample	Description
AutoWord	Demonstrates how to automate Word from Visual C#.
CarsSelector	Demonstrates how to create a Web Forms page for selecting cars that simulates a simple e-commerce page. The sample demonstrates using the ASP.NET technology.
LocalizedHelloWorld	Demonstrates how to create and use satellite assemblies for console applications.
MultilingualForm	Demonstrates how to create and localize Windows Forms.
PerfMon	Demonstrates how to monitor system performance using performance counters and is an introduction to p erformance counters.
ProcessController	Demonstrates how to build a Visual C# Windows application for processes and services monitoring.
Scribble	Demonstrates a Windows Forms MDI drawing application.

See Also

Visual C# Samples

AutoWord Sample: Demonstrates Automating Microsoft Word

The AutoWord sample includes four examples that show how to automate Microsoft Word from Visual C#. The examples demonstrate how to automate the following activities:

- Starting Word
- Creating a new document
- Opening an existing document
- Hooking events in the Word application

The sample also shows how to create an interop assembly for Microsoft Word, and shows some rudimentary techniques for automating Word.

> **Note** One of the projects in this sample is a C++ project, so Visual C++ must also be installed.

Projects in the Sample

This sample contains five projects and produces three assemblies.

Building and Running the Sample

To create the assemblies

1. Open the solution AutoWord.sln.
2. Right-click the **RunCreateWordAssembly** project, then click **Build.**
3. For each of the example projects, right-click the **References** item, then click **Add Reference**. Click the **Browse** button, select WORD.DLL from the list of .dll files, then click the **Open** button. Click the **OK** button to close the reference browser, and click the **OK** button for the popup window telling you that the directory WORD.DLL was found in will be added to the reference path.
4. If you have installed Microsoft Word XP or Office XP, add **OFFICEXP** to the list of compilation constants. For each of the example projects, right-click the project root, then click **Properties**. Select **Configuration Properties**, then **Build**, and then add **OFFICEXP** to the list of **Conditional Compilation Constants**. Click the **OK** button.

   > **Note** It is not enough to define **OFFICEXP** in the Debug and the Release configurations. You must define **OFFICEXP** in each configuration.

5. From the **Build** menu, click **Build Solution**.

To run the automation examples

1. Right-click **Example1** in **Solution Explorer** and select **Set as Startup Project** from the shortcut menu, then build and run the project.

   This example shows you how to instantiate the Word object without any documents.

2. Right-click **Example2** in **Solution Explorer** and select **Set as Startup Project** from the shortcut menu, then build and run the project.

   This example shows you how to create a new Word document and put some text into it.

3. Right-click **Example3** in **Solution Explorer** and select **Set as Startup Project** from the shortcut menu, then build and run the project.

   This example shows you how to open an existing Word document and modify it.

4. Right-click **Example4** in **Solution Explorer** and select **Set as Startup Project** from the shortcut menu, then build and run the project.

   This example shows you how to hook various events in Word.

Classes and Keywords

This sample demonstrates the following classes:

System.Diagnostics.Process, System.Environment, System.Reflection.Missing, System.Text.StringBuilder, System.Windows.Forms.MessageBox, System.Thread.Sleep, System.Reflection.Missing.Value, Microsoft.Win32.RegistryKey, Word.Application, Word.ApplicationEvents2_DocumentChangeEventHandler, Word.ApplicationEvents2_DocumentOpenEventHandler, Word.ApplicationEvents2_NewDocumentEventHandler, Word.ApplicationEvents2_StartupEventHandler, Word.Documents, Word._Document, Word.Range, Word.Words

This sample demonstrates the following keyword:

event

See Also

Visual C# Samples | General Samples

CarsSelector Sample: Demonstrates Creating and Using ASP.NET Web Pages

The CarsSelector sample shows you how to build Web Forms page for selecting cars that simulates a simple e-commerce page. The sample demonstrate the following features:

- Data binding: In this sample you use more than one type of data binding:
  - Simple data binding.
  - Binding to a data table created programmatically (the data source is not coming from a database).
- Using the event handler to dynamically change the binding of some controls.
- Using validation controls.
- Using the Calendar control and validating its entries without using the validation controls.

Requirements

To build and run this sample, you need a computer running Internet Information Services 5.0 with FrontPage Server Extensions installed.

Building and Running the Sample

To create the solution

1. Copy the sample files to your hard disk.
2. Create a new **Web Application** and name it CarsSelector.
3. Delete the default Web Forms page by right-clicking the file **WebForm1.aspx**, and then click **Delete** on the shortcut menu.
4. Add the file CarsSelectorSample.aspx to the project. To do that, right-click the project name, point to **Add** on the shortcut menu, and click **Add Existing Item**. Change the **Types of files** to All Files (*.*), then browse to the file CarsSelectorSample.aspx, select it, and click **Open**.
5. Add the file CarsSelectorSample.aspx.cs to the project. To do that, right-click the project name, point to **Add** on the shortcut menu, and click **Add Existing Item**. Then browse to the file CarsSelectorSample.aspx.cs, select it, and click **Open**. Click **OK** to overwrite the existing file.
6. Set the Web Forms page as a start page. To do that, right-click the file **CarsSelectorSample.aspx** and click **Set As Start Page** on the shortcut menu.

For more information on creating Web Forms applications**,** see Creating and Managing Web Forms.

To build and run the sample

1. On the **Build** menu, click **Build Solution**.
2. On the **Debug** menu, click **Start Without Debugging**.

   The Web Forms page appears. You can select a car and make choices, and then you see the result bound to a label.

To use the Web form

1. Start by selecting a car brand from the drop-down list **Brand**.
2. Choose a model from the **Model** drop-down list. You will notice that the contents of this list depend on the selected brand of the car.
3. Choose the **Year**, **Color**, and **Mileage** from the appropriate drop-down list.
4. Enter an e-mail address in the **Your Email** text box.
5. Select one or more check boxes from the **Features Desired**.
6. Select a valid date in the **Desired Available Date** calendar.
7. Click **Submit**.

Classes and Keywords

This sample demonstrates the following class:

System.Web.UI.Page

This sample demonstrates the following keywords:

DropDownList, AutoPostback, CheckBoxList, RequiredFieldValidator, RegularExpressionValidator, Dilatable

See Also

Visual C# Samples | General Samples

LocalizedHelloWorld Sample: Demonstrates Satellite Assemblies for Console Applications

The LocalizedHelloWorld sample demonstrates creating and using satellite assemblies for console applications. It shows how to create resources for C# console applications, localize the resources, and build localized satellite assemblies.

Although this sample was written using C#, the sample principles and techniques can be used to localize any managed application written in any language.

Building and Running the Sample

To build and run this sample

1. Open the solution LocalizedHelloWorld.sln.
2. From the **Build** menu, click **Build Solution**.
3. From the **Debug** menu, click **Start Without Debugging**.
4. Type in a culture supported by the application to see the resources in that culture (the sample ships with **en**, **en-US**, **ko-KR**, and **default** resources which are in English).

To add a new culture

The following procedure adds resources for English (United Kingdom):

1. Add a new text file named "MyResources.en-GB.txt" to the Resources C++ utility project.
2. Add the appropriate <name>=<value> lines to the new text file along with any desired comments.
3. Add a custom build step for the new file (this should be the same as for the other resource files in the project):

   ResGen.exe "$(InputPath)"

4. Set the output of the custom build step to "$(InputName).resources."
5. Build this project (no need to rebuild the main assembly).
6. Add an existing file named "..\Resources\MyResources.en-GB.resources" to the SatelliteAssemblies C++ utility project.
7. Add a custom build step for this file (this should be similar to the other files' custom builds steps):

   mkdir $(ProjectDir)..\bin\$(OutDir)\en-GB

   al.exe /culture:en-GB /out:"$(ProjectDir)..\bin\$(OutDir)\en-GB\HelloWorld.resources.dll" /embed:"$(InputPath)","HelloWorld.Resources.$(InputFileName)" /template:"$(ProjectDir)..\obj\$(IntDir)\HelloWorld.exe"

8. Set the output of the custom build step to "$(ProjectDir)..\bin\$(OutDir)\en-GB\HelloWorld.resources.dll"
9. Build this project (no need to rebuild the main assembly).
10. Run the sample and try your new resource.

To build the sample from the command line

Note You can add options to the C# command-line for debugging or other reasons.

1. Compile the resource for all cultures (from the Resources subdirectory):

   ResGen.exe MyResources.txt

   ResGen.exe MyResources.en-US.txt

   ResGen.exe MyResources.ko-KR.txt

2. Create the output directory "bin\Debug" or use a different directory in the following steps.
3. Compile the main assembly and embed the default resources (from the root project directory):

   csc.exe /out:bin\Debug\HelloWorld.exe /res:Resources\MyResources.resources,HelloWorld.Resources.MyResources.resources Class1.cs AssemblyInfo.cs

4. Build the satellite assemblies (from the root project directory):

   mkdir bin\Debug\en-US

   mkdir bin\Debug\ko-KR

   al.exe /culture:en-US /out:bin\Debug\en-US\HelloWorld.resources.dll /embed:Resources\MyResources.en-US.resources,HelloWorld.Resources.MyResources.en-US.resources /template:bin\Debug\HelloWorld.exe

   al.exe /culture:ko-KR /out:bin\Debug\ko-KR\HelloWorld.resources.dll /embed:Resources\MyResources.ko-KR.resources,HelloWorld.Resources.MyResources.ko-KR.resources /template:bin\Debug\HelloWorld.exe

Projects in the Sample

This sample contains three projects and produces three assemblies.

Resources C++ Utility Project

This is the first project. It builds the managed resources with a custom build step that calls ResGen.exe to compile the .txt files into .resources files. The resource text file contains lines of <name>=<value> statements and comment lines that begin with a pound sign (#). In code, the strings will be loaded using

the name given in this file.

Notice that for the Korean resources the file must be saved as UTF-8 to preserve the non-ASCII characters. Also note that individual cultures are not required to have a value for every name.

If a value doesn't exist for the proper culture, the resource manager automatically falls back to the parent cultures. For example, if the current culture is ko-KR, the resource manager will first look for the name in the "ko-KR" resources, then in the "ko" resource, and then finally in the default resources. Technically these resource files can be named anything, but typically they are named with the following convention:

<basename>[.<culture>].resources

If the culture is empty, it is assumed to be the default culture.

HelloWorld C# Console Project

This is the second project. It builds the main C# application and embeds the default resources (MyResources.resources built from MyResources.txt by the first project).

SatelliteAssemblies C++ Utility Project

This is the third project. It builds the two satellite assemblies that contain the localized resources with custom build steps. It calls Al.exe to create the satellite assembly, passing in the resource file with the "/embed" option, setting the culture with the "/culture" option, and copying the assembly information (title, version, and so on) from the main assembly with the "/template" option.

Note that the satellite assemblies must be in a subdirectory named after their culture of the main assembly, and the satellite assembly's file name is the same as the main assembly, but with ".resources" inserted before the file extension. For example, if the main assembly was "c:\bin\a.dll", the en-US satellite assembly would be "c:\bin\en-US\a.resources.dll".

Also note the second argument to the "/embed" option, the resources name. This name is the name that the resource manager will use to find the given resource group, and the name must include the proper culture string (case-sensitive) after the name and before ".resources". Because Al.exe gets most of the information from the main assembly, this third project must be built after the second one.

Classes and Keywords

This sample demonstrates the following classes:

System.Resources.ResourceManager

This sample demonstrates the following keywords:

managed resources, satellite assembly, localized resources, ResGen.exe, al.exe, /embed, /res, /template, /culture, System.Resources, System.Resources.ResourceManager, System.Resources.ResourceManager.GetString

See Also

Visual C# Samples | General Samples | Console: Localized Hello World Sample (Visual Basic)

MultilingualForm Sample: Demonstrates Creating and Localizing Windows Forms

The MultilingualForm sample demonstrates creating and localizing windows forms. It shows how to use the Windows Forms designer to create, edit, and build localized applications.

For more information about localized applications, see Globalizing and Localizing.

Building and Running the Sample

To build and run this sample

1. Open the solution MultilingualForm.sln.
2. From the **Build** menu, click **Build Solution**.
3. From the **Debug** menu, click **Start Without Debugging**.
4. To see a different culture's resources, type a valid RFC 1766 culture name into the edit box and click the **switch** button. (This sample only comes with resources for "en-US" and "ko-KR".)

To localize your own Windows Forms or Web Forms

1. Select the form to be localized in the appropriate forms designer.
2. Set the form's **Localizable** property to true.
3. For each language you want to localize to, do the following:
   a. Set the form's **Language** property to the desired language.
   b. Enter the new text in the new language for each control.

   You can also change the appearance of the form's controls (size, position, and so on) for each language.

Classes and Keywords

This sample demonstrates the following classes:

System.Resources.ResourceManager

This sample demonstrates the following keywords:

System.Windows.Forms.Form.Localizable; System.Windows.Forms.Form.Language; System.Resources; System.Resources.ResourceManager

See Also

Visual C# Samples | General Samples | Windows Forms: Multilingual Form Sample (Visual Basic)

PerfMon Sample: Demonstrates How to Monitor System Performance Using Performance Counters

The PerfMon sample is an introduction to performance counters. It demonstrates how to monitor local or remote system performance. It shows how to dynamically create timers, context menus, and graphic displays. It also demonstrates the following activities:

- Including new C# classes derived from other components.
- Dynamically adding and removing event handlers and components.
- Using the **TreeView** control, **TreeNode** class, and **Timer** component.
- Using the **PerformanceCounterCategory** and **PerformanceCounter** components.

For more information on performance counters, see PerformanceCounter Class.

> **Note** This sample is functional only on Windows NT 4.0 and Windows 2000 and later.

Building and Running the Sample

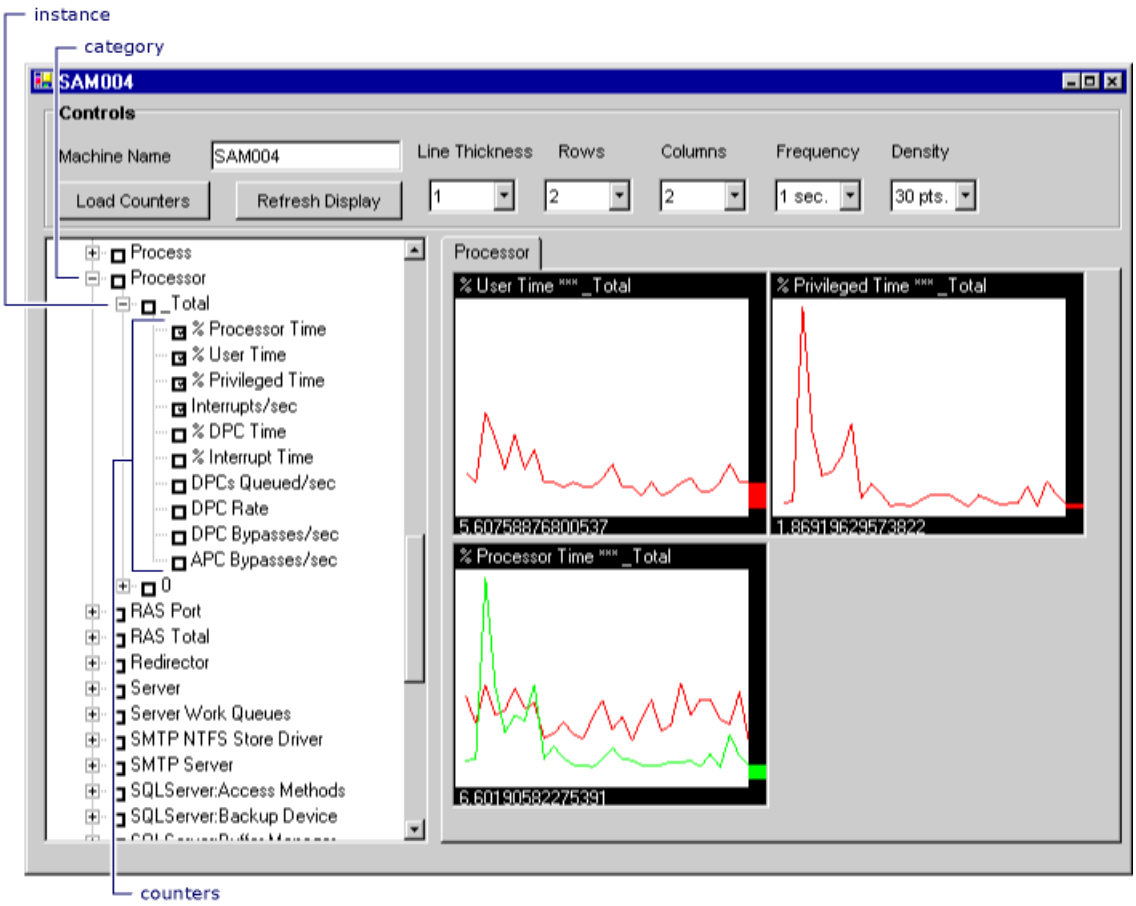
To build and run this sample

1. Open the solution PerfMon.sln.
2. From the **Build** menu, click **Build Solution**.
3. From the **Debug** menu, click **Start Without Debugging**.

   The main application window will appear.

The following figure shows the performance monitor program running. Refer to the figure, as needed, while you go through the various performance monitoring procedures.

Performance Monitor



The Display Panel Properties

Each graphic display in the figure shows the following:

- The top of the display shows the name of the counter and the name of the instance separated by "***". If the category does not contain an instance name (for example, the ICMP category), the "***" is followed by a blank space.
- The bottom of the display shows the current counter value. This value may change periodically (the default is every second).
- The right side of the display shows a red bar. The height of the bar depends on the last value monitored on the counter.
- The graph plotted in the display shows the performance history for that counter.

To load the performance counters of a system

This application is an MDI application. It starts with four windows that can be used to monitor four computers. You can open as many windows as you want by clicking the **New PerfMon** command on the **Options** menu. In the following procedures, only one window is used. Therefore, you can close all of the windows except one.

1. In the **Machine Name** field, type the name of the system you want to monitor. If you do not type anything, the local machine will be monitored.
2. Click the **Load Counters** button (or press **ALT+L**). The application will load **PerformanceCounterCategory**, **InstanceName**, and **PerformanceCounter** components on the specified machine. The **Load Counters** button will be disabled until all of the counters on the system have been loaded.

   The tree view in the left pane will contain all of the performance categories and the performance counters available. Each leaf in the tree represents a performance counter. You can monitor the completion progress by watching the progress indicator on the bottom of the application window. After the tree has been completed, the **Load Counters** button will be enabled again.

   > **Note** To monitor a specific machine, you need administrator rights to the machine; otherwise, the **PerformanceCounterCategory** components will not be able to load the categories (security protection).

To monitor the performance of a system

1. Load the performance counters by performing the preceding steps.
2. Select one of the counters by expanding the corresponding category node. For example, to watch the performance of the processor, expand the **Processor** node, and then select one of the **Processor** instances (for example, "**_Total**" or "**0**"). Expand the instance node as well to see the counters.
3. On the leaf level, the counters are represented by check boxes. Select one of the counters, for example, **%Processor Time**. A tab, which has the name **Processor**, will appear in the right pane, and a graphic display of the selected counter will appear on the tab page.
4. To remove a specific graphic display from the tab page, clear the check box for the corresponding counter. When all of the counters in a specific page are cleared, the corresponding tab disappears.

To manage the displays properties

> **Note** Each time you change any of the display properties, click the **Refresh Display** button to apply the changes. The changes will apply for current or future displays.

- To change the line thickness, select the thickness you want from the **Line Thickness** box.
- To change the number of rows/display, select the number you want from the **Rows** box. For example, selecting one row makes the display occupy the full height of the tab page.
- To change the number of columns/display, select the number you want from the **Columns** box. For example, selecting one column makes the display occupy the full width of the tab page.

  > **Note** If the number of graphic displays exceeds the visible area of the tab page, scroll bars will be created automatically.

- To change the time interval, select the number of seconds from the **Frequency** box.
- To change the number of points to display in the graph, select a number from the **Density** box.

To merge two or more displays

1. Drag the display you want to move and drop it on the destination display. Notice that the background of the destination display becomes black when it is selected and is ready to receive the moved display. Both displays will merge, and two different graphic colors will appear (red and green in this case). For example, if you drop "**%Processor Time *** _total**" on "**%User Time *** _total**", the green color will represent the first and the red color will represent the second. The data shown in the display's top, bottom, and right side will only represent the **"%Processor Time *** _total**" counter.

2. To switch from one graph to another after a display merge, right-click the display and select the name of the graph from the shortcut menu. In the preceding example, if you want to change the display data from "**% Processor Time *** _total**" to "**%User Time *** _total**", right-click the display and select **"%User Time *** _total**" from the shortcut menu. The display will show the data of the selected counter, and the bar color on the right side will change to red (or the color representing the selected counter).
3. If you have more than two merged counters, each counter will be represented by a different color. To select the data view for any one of the counters, perform the preceding step.

To separate two or more merged displays

1. Locate the counter's check box of the display you want to separate.
2. Clear the counter's check box. The corresponding display will be removed.
3. Select the counter's check box again. A new display will appear on the tab page.

To use menu options

- To stop the graphics plotting:

  1. Click the **Option** menu (or press ALT+O).
  2. Click **Stop Graphics**.

  The graphics will freeze and no more lines will be plotted.

- To restart the graphics plotting:

  1. Click the **Option** menu.
  2. Click **Start Graphics**.

  The displays will continue plotting from the last position.

- To exit the application:

  1. Click the **Option** menu.
  2. Click **Exit**.

Classes and Keywords

This sample demonstrates the following classes:

System.Windows.Forms.Form, System.Windows.Forms.TabControl, System.Windows.Forms.TabPage, System.Windows.Forms.Panel, System.Windows.Forms.ProgressBar, System.Windows.Forms.TreeView, System.Windows.Forms.TreeNode, System.Windows.Forms.MouseButtons, System.Windows.Forms.ContextMenu, System.Windows.Forms.MenuItem, System.Collection.HashTable, System.Drawing.Color, System.Drawing.Graphics, System.Diagnostics.PerformanceCounterCategory, System.Diagnostics.PerformanceCounter

This sample demonstrates the following keywords:

PerformanceCounterCategory, PerformanceCounter, Graphics, GraphicTab, HashTable, DisplayPanel, ContextMenu, MenuItem, Draw, Color, TreeView, TreeNode, ProgressBar, TabControl, TabPage

See Also

Visual C# Samples | General Samples | Windows Forms: Performance Monitor Sample (Visual Basic)

ProcessController Sample: Demonstrates Processes and Services Monitoring

The ProcessController sample shows how to build a C# Windows application for processes and services monitoring. The sample also demonstrates the following activities:

- Dynamically adding and removing event handlers.
- Using the **Process** component to:
  - Read the running processes on the local/remote machine.
  - Start a process on the local machine.
  - Stop a process on the local/remote machine.
  - Read the modules used by some running process.
  - Read all of the characteristics of some running process.
- Using the **ServiceController** component to:
  - Read the running services and device drivers on the local/remote machine.
  - Start/Stop/Pause a service on the local/remote machine.
  - Check properties of the services or device drivers on the local/remote machine.
- Handling security issues related to the use of the **Process** and the **ServiceController** components.

  Note This sample only runs on the Windows NT or Windows 2000 operating systems since the **ServiceController** component is only supported on Windows NT platforms.

Building and Running the Sample

To build and run this sample

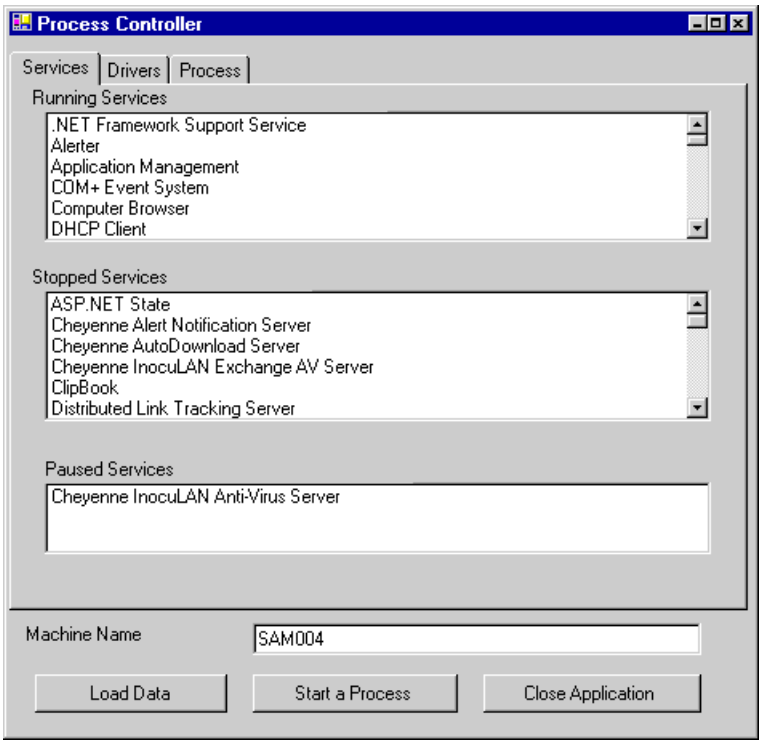
1. Open the solution ProcessController.sln.
2. From the **Build** menu, click **Build Solution**.
3. From the **Debug** menu, click **Start Without Debugging**.

   The main application window will appear.

The following figure shows the ProcessController program running. Refer to the figure, as needed, while you perform the following test procedures:

- Load all services, device drivers, and processes running on some machine in the network.
- Start/Pause/Stop a service or a device driver running on the local/remote machine.
- Start a process running on the local machine.
- Terminate a process running on the local/remote machine.

Process Controller



To load all of the services, device drivers, and processes

1. Type the name of the machine you want in the **Machine Name** box.
2. Click the **Load Data** button (or press ALT+L).

To start a service

1. Click a service name in the **Stopped Services** list.
2. Right-click the service and click **Start Service** from the shortcut menu.

   After the service starts, its name should appear in the topmost list labeled **Running Services**.

   Note You cannot start a service if you do not have the administrator rights on that machine.

To stop a service

1. Click a service name in the **Running Services** list.
2. Right-click the service and click **Stop Service** from the shortcut menu.

   After the service has been stopped, the name should appear in the **Stopped Services** list.

   Note Some services do not accept the **Stopped** state. Also, you cannot stop a service if you do not have the administrator rights on that machine.

To pause a running service

1. Click a service name in the **Running Services** list.
2. Right-click the service and click **Pause** from the shortcut menu.

   If the service is paused, the service name should appear in the **Paused Services** list at the bottom of the window.

   Note Some services do not accept the **Paused** state. Also, you cannot pause a service if you do not have the administrator rights on that machine.

To check some of the service's information

1. Click a service name in any of the three lists.
2. Right-click the service and click **Show Service Info** from the shortcut menu. A new window containing the service information will appear.

To verify and manage device drivers

1. Click the **Drivers** tab at the top of the window.

2. Go through the preceding steps to start, stop, and pause a service. Exactly the same rules mentioned for services apply to the device drivers.

To verify and manage processes

1. Click the **Process** tab at the top of the window.
2. Select a process name in the leftmost list. At this point, information about that service should appear in the list on the right.

   Note If you are not an administrator on the target machine, you will not be able to see or manage any process.

To start a process

1. Click the **Start a Process** button (or press ALT+S).
2. Select the process you want to start in the **Open** dialog box.
3. Click the **Open** button in the dialog box. If your target machine is the local machine, the process name should appear in the leftmost list.

   Note If the target machine is a remote machine, the process will start on the local machine, by default, if possible.

To stop a process

1. Click a process in the leftmost list.
2. Right-click the process and click **Terminate Process** from the shortcut menu.

   After the process has been terminated, its name should disappear from the list.

To load the services, drivers, and processes from another machine

1. Type the name of the machine in the **Machine Name** box.
2. Click the **Load Data** button (or press ALT+L).

To stop the ProcessController application

- Click the **Close Application** button (or press ALT+C).

  After you click the button, the window will slowly fade away. This effect is obtained by handling the Windows Opacity property (file: MainForm.cs, method: cmdClose_Click).

Description of the Sample Classes

File	Description
MainForm.cs	Describes the MainForm class, which is a **System.Windows.Forms.Form** type class. Its designer contains m ost of the UI controls needed for this application.
ProcessControllerM anager.cs	Contains the implementation of the ProcessControllerManager class. This class handles processes managem ent.
DriverControllerMa nager.cs	Contains the implementation of the DriverControllerManager class. This class is in charge of managing the d evice drivers on a given machine.
ServiceControllerM anager.cs	Contains the implementation of the ServiceControllerManager class. This class describes the logic to control the services on a given machine.
ServiceInfo.cs	Implements the ServiceInfo class, which is a **System.Windows.Forms.Form** class and is used to display ser vices and drivers information only.

Classes and Keywords

This sample demonstrates the following classes:

System.Windows.Forms.Form, System.Windows.Forms.TabControl, System.Windows.Forms.ListBox, System.Windows.Forms.TabPage, System.Windows.Forms.OpenFileDialog, System.Collections.Hashtable, System.Windows.Forms.Timer, System.EventHandler, System.Windows.Forms.ContextMenu, System.Diagnostics.Process, System.ServiceProcess.ServiceController

This sample demonstrates the following keywords:

C# Windows Application; Process component; ServiceController component; C# Class inclusion; TabControl and TabPage; Event Handler dynamic load; ListView control; ContextMenu control and MenuItems; HashTable object; Timer control; System security

issues; System.Array.Sort

See Also

Visual C# Samples | General Samples | Windows Forms: Process Controller Sample (Visual Basic)

Scribble Sample: Visual C# MDI Drawing Application

The Scribble sample shows how to develop a Windows Forms MDI application using Visual C# and the .NET Framework classes. Scribble is a small drawing application that lets you draw free-hand drawings using the mouse and save the images in a file.

Building and Running the Sample

To build and run this sample

1. Open the solution Scribble.sln.
2. From the **Build** menu, click **Build Solution**.
3. From the **Debug** menu, click **Start Without Debugging**.

When you run the sample, you get a drawing surface and the following menu items:

- **File** menu for input and output operations.
- **Edit** menu to cut, copy, and paste.
- **Pen** menu to select the pen thickness.
- **View** menu to show or hide the toolbar and the status bar.
- **Window** menu to create, tile, or cascade windows.
- **Help** menu to display the help topics and the About Scribble text box.

In addition to the menu options, the user interface includes a toolbar with buttons to duplicate the menu commands: **New**, **Open**, **Save**, **Print Preview**, **Print**, and **Help**.

Other Versions of Scribble

Scribble is also available as an MFC sample, as a Managed Extensions for C++ sample, and as Visual Basic sample:

- Scribble Sample: MFC MDI Drawing Application
- Scribble Sample: MDI Drawing Application Using Managed Extensions for C++
- Windows Forms: Scribble Sample

Classes and Keywords

This sample demonstrates the following classes:

System.Windows.Forms; System.Collections.Arraylist; System.Drawing.Printing; System.Runtime.Serialization; System.Runtime.Serialization.Formatters.Binary

This sample demonstrates the following keywords:

System.Windows.Forms.Form; System.Windows.Forms.Form.MDIParent; System.Windows.Forms.Form.MDIChildren; System.Windows.Forms.ImageList; System.Windows.Forms.HelpProvider; System.Windows.Forms.ToolBar; System.Windows.Forms.StatusBar; System.Windows.Forms.OpenFileDialog; System.Windows.Forms.SaveFileDialog; System.Windows.Forms.Menu; System.Windows.Forms.MenuItem; System.Windows.Forms.PrintPreviewDialog; System.Windows.Forms.Help; System.Drawing.Printing; System.Drawing.Printing.PrintDocument; System.Drawing.Pen; System.Drawing.Graphics; System.Drawing.Graphics.DrawLine; System.Collections.ArrayList; System.Runtime.Serialization.Formatters.Binary

See Also

Visual C# Samples | General Samples

Tutorial Sample

The code for the ATL Server Tutorial.

Demonstrates Web access to a database for a simple online store. When you run the application, a Web page prompts you for a username and password. You can either enter a username and password for an existing customer in the database, or create a new account, in which case the application will write the new username and password to the database.

Building and Running the Sample

To build and run this sample

1. Open the solution file, Tutorial.sln, in the Visual Studio development environment.
2. Build the solution. This will also deploy the solution to the local Web server.
3. Configure the data link to connect to the Tutorial.mdb database. In the sample's Tutorial\Tutorial\bin directory, double-click Tutorial.udl.

   The **Data Link Properties** dialog box appears.
4. On the **Provider** tab, select **Microsoft Jet 4.0 OLE DB Provider**.
5. On the **Connection** tab, click the Ellipsis (...) button to specify the Tutorial.mdb file in the Tutorial\Tutorial\bin directory. Leave the username and password blank.
6. Ensure that Tutorial.mdb is not read-only. Right-click the file and select **Properties**; on the **General** tab, ensure that the **Read-only** check box is cleared.
7. Change the NTFS security settings on the Tutorial.mdb file to allow read-write access by the Internet Guest Account, **IUSR**_*Machine*. To do this:
   a. Right-click the Tutorial.mdb file and select **Properties** on the shortcut menu.
   b. In the **Properties** dialog box, select the **Security** tab and click **Add**.
   c. The **Select Users or Groups** dialog box appears; in that dialog box, set the following:

      Object Types to **Users**.

      Locations to the local machine.

      Check Names to **IUSR**_*Machine*.

   d. Select the Internet Guest Account that you just created and in the **Permissions** box, check the **Read** and **Write** boxes if they are not already.

      > **Note** See ATL Server Security for information about the accounts used by a typical ATL Server application running in IIS.
8. Change the NTFS security settings on the folder containing Tutorial.mdb to allow only write access by **IUSR**_*Machine* (using the procedure described in Step 7).
9. Change the NTFS security settings on the Tutorial.udl file to allow read access by the **IUSR**_*Machine* account (using the procedure described in Step 7).
10. Use a Web browser to view http://localhost/tutorial/login.srf.

For more information, see ATL Server Tutorial.

See Also

ATL Server Samples | ATL Server | ATL Server Reference

Hello World Sample

This sample shows several versions of a Hello World program in C#.

This sample contains the source code for the Hello World Tutorial.

Building and Running the Sample Within Visual Studio

To build and run the Hello World samples

1. Open the solution (HelloWorld.sln).
2. In **Solution Explorer**, right-click the HelloWorld1 project and click **Set as StartUp Project**.
3. From the **Debug** menu, click **Start Without Debugging**.
4. Press any key to close HelloWorld1.
5. In **Solution Explorer**, right-click the HelloWorld2 project and click **Set as StartUp Project**.
6. From the **Debug** menu, click **Start Without Debugging**.
7. Press any key to close HelloWorld2.
8. In **Solution Explorer**, right-click the HelloWorld3 project and click **Set as StartUp Project**.
9. In **Solution Explorer**, right-click the HelloWorld3 project and click **Properties**.
10. Open the Configuration Properties folder and click **Debugging**.
11. In the Command Line Arguments property, type "A B C D" and click **OK**.
12. From the **Debug** menu, click **Start Without Debugging**.
13. Press any key to close HelloWorld3.
14. In **Solution Explorer**, right-click the HelloWorld4 project and click **Set as StartUp Project**.
15. From the **Debug** menu, click **Start Without Debugging**.
16. Press any key to close HelloWorld4.

Building and Running the Sample from the Command Line

To build and run the Hello World samples

1. Use the **Change Directory** command to change to the HelloWorld directory.
2. Type the following:

```
cd HelloWorld1
csc Hello1.cs
Hello1
```

3. Type the following:

```
cd ..\HelloWorld2
csc Hello2.cs
Hello2
```

4. Type the following:

```
cd ..\HelloWorld3
csc Hello3.cs
Hello3 A B C D
```

5. Type the following:

```
cd ..\HelloWorld4
csc Hello4.cs
Hello4
```

See Also

Command Line Parameters Sample

This sample shows how the command line can be accessed and two ways of accessing the array of command-line parameters.

This sample contains the source code for the Command Line Parameters Tutorial.

Building and Running the Sample Within Visual Studio

To build and run the Command Line Parameters samples

1. Open the solution (CommandLine.sln).
2. In **Solution Explorer**, right-click the CmdLine1 project and click **Set as StartUp Project**.
3. In **Solution Explorer**, right-click the project and click **Properties**.
4. Open the Configuration Properties folder and click **Debugging**.
5. In the Command Line Arguments property, type the command-line parameters (see the tutorial for an example) and click **OK**.
6. From the **Debug** menu, click **Start Without Debugging**.
7. Repeat the preceding steps for CmdLine2.

Building and Running the Sample from the Command Line

To build and run the Command Line Parameters samples

1. Use the **Change Directory** command to change to the CmdLine1 directory.
2. Type the following:

```
csc cmdline1.cs
cmdline1 A B C
```

3. Use the **Change Directory** command to change to the CmdLine2 directory.
4. Type the following:

```
csc cmdline2.cs
cmdline2 John Paul Mary
```

See Also

Command Line Parameters Tutorial | Tutorial Samples

Arrays Sample

This sample describes and shows how arrays work in C#.

This sample contains the source code for the Arrays Tutorial.

Building and Running the Sample Within Visual Studio

To build and run the Arrays sample

1. Open the solution (Arrays.sln).
2. From the **Debug** menu, click **Start Without Debugging**.

Building and Running the Sample from the Command Line

To build and run the Arrays sample

- Type the following at the command prompt:

```
csc arrays.cs
arrays
```

See Also

Arrays Tutorial | Tutorial Samples

Properties Sample

This sample shows how properties are an integral part of the C# programming language. It demonstrates how properties are declared and used.

This sample contains the source code for the Properties Tutorial.

Building and Running the Sample Within Visual Studio

To build and run the Properties samples

1. Open the solution (Properties.sln).
2. In **Solution Explorer**, right-click the person project and click **Set as StartUp Project**.
3. From the **Debug** menu, click **Start Without Debugging**.
4. Repeat the preceding steps for shapetest.

Building and Running the Sample from the Command Line

To build and run the Properties samples

1. Use the **Change Directory** command to change to the person directory.
2. Type the following:

```
csc person.cs
person
```

3. Use the **Change Directory** command to change to the shapetest directory.
4. Type the following:

```
csc abstractshape.cs shapes.cs shapetest.cs
shapetest
```

See Also

Properties Tutorial | Tutorial Samples

Libraries Sample

This sample shows how to create and use a DLL in C#.

This sample contains the source code for the Libraries Tutorial.

Building and Running the Sample Within Visual Studio

To build and run the Libraries sample

1. Open the solution (Libraries.sln).
2. In **Solution Explorer**, right-click the FunctionTest project and click **Set as StartUp Project**.
3. In **Solution Explorer**, right-click the FunctionTest project and click **Properties**.
4. Open the Configuration Properties folder and click **Debugging**.
5. In the Command Line Arguments property, enter `3 5 10`.
6. Click **OK**.
7. From the **Debug** menu, click **Start Without Debugging**. This will automatically build the library in Functions and execute the program.

Building and Running the Sample from the Command Line

To build and run the Libraries sample

1. Use the **Change Directory** command to change to the Functions directory.
2. Type the following:

```
csc /target:library /out:Functions.dll Factorial.cs DigitCounter.cs
```

3. Use the **Change Directory** command to change to the FunctionTest directory.
4. Type the following:

```
copy ..\Functions\Functions.dll .
csc /out:FunctionTest.exe /R:..\Functions.DLL FunctionClient.cs
FunctionTest 3 5 10
```

See Also

Libraries Tutorial | Tutorial Samples

Versioning Sample

This sample demonstrates versioning in C# through the use of the **override** and **new** keywords. Versioning maintains compatibility between base and derived classes as they evolve.

This sample contains the source code for the Versioning Tutorial.

Building and Running the Sample Within Visual Studio

To build and run the Versioning sample

1. Open the solution (Versioning.sln).
2. From the **Debug** menu, click **Start Without Debugging**.

Building and Running the Sample from the Command Line

To build and run the Versioning sample

- Type the following at the command prompt:

```
csc versioning.cs
versioning
```

See Also

Versioning Tutorial | Tutorial Samples

Collection Classes Sample

This sample shows how to implement a collection class that can be used with the **foreach** statement.

This sample contains the source code for the Collection Classes Tutorial.

Building and Running the Sample Within Visual Studio

To build and run the Collection Classes samples

1. Open the solution (CollectionClasses.sln).
2. In **Solution Explorer**, right-click the CollectionClasses1 project and click **Set as StartUp Project**.
3. From the **Debug** menu, click **Start Without Debugging**.
4. Repeat the preceding steps for CollectionClasses2.

Building and Running the Sample from the Command Line

To build and run the Collection Classes samples

1. Use the **Change Directory** command to change to the CollectionClasses1 directory.
2. Type the following:

```
csc tokens.cs
tokens
```

3. Use the **Change Directory** command to change to the CollectionClasses2 directory.
4. Type the following:

```
csc tokens2.cs
tokens2
```

See Also

Collection Classes Tutorial | Tutorial Samples

Structs Sample

This sample presents the syntax and usage of structs. It also covers the important differences between classes and structs.

This sample contains the source code for the Structs Tutorial.

Building and Running the Sample Within Visual Studio

To build and run the Structs samples

1. Open the solution (Structs.sln).
2. In **Solution Explorer**, right-click the Struct1 project and click **Set as StartUp Project**.
3. From the **Debug** menu, click **Start Without Debugging**.
4. Repeat the preceding steps for Struct2.

Building and Running the Sample from the Command Line

To build and run the Structs samples

1. Use the **Change Directory** command to change to the Struct1 directory.
2. Type the following:

```
csc struct1.cs
struct1
```

3. Use the **Change Directory** command to change to the Struct2 directory.
4. Type the following:

```
csc struct2.cs
struct2
```

See Also

Structs Tutorial | Tutorial Samples

Indexers Sample

This sample shows how C# classes can declare indexers to provide array-like access to the classes.

This sample contains the source code for the Indexers Tutorial.

Building and Running the Sample Within Visual Studio

To build and run the Indexers sample

1. Open the solution (Indexers.sln).
2. In **Solution Explorer**, right-click the indexers project and click **Properties**.
3. Open the Configuration Properties folder and click **Debugging**.
4. In the Command Line Arguments property, enter `..\..\Test.txt`.
5. Click **OK**.
6. From the **Debug** menu, click **Start Without Debugging**.

Building and Running the Sample from the Command Line

To build and run the Indexers sample

1. To compile the sample program, type the following at the command prompt:

```
csc indexer.cs
```

2. The sample program reverses the bytes in a file given as a command-line argument. For example, to reverse the bytes in Test.txt and see the result, issue the following commands:

```
indexer Test.txt
type Test.txt
```

3. To change the reversed file back to normal, run the program on the same file again.

See Also

Indexers Tutorial | Tutorial Samples

Indexed Properties Sample

This sample shows how C# classes can declare indexed properties to represent an array-like collection of different kinds of things.

This sample contains the source code for the Indexed Properties Tutorial.

Building and Running the Sample Within Visual Studio

To build and run the Indexed Properties sample

1. Open the solution (IndexedProperties.sln).
2. From the **Debug** menu, click **Start Without Debugging**.

Building and Running the Sample from the Command Line

To build and run the Indexed Properties sample

- Type the following at the command prompt:

```
csc indexedproperty.cs
indexedproperty
```

See Also

Indexed Properties Tutorial | Tutorial Samples

User-Defined Conversions Sample

This sample shows how to define conversions to or from classes or structures, and how such conversions are used.

This sample contains the source code for the User-Defined Conversions Tutorial.

Building and Running the Sample Within Visual Studio

To build and run the User-Defined Conversions samples

1. Open the solution (UserConversions.sln).
2. In **Solution Explorer**, right-click the Conversion1 project and click **Set as StartUp Project**.
3. From the **Debug** menu, click **Start Without Debugging**.
4. Repeat the preceding steps for Conversion2.

Building and Running the Sample from the Command Line

To build and run the User-Defined Conversions samples

1. Use the **Change Directory** command to change to the Conversion1 directory.
2. Type the following:

```
csc conversion.cs
conversion
```

3. Use the **Change Directory** command to change to the Conversion2 directory.
4. Type the following:

```
csc structconversion.cs
structconversion
```

See Also

User-Defined Conversions Tutorial | Tutorial Samples

Operator Overloading Sample

This sample demonstrates how user-defined classes can overload operators.

This sample contains the source code for the Operator Overloading Tutorial.

Building and Running the Sample Within Visual Studio

To build and run the Operator Overloading samples

1. Open the solution (OperatorOverloading.sln).
2. In **Solution Explorer**, right-click the Complex project and click **Set as StartUp Project**.
3. From the **Debug** menu, click **Start Without Debugging**.
4. Repeat the preceding steps for Dbbool.

Building and Running the Sample from the Command Line

To build and run the Operator Overloading samples

1. Use the **Change Directory** command to change to the Complex directory.
2. Type the following:

```
csc complex.cs
complex
```

3. Use the **Change Directory** command to change to the Dbbool directory.
4. Type the following:

```
csc dbbool.cs
dbbool
```

See Also

Operator Overloading Tutorial | Tutorial Samples

Delegates Sample

This sample demonstrates the delegate types. It shows how to map delegates to static and instance methods, and how to combine them (multicast).

This sample contains the source code for the Delegates Tutorial.

Building and Running the Sample Within Visual Studio

To build and run the Delegates samples

1. Open the solution (Delegates.sln).
2. In **Solution Explorer**, right-click the Delegates1 project and click **Set as StartUp Project**.
3. From the **Debug** menu, click **Start Without Debugging**.
4. Repeat the preceding steps for Delegates2.

Building and Running the Sample from the Command Line

To build and run the Delegates samples

1. Use the **Change Directory** command to change to the Delegates1 directory.
2. Type the following:

```
csc bookstore.cs
bookstore
```

3. Use the **Change Directory** command to change to the Delegates2 directory.
4. Type the following:

```
csc compose.cs
compose
```

See Also

Delegates Tutorial | Tutorial Samples

Events Sample

This sample shows how to declare, invoke, and hook up to events in C#.

This sample contains the source code for the Events Tutorial.

Building and Running the Sample Within Visual Studio

To build and run the Events samples

1. Open the solution (Events.sln).
2. In **Solution Explorer**, right-click the Events1 project and click **Set as StartUp Project**.
3. From the **Debug** menu, click **Start Without Debugging**.
4. Repeat the preceding steps for Events2.

Building and Running the Sample from the Command Line

To build and run the Events samples

1. Use the **Change Directory** command to change to the Events1 directory.
2. Type the following:

```
csc events1.cs
events1
```

3. Use the **Change Directory** command to change to the Events2 directory.
4. Type the following:

```
csc events2.cs
events2
```

See Also

Events Tutorial | Tutorial Samples

Explicit Interface Implementation Sample

This sample demonstrates how to explicitly implement interface members and how to access those members from the interface instances.

This sample contains the source code for the Explicit Interface Implementation Tutorial.

Building and Running the Sample Within Visual Studio

To build and run the Explicit Interface Implementation samples

1. Open the solution (ExplicitInterface.sln).
2. In **Solution Explorer**, right-click the ExplicitInterface1 project and click **Set as StartUp Project**.
3. From the **Debug** menu, click **Start Without Debugging**.
4. Repeat the preceding steps for ExplicitInterface2.

Building and Running the Sample from the Command Line

To build and run the Explicit Interface Implementation samples

1. Use the **Change Directory** command to change to the ExplicitInterface1 directory.
2. Type the following:

```
csc explicit1.cs
explicit1
```

3. Use the **Change Directory** command to change to the ExplicitInterface2 directory.
4. Type the following:

```
csc explicit2.cs
explicit2
```

See Also

Explicit Interface Implementation Tutorial | Tutorial Samples

Conditional Methods Sample

This sample demonstrates conditional methods, which provide a powerful mechanism by which calls to methods can be included or omitted depending on whether a symbol is defined.

This sample contains the source code for the Conditional Methods Tutorial.

Building and Running the Sample Within Visual Studio

To build and run the Conditional Methods sample

1. Open the solution (ConditionalMethods.sln).
2. In **Solution Explorer**, right-click the project and click **Properties**.
3. Open the Configuration Properties folder and click **Debugging**.
4. Set the Command Line Arguments property to "A B C" (without the quotation marks).
5. In the Configuration Properties folder, click **Build**.
6. Modify the Conditional Compilation Constants property (for example, add or delete DEBUG) and click **OK**.
7. From the **Debug** menu, click **Start Without Debugging**.

Building and Running the Sample from the Command Line

To build and run the Conditional Methods sample

- To include the conditional method, compile and run the sample program by typing the following at the command prompt:

```
csc CondMethod.cs tracetest.cs /d:DEBUG
tracetest A B C
```

See Also

Conditional Methods Tutorial | Tutorial Samples

XML Documentation Sample

This sample shows how to use XML to document code.

This sample contains the source code for the XML Documentation Tutorial.

Building and Running the Sample Within Visual Studio

To build the XML Documentation sample

1. Open the solution (XMLdoc.sln).
2. In **Solution Explorer**, right-click the project and click **Properties**.
3. Open the Configuration Properties folder and click **Build**.
4. Set the XML Documentation File property to XMLsample.xml.
5. From the **Build** menu, click **Build**. The XML output file will be in the debug directory.

Building and Running the Sample from the Command Line

To build the XML Documentation sample

1. To generate the sample XML documentation, type the following at the command prompt:

```
csc XMLsample.cs /doc:XMLsample.xml
```

2. To see the generated XML, issue the following command:

```
type XMLsample.xml
```

See Also

XML Documentation Tutorial | Tutorial Samples

Platform Invoke Sample

This sample demonstrates how to call platform invokes (exported DLL functions) from C#.

This sample contains the source code for the Platform Invoke Tutorial.

Building and Running the Sample Within Visual Studio

To build and run the Platform Invoke samples

1. Open the solution (Pinvoke.sln).
2. In **Solution Explorer**, right-click the PinvokeTest project and click **Set as StartUp Project**.
3. From the **Debug** menu, click **Start Without Debugging**.
4. Repeat the preceding steps for Marshal and Pinvoke.

Building and Running the Sample from the Command Line

To build and run the Platform Invoke samples

1. Use the **Change Directory** command to change to the PinvokeTest directory.
2. Type the following:

```
csc PinvokeTest.cs
PinvokeTest
```

3. Use the **Change Directory** command to change to the Marshal directory.
4. Type the following:

```
csc Marshal.cs
Marshal
```

5. Use the **Change Directory** command to change to the Pinvoke directory.
6. Type the following:

```
csc logfont.cs pinvoke.cs
pinvoke
```

See Also

Platform Invoke Tutorial | Tutorial Samples

COM Interop Part 1 Sample

This sample demonstrates how a C# program can interoperate with an unmanaged COM component.

This sample contains the source code for the COM Interop Part 1: C# Client Tutorial.

Building and Running the Sample Within Visual Studio

To build and run the COM Interop Part 1 samples

1. Open the solution (COMInteropPart1.sln).
2. In **Solution Explorer**, right-click the Interop1 project and click **Set as StartUp Project**.
3. In **Solution Explorer**, right-click the Interop1 project and click **Properties**.
4. Open the Configuration Properties folder and click **Debugging**.
5. In the Command-Line Arguments property, enter an AVI file such as `c:\winnt\clock.avi`.
6. Click **OK**.
7. From the **Debug** menu, click **Start Without Debugging**.
8. Repeat the preceding steps for Interop2.

If the QuartzTypeLib.dll included with Interop1 is out of date

1. In **Solution Explorer**, open **References** for Interop1.
2. Right-click **QuartzTypeLib** and click **Remove**.
3. Right-click **References** and click **Add Reference**.
4. In the **COM** tab, select the component named "ActiveMovie control type library."
5. Click **Select** and **OK**.
6. Rebuild Interop1.

   Note Adding a reference to the component does the same thing as invoking tlbimp at the command line to create QuartzTypeLib.dll (as shown below).

Building and Running the Sample from the Command Line

To build and run the COM Interop Part 1 samples

1. Use the **Change Directory** command to change to the Interop1 directory.
2. Type the following:

```
tlbimp %windir%\system32\quartz.dll /out:QuartzTypeLib.dll
csc /r:QuartzTypeLib.dll interop1.cs
interop1 %windir%\clock.avi
```

3. Use the **Change Directory** command to change to the Interop2 directory.
4. Type the following:

```
csc interop2.cs
interop2 %windir%\clock.avi
```

See Also

COM Interop Part 1: C# Client Tutorial | Tutorial Samples

COM Interop Part 2 Sample

This sample demonstrates using a C# server with a C++ COM client.

This sample contains the source code for the COM Interop Part 2: C# Server Tutorial.

> **Note** Visual C++ must be installed to compile this sample.

Building and Running the Sample Within Visual Studio

To build and run the COM Interop Part 2 sample

1. Open the solution (CSharpServerWithCOMClient.sln).
2. In **Solution Explorer**, right-click the COMClient project and click **Properties**.
3. Open the Configuration Properties folder and click **Debugging**.
4. In the Command Line Arguments property, enter a name.
5. Click **OK**.
6. From the **Debug** menu, click **Start Without Debugging**.

Building and Running the Sample from the Command Line

To build and run the COM Interop Part 2 sample

1. Use the **Change Directory** command to change to the COMInteropPart2\COMClient directory.
2. Copy the C# server code into the COMClient directory:

```
copy ..\CSharpServer\CSharpServer.cs
```

3. Compile the server:

```
csc /target:library CSharpServer.cs
regasm CSharpServer.dll /tlb:CSharpServer.tlb
```

4. Compile the client (make sure the path and environment variables are set properly with vcvars32.bat):

```
cl COMClient.cpp
```

5. Run the client:

```
COMClient friend
```

See Also

COM Interop Part 2: C# Server Tutorial | Tutorial Samples

Attributes Sample

This sample shows how to create custom attribute classes, use them in code, and query them through reflection.

This sample contains the source code for the Attributes Tutorial.

Building and Running the Sample Within Visual Studio

To build and run the Attributes sample

1. Open the solution (Attributes.sln).
2. From the **Debug** menu, click **Start Without Debugging**.

Building and Running the Sample from the Command Line

To build and run the Attributes sample

- Type the following at the command prompt:

```
csc AttributesTutorial.cs
AttributesTutorial
```

See Also

Attributes Tutorial | Tutorial Samples

Security Sample

This sample shows how to modify security permissions through permission classes and permission attributes.

This sample contains the source code for the Security Tutorial.

Building and Running the Sample Within Visual Studio

To build and run the Security samples

1. Open the solution (Security.sln).
2. In **Solution Explorer**, right-click the Security1 project and click **Set as StartUp Project**.
3. From the **Debug** menu, click **Start Without Debugging**.
4. Repeat the preceding steps for Security2 and Security3.

Building and Running the Sample from the Command Line

To build and run the Security samples

1. Use the **Change Directory** command to change to the Security1 directory.
2. Type the following:

```
csc ImperativeSecurity.cs
ImperativeSecurity
```

3. Use the **Change Directory** command to change to the Security2 directory.
4. Type the following:

```
csc DeclarativeSecurity.cs
DeclarativeSecurity
```

5. Use the **Change Directory** command to change to the Security3 directory.
6. Type the following:

```
csc SuppressSecurity.cs
SuppressSecurity
```

See Also

Security Tutorial | Tutorial Samples

Threading Sample

This sample demonstrates various thread activities:

- Creating and executing a thread.
- Thread synchronization.
- Interaction between threads.
- Using a thread pool.

This sample contains the source code for the Threading Tutorial.

Building and Running the Sample Within Visual Studio

To build and run the Threading samples

1. Open the solution (Threads.sln).
2. In **Solution Explorer**, right-click the StopJoin project and click **Set as StartUp Project**.
3. From the **Debug** menu, click **Start Without Debugging**.
4. Repeat the preceding steps for MonitorSample, SimplePool, and Mutex.

Building and Running the Sample from the Command Line

To build and run the Threading samples

1. Use the **Change Directory** command to change to the Threads directory.
2. Type the following:

```
cd StopJoin
csc StopJoin.cs
StopJoin
```

3. Type the following:

```
cd ..\MonitorSample
csc MonitorSample.cs
MonitorSample
```

4. Type the following:

```
cd ..\SimplePool
csc SimplePool.cs
SimplePool
```

5. Type the following:

```
cd ..\Mutex
csc Mutex.cs
Mutex
```

See Also

Threading Tutorial | Tutorial Samples

Unsafe Code Sample

This sample demonstrates how to use unmanaged code (code using pointers) in C#.

This sample contains the source code for the Unsafe Code Tutorial.

Building and Running the Sample Within Visual Studio

To build and run the Unsafe Code samples

1. Open the solution (Unsafe.sln).
2. In **Solution Explorer**, right-click the FastCopy project and click **Set as StartUp Project**.
3. From the **Debug** menu, click **Start Without Debugging**.
4. In **Solution Explorer**, right-click the ReadFile project and click **Set as StartUp Project**.
5. In **Solution Explorer**, right-click the ReadFile project and click **Properties**.
6. Open the Configuration Properties folder and click **Debugging**.
7. In the Command Line Arguments property, enter `..\..\ReadFile.cs`.
8. Click **OK**.
9. From the **Debug** menu, click **Start Without Debugging**.
10. In **Solution Explorer**, right-click the PrintVersion project and click **Set as StartUp Project**.
11. From the **Debug** menu, click **Start Without Debugging**.

Building and Running the Sample from the Command Line

To build and run the Unsafe Code samples

1. Use the **Change Directory** command to change to the Unsafe directory.
2. Type the following:

```
cd FastCopy
csc FastCopy.cs /unsafe
FastCopy
```

3. Type the following:

```
cd ..\ReadFile
csc ReadFile.cs /unsafe
ReadFile ReadFile.cs
```

4. Type the following:

```
cd ..\PrintVersion
csc PrintVersion.cs /unsafe
PrintVersion
```

See Also

Unsafe Code Tutorial | Tutorial Samples

OLE DB Sample

This sample demonstrates how to use a Microsoft Access database from C#. It shows how you can create a dataset and add tables to it from a database. The BugTypes.mdb database used in the sample program is a Microsoft Access 2000 .mdb file.

This sample contains the source code for the OLE DB Tutorial.

Building and Running the Sample Within Visual Studio

To build and run the OLE DB sample

1. Open the solution (OleDBSample.sln).
2. From the **Debug** menu, click **Start Without Debugging**.

Building and Running the Sample from the Command Line

To build and run the OLE DB sample

- Type the following at the command prompt:

```
csc oledbsample.cs
oledbsample
```

See Also

OLE DB Tutorial | Tutorial Samples

Cross-Language Samples

The following table provide links to samples that demonstrate cross-language interoperability between Visual C# and Visual C++.

Sample	Description
DataSetConsumer	Demonstrates how to consume multiple-table DataSets in a SPROXY-generated SOAP client. The Visual C# code creates a multiple-table DataSet and exposes it through a Web method.
ManagedEvents	Demonstrates event handling interoperability between Managed Extensions for C++ and Visual C# objects. The Visual C# code implements both an event source and an event receiver.
JrnlPost	Creates a single DLL containing native business logic, an interface and type library for COM clients, and an assembly with corresponding metadata for .NET Framework clients. The Visual C# code instantiates a managed object from the DLL and calls a couple of methods on it. It illustrates that the DLL contains both COM and .NET Framework classes.
MCppWrapper	Illustrates how a Managed Extensions __gc class can be used as a proxy class (or wrapper) for an unmanaged C++ class in a DLL. The Visual C# code is a client that reads a string and position from the console and displays a suffix.
TilePuzzle	An interactive sliding tile puzzle that demonstrates the interoperability of managed and unmanaged components. The Visual C# code provides the user interface to the tile-puzzle board. This component interfaces with the TileDriver Managed Extensions for C++ component to find valid states when a user tries to move a tile. It also calls into the algorithm engine when solving a puzzle.
CustomAttributes	Demonstrates implementing a custom attribute. Custom attributes can be used to create type-checked user-defined metadata. The Visual C# custom attribute adds several different types of information to the metadata of the target class and defines several different parameter lists for usage.
Delegates	Demonstrates custom delegates in a managed application. The Visual C# DLL in the sample implements a basic Visual C# class (`CDel`) with a custom method.
OnlineAddressBook	Stores an address book in an XML Web service created with ATL Server. The sample has a Visual C# client that calls into the XML Web service.
PlaneParts	Demonstrates how the various technologies available in Visual Studio .NET can be used together in an enterprise application. Visual C# is used for the Web Forms and to create a Windows Service.

See Also

Visual C# Samples

Tailspin Toys Application

The Tailspin Toys application demonstrates how the various technologies available in Visual Studio .NET can be used together in an Enterprise application. The application demonstrates a simple transacted ordering system internal to a company, arbitrarily chosen as airplane parts.

The application is a multitier application with a browser client, a Web server, server components, and a database tier. In addition, business-to-business (B2B) communication with an external vendor that accepts orders via an XML Web service is demonstrated.

Installing Tailspin Toys

The Tailspin Toys application can be found at Samples\VCSharp\General\Tailspin on the MSDN CD (disk 1) or DVD. It is packaged as a self-extracting executable file containing two installation files (.msi) that contain Visual Studio solutions and accompanying documentation.

Description

The application consists of ATL Server-based Web applications used for ordering parts and for fulfilling orders from a warehouse, a message-queue–based order-handling system written in Visual C# running as a Windows Service and using serialization to convert XML messages to and from objects, a SQL 2000 database using SQL XML and OLE DB, a transactional accounting system based on Visual C# ServicedComponents, a Visual C# Web Forms application, and business-to-business communication using XML Web services with a separate vendor site.

Features

Features demonstrated include:

- ATL Server Web application using database session management
- Visual C# Web Forms application using ADO.NET
- Transacted, stateless account management using Visual C# ServicedComponents and ADO.NET
- Use of MSMQ to implement an order-handling system
- Use of Visual C# Windows Service to process MSMQ messages
- Use of MSXML Document Object Model (DOM)
- XML Web services used for business-to-business communication
- COM Interop from Visual C#
- Use of SQL 2000 XML support via OLE DB
- Interop of native COM+ and managed ServicedComponents
- Serialization of common language runtime objects as XML
- Secure logins using password encryption

See Also

Visual C# Samples

QuickStarts and Other Samples

The .NET Framework SDK includes QuickStart tutorials and other samples that demonstrate .NET Framework features and Visual C# code. For more information, see the following topics in the .NET Framework SDK:

- Samples
- QuickStarts

Samples

If you install the samples with the .NET Framework SDK, you can find them at the following path:

\Program Files\Microsoft Visual Studio .NET 2003\FrameworkSDK\Samples

If you do not want to install the samples on your computer, you can browse the Samples folder in the .NET Framework SDK installation directory on the product CD or DVD.

QuickStarts

The QuickStart Tutorials are the fastest way to understand what the .NET Framework technology offers leading-edge developers. The QuickStarts are a series of samples and supporting documentation designed to quickly acquaint you with the syntax, architecture, and power of Visual Studio .NET and the .NET Framework.

To access the QuickStarts

1. Click the Windows **Start** button.
2. Point to **Programs**.
3. Point to **Microsoft .NET Framework SDK version 1.1**.
4. Click **Samples and QuickStart Tutorials**.

   The main page of the "Microsoft .NET Framework SDK QuickStarts, Tutorials and Samples" application will appear.

To run the samples, follow the directions to install the samples database and complete the installation. After you have installed the QuickStarts, the "QuickStarts Tutorials" link will be live.

The QuickStart tutorials include samples on the ASP.NET and Windows Forms applications, in addition to many how-to applications that cover the most compelling features of the .NET Framework technology.

See Also

Visual C# Samples